# DELAYED CODE GENERATION IN A SMALLTALK-80 COMPILER

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE

October 1992

By
Ian K. Piumarta
Department of Computer Science

# Contents

**Abstract** 9

**Acknowledgements** 12

**1 Introduction** 13
- 1.1 Scope of this Thesis . . . 14
- 1.2 Overview of this Thesis . . . 14
- 1.3 Prerequisites . . . 15

**2 Fundamentals** 17
- 2.1 Smalltalk-80 . . . 17
  - 2.1.1 Objects, Classes, Messages and Inheritance . . . 17
  - 2.1.2 Deferred Execution: Blocks . . . 20
  - 2.1.3 Execution Flow Control . . . 21
- 2.2 Compilation . . . 21
  - 2.2.1 Compiler Anatomy . . . 22
  - 2.2.2 Target Languages . . . 22
- 2.3 Summary . . . 23

**3 Related Work** 24
- 3.1 A Proprietary Smalltalk: ParcPlace PS2.3 . . . 24
  - 3.1.1 The Virtual Machine . . . 25
    - 3.1.1.1 Dynamic Method Translation . . . 26
    - 3.1.1.2 Multiple Representation of Contexts . . . 26
    - 3.1.1.3 Caches . . . 27
  - 3.1.2 The Compiler . . . 28
- 3.2 Orthogonal Code Generation . . . 29
  - 3.2.1 Data Representation . . . 29
  - 3.2.2 Operations on Data Descriptors . . . 30
  - 3.2.3 Data Descriptors . . . 30
  - 3.2.4 Strengths and Weaknesses of Data Descriptor Techniques . . . 32
- 3.3 Optimizing Compilers for Smalltalk-80 . . . 33
  - 3.3.1 Typed Smalltalk . . . 33
- 3.4 Summary . . . 34

**4 Benchmarks** **35**
4.1 Benchmarking Smalltalk-80 . . . . . . . . . . . . . . . . . . . . . . . 36
4.1.1 Benchmark Framework . . . . . . . . . . . . . . . . . . . . . . 36
4.1.2 Micro-Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . 37
4.1.3 Macro-Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . 38
4.2 Benchmark Characteristics . . . . . . . . . . . . . . . . . . . . . . . 39
4.3 Fairness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

**5 68020 Native Code Smalltalk-80** **42**
5.1 Runtime Environment and Conventions . . . . . . . . . . . . . . . . 42
5.1.1 Object Memory and Object Format . . . . . . . . . . . . . . . 43
5.1.2 Register Usage . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.1.3 Stack Discipline . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.1.4 Runtime Support . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.1.5 Omissions in the Runtime System . . . . . . . . . . . . . . . . 49
5.2 Garbage Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.1 Garbage Collection and the Stack . . . . . . . . . . . . . . . . 50
5.3 The Compiler Front End . . . . . . . . . . . . . . . . . . . . . . . . 51
5.3.1 Scanning and Parsing . . . . . . . . . . . . . . . . . . . . . . . 51
5.3.2 The Parse Tree . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.3.3 Optimizing the Parse Tree . . . . . . . . . . . . . . . . . . . . 52
5.4 Code Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.4.1 Overview of Code Generation . . . . . . . . . . . . . . . . . . 53
5.4.2 Machine Objects: the M68000 . . . . . . . . . . . . . . . . . . 54
5.4.3 Code for Leaf Nodes . . . . . . . . . . . . . . . . . . . . . . . 56
5.4.4 Assignment Statements . . . . . . . . . . . . . . . . . . . . . . 58
5.4.5 Message Sends . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.4.6 Method Entry and Exit . . . . . . . . . . . . . . . . . . . . . . 60
5.4.7 Primitive Methods . . . . . . . . . . . . . . . . . . . . . . . . 61
5.4.8 Code for Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . 62
5.4.9 Block Problems . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.5 Optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.5.1 Inlining Special Selectors . . . . . . . . . . . . . . . . . . . . 67
5.5.2 Inlining Control Selectors . . . . . . . . . . . . . . . . . . . . 70
5.5.3 Peephole Optimizations . . . . . . . . . . . . . . . . . . . . . . 72
5.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

**6 Delayed Code Generation** **74**
6.1 Shortcomings of the Naïve Code Generator . . . . . . . . . . . . . . 75
6.2 Classifying the Problem . . . . . . . . . . . . . . . . . . . . . . . . 78
6.3 Operand Descriptors . . . . . . . . . . . . . . . . . . . . . . . . . . 79
6.3.1 Representation of Operand Descriptors . . . . . . . . . . . . . 80
6.4 Code Generation with Operand Descriptors . . . . . . . . . . . . . . 81

6.4.1 Operand Descriptors generated at Leaf Nodes . . . . . . . . . 81
6.4.2 Assignment . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.4.3 Message Sends . . . . . . . . . . . . . . . . . . . . . . . . 86
6.4.4 Method Entry and Exit . . . . . . . . . . . . . . . . . . . . 87
6.4.5 Primitives and Blocks . . . . . . . . . . . . . . . . . . . . 88
6.5 Optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
6.5.1 Inlined Special Selectors . . . . . . . . . . . . . . . . . . . 89
6.5.1.1 Arithmetic Operations . . . . . . . . . . . . . . . . . 89
6.5.1.2 Relational Operations . . . . . . . . . . . . . . . . . 93
6.5.2 Inlined Control Constructs . . . . . . . . . . . . . . . . . . . 94
6.5.2.1 Control Flow Forking . . . . . . . . . . . . . . . . . 94
6.5.2.2 Deferred Message Sends Revisited . . . . . . . . . 95
6.5.2.3 Code for Conditionals . . . . . . . . . . . . . . . . . 97
6.5.2.4 Code for Loops . . . . . . . . . . . . . . . . . . . . 99
6.5.3 Other Optimizations . . . . . . . . . . . . . . . . . . . . . . 100
6.6 Code Generation for Other Languages . . . . . . . . . . . . . . . . 100
6.6.1 Operand Descriptors for Other Addressing Modes . . . . . . 100
6.6.2 Operand Sizing . . . . . . . . . . . . . . . . . . . . . . . . . 101
6.6.3 Logical Values . . . . . . . . . . . . . . . . . . . . . . . . . 101
6.6.4 Coercion of Operand Types . . . . . . . . . . . . . . . . . . 103
6.6.5 Operands with Side Effects . . . . . . . . . . . . . . . . . . 105
6.6.6 Argument Order . . . . . . . . . . . . . . . . . . . . . . . . 105
6.6.7 Register Allocation . . . . . . . . . . . . . . . . . . . . . . . 106
6.6.7.1 Machine Models and Allocation Strategy . . . . . . 106
6.6.7.2 Register Allocator Performance . . . . . . . . . . . 108
6.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

**7 Results** **112**
7.1 A Brief Note Concerning Definitions . . . . . . . . . . . . . . . . . 113
7.2 Performance of Generated Code . . . . . . . . . . . . . . . . . . . 114
7.3 Compiler Efficiency . . . . . . . . . . . . . . . . . . . . . . . . . . 114
7.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

**8 Conclusion** **117**
8.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117

**A Parse Tree Nodes** **119**
A.1 Leaf Nodes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
A.2 Message Nodes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
A.3 Special Action Nodes . . . . . . . . . . . . . . . . . . . . . . . . . 120
A.4 Method and Block Nodes . . . . . . . . . . . . . . . . . . . . . . . 120

**B Raw Results** 123
B.1 Unexpected Results . . . . . 132
B.2 Absolute Performance of the DCG Compiler . . . . . 132

**C Assembly Language Conventions** 133

**D Examples** 134
D.1 Conditional Statement . . . . . 134
D.2 String Copy Loop . . . . . 134
D.3 Function Call . . . . . 135
D.4 Generated Code For `nfib` . . . . . 136

**E Further Implementation Notes** 139
E.1 Further Improvements in the Generated Code . . . . . 139
E.1.1 Shared Deferred Sends . . . . . 139
E.1.2 Better Use of Class Information . . . . . 140
E.2 Support for Debugging . . . . . 140
E.2.1 Failed Message Sends . . . . . 140
E.3 Inline Caches . . . . . 141
E.3.1 Inline Cache Design . . . . . 142
E.3.2 Example Inline Cache . . . . . 143
E.3.3 Interference with Deferred Sends . . . . . 145
E.3.4 Cache Consistency . . . . . 145

**Bibliography** 147

# List of Tables

7.1 Relative Performance of Naïve and Delayed Code Generators . . . . 114
7.2 Compilation Times for Naïve and Delayed Code Generation . . . . . 115

B.1 Naïve Code Execution Time. . . . . . . . . . . . . . . . . . . . . . . . 125
B.2 DCG Code Execution Time. . . . . . . . . . . . . . . . . . . . . . . . 126
B.3 PS2.3 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
B.4 Benchmark Message Send Activity . . . . . . . . . . . . . . . . . . . 128
B.5 Benchmark Primitive Call Activity . . . . . . . . . . . . . . . . . . . . 129
B.6 Naïve Code Memory Utilization . . . . . . . . . . . . . . . . . . . . . 130
B.7 DCG Code Memory Utilization . . . . . . . . . . . . . . . . . . . . . 131
B.8 Comparison Between PS2.3 and Native Code Smalltalk-80 . . . . . . 132

# List of Figures


2.1 Structural and Functional Inheritance . . . . . . . . . . . . . . . . . . 19

3.1 Special Selectors . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.2 Example Data Descriptors . . . . . . . . . . . . . . . . . . . . . . . 30
3.3 Addition Operations on Data Descriptors . . . . . . . . . . . . . . . . 31

4.1 Messages Sent from Benchmarks . . . . . . . . . . . . . . . . . . . . . 39

5.1 Object Table Format . . . . . . . . . . . . . . . . . . . . . . . . . 43
5.2 Address Register Assignments . . . . . . . . . . . . . . . . . . . . . 45
5.3 Format of Stack Frames . . . . . . . . . . . . . . . . . . . . . . . . 47
5.4 Runtime Support Dispatch Table . . . . . . . . . . . . . . . . . . . . 48
5.5 Compiler Structure . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.6 Class Hierarchy for 68000 Operands . . . . . . . . . . . . . . . . . . 55
5.7 Hierarchy for 68000 Instructions . . . . . . . . . . . . . . . . . . . 55
5.8 The Primitive Call Mechanism . . . . . . . . . . . . . . . . . . . . . 62

6.1 Operand Descriptors Generated at Leaf Nodes . . . . . . . . . . . . . 82
6.2 Parse Tree for Assignment . . . . . . . . . . . . . . . . . . . . . . . 86
6.3 Definition of ‘emitInlinedBinary:’ . . . . . . . . . . . . . . . . . . . 90
6.4 Definition of ‘emitInlinedOp’ . . . . . . . . . . . . . . . . . . . . . 92
6.5 Control Constructs via Forking . . . . . . . . . . . . . . . . . . . . 94
6.6 Definition of ‘emitIfTrue’ . . . . . . . . . . . . . . . . . . . . . . . 98
6.7 Register Allocation and Deallocation . . . . . . . . . . . . . . . . . . 107
6.8 Graph Coloring Example . . . . . . . . . . . . . . . . . . . . . . . . 109
6.9 Parse Tree for Graph Coloring Example . . . . . . . . . . . . . . . . 109
6.10 Generated Code for Graph Coloring Example . . . . . . . . . . . . . 110

A.1 Structure of Leaf Nodes . . . . . . . . . . . . . . . . . . . . . . . . 120
A.2 Structure of Message Nodes . . . . . . . . . . . . . . . . . . . . . . 121
A.3 Structure of Return and Assignment Nodes . . . . . . . . . . . . . . 121
A.4 Structure of Block and Method Nodes . . . . . . . . . . . . . . . . . 122

B.1 Key to Benchmark Class Names . . . . . . . . . . . . . . . . . . . . 124
B.2 Key to Benchmark Optimization Names . . . . . . . . . . . . . . . . . 124

D.1 If Statement Example . . . . . . . . . . . . . . . . . . . . . . . . . 135

D.2 String Copy Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136
D.3 Function Call Example . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

# Abstract

More than any other programming system, Smalltalk-80 stretches the object-oriented paradigm to its limits. Representing all programmer-accessible data (input and output facilities, contexts, processes, functions, and so on) as objects is the cause of many implementation difficulties. Polymorphism, the dynamic binding of function names to function bodies at runtime, and the transparent management of dynamic memory allocation only aggravate the situation further.

Traditional implementations (in other words, all the commercially available implementations) try to narrow the semantic gap between the language and the platform upon which it runs by compiling Smalltalk for an idealized virtual machine that uses simple language-oriented instructions. This approach has advantages for both the compiler writer (the target language is optimized for running programs written in the source language) and for the runtime system (compiled code is small and easy to map back to the source for debugging). Reducing the complexity of the compiler also speeds up compilation, which is highly desirable in exploratory programming environments such as Smalltalk-80. The down side is that reducing the complexity of the compiler and target language causes a corresponding increase in complexity in the runtime system which has to work much harder if it is to ensure efficient execution of code.

This thesis argues and demonstrates that it is possible to compile Smalltalk-80 directly into machine code for stock hardware, and to do this efficiently in terms of both compiler performance (the compiler must be small and fast) and generated code performance. The techniques developed for 'delayed code generation' in single-pass recursive descent compilation (or code generation by walking a parse tree) are applicable to almost any language, and some discussion of the application of delayed code generation to the compilation of C is also presented.

Some investigation into the applicability and effectiveness of various compile- and run-time optimizations is presented, quantified by benchmark results covering a wide range of activities from critical operations (such as addition) to entire subsystems (such as the text display interface).

For my parents.

# DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

# Acknowledgements

First and foremost I thank my supervisor Dr. T. P. Hopkins for introducing me to object-oriented ideas, and providing the help and encouragement that made this work possible.

For efforts in many orthogonal directions far above and beyond the call of duty, Dr. Mario Wolczko deserves praise. He has conversed with me frequently on the knottier problems of Smalltalk implementation, and more recently on the subtleties of its compilation. He has kept the machines of the MUSHROOM group running smoothly for the duration of the project, and responded quickly and efficiently to my relentless queries on many mundane topics — in particular the constant battle to coerce LaTeX into serving me, rather than the reverse. I must also thank the other members of the MUSHROOM group for a congenial atmosphere in which to work.

Thanks are due to Eliot Miranda for his many useful comments and suggestions regarding changes and additions to the original thesis for this technical report.

Special thanks are due to the Medical Informatics Group at the University of Manchester for the provision of computing facilities, office space and a salary while putting the finishing touches to this thesis.

The work described in this thesis was supported, in part, by the U.K. Science and Engineering Research Council.

**Colophon**
This thesis was typeset in 12pt Times Roman using the LaTeX document preparation system, and printed on a Sun SparcPrinter. All diagrams were prepared using FrameMaker.

# Chapter 1

# Introduction

> The change which came with the scientific age was not, whatever some anthropologists may say, the introduction of a new way of thinking, but the rigorous and exclusive use of an old one. That this has meant enormous alterations in our attitudes and in our mythology no one would deny; but in the broad sense the 'scientific method' has always existed. To collect evidence by observation, to generalize from your information, and then to test your general pattern by prediction and further observation, is not a procedure invented by Western Man since the Renaissance; it is the activity which made all human civilization possible.
>
> J. Austin Baker, *The Foolishness of God*, 1970.

Smalltalk-80 [GR83] is the archetypal object-oriented "Explorat-ory Programming Environment" (EPE). It provides a graphical user interface based on multiple overlapping windows in a bitmapped display for output, and a mouse/pointer and keyboard for input. A complete graphical environment is provided that supports the system's user interface and provides facilities for the user to painlessly create new user interfaces for applications. An interactive, incremental compiler is provided for altering or adding behavior to the system, the effects of which are felt immediately; this kind of dynamic behavior is essential in an EPE.

Most EPEs are Lisp-based, and code is usually interpreted whilest under development. This is the case for several reasons. The frequent recompilation of single functions by typically large compilers would slow down development work; leaving the code interpreted removes any delay between entering the code and testing it. Diagnostics are usually better for interpreted code too, typically providing the user with backtracking facilities and the ability to interrupt the program and inspect the code being executed at any time. The price paid for these features is the relatively inefficient execution of code while it is under development. Compilation normally only takes place when a fairly large block of definitions has stabilized, at which time the whole block is compiled.

Smalltalk-80 takes a different approach. Code is never interpreted, but is always recompiled if it is altered; compilation takes place as soon as a change is 'accepted'. This has several implications. First, code that is altered runs at full speed immediately

after the alteration is accepted. Second, the compiler must be small, otherwise there would be annoying delays both during compilation while the user interface is paged out and the compiler paged in, and after compilation while the compiler is paged out and the interface paged back in. Third, the compiler must be fast. Smalltalk-80 applications are typically made of a large number of functions, or 'methods', each of which is compiled independently of the others. Since compilation is an interactive process in Smalltalk-80, the faster the compilation of each method the better.

The above considerations, plus the fact that the Smalltalk-80 language has a fairly simple LL(1) grammar, means that top-down recursive descent is an ideal technique for its compilation. Such compilers are small since they do not need the large parsing tables found in compilers for the more complex LR grammars. This means they will not have an unreasonable impact on the size of the Smalltalk-80 system. Recursive descent compilers are also very fast, since they generally employ only one or two passes to produce the final object code; they are therefore suited to an interactive environment where any severe delays in the compilation of a method are unacceptable.

## 1.1 Scope of this Thesis

The work described in this thesis grew from an investigation into the compilation of Smalltalk-80 into native code for the MUSHROOM [Wil89][HWW87] hardware. After a few weeks struggling with a very slow simulator for this machine, I decided to produce code for the MC68020-based Sun3 workstation instead, intending to slot a MUSHROOM back end into the compiler at a later stage. Although these target architectures are very different, the facets of compilation and code generation for Smalltalk-80 that I wished to investigate were broadly the same.

The compiler evolved steadily from a very simple recursive descent approach that performed no optimization and, as it grew, analyses on the performance gains made from particular optimizations were undertaken as a matter of course. These results are presented herein as part of the discussion of the effectiveness of the more usual Smalltalk-80 optimizations, but applied in a native code environment.

The most novel aspect of the mature compiler is the code generation technique which was developed in response to the poor quality of the code produced by the initial rather naïve code generator. While developing the novel code generator it became obvious that the techniques used could be applied to languages other than Smalltalk-80. Indeed, the techniques are applicable to any tree-walking code generator (or single-pass recursive descent compiler) for almost any language. Some investigation of the application of the techniques in the compilation of C was therefore undertaken.

## 1.2 Overview of this Thesis

The first part of this thesis presents some groundwork needed to tackle the later sections. I describe the features of Smalltalk-80 that set it apart from other languages,

and the implications that these features have for the compiler writer.

Chapter 3 describes the best commercial implementation that was available at the time this work was started, namely ParcPlace Systems' Smalltalk-80 version 2.3 (hereafter referred to as PS2.3). The historical influences on its design and the various approaches to improving its efficiency are discussed. It is this system which is used as a performance model against which to test the efficiency of the native code system. Some other related implementations of Smalltalk-80 and Smalltalk-80-like languages that have novel features to improve efficiency are also briefly described. This chapter also outlines the work that has been undertaken on the use of 'data descriptors' in compilers for languages such as PL/I and concurrent Euclid. The data-descriptor based code generation techniques employed by compilers for these languages are similar in some ways to the techniques developed for the novel Smalltalk-80 native code generator.

Following this in chapter 4 is a discussion of benchmarking in general: what we do and do not want benchmarks to do for us. I describe the benchmarks that I chose to meter both the performance of PS2.3 and of the Native Code Smalltalk-80 system. Several different types of benchmark are described, each of which has its own advantages in measuring a particular aspect of the system's performance.

The main part of the thesis begins with chapter 5, a description of the MC68020 native code implementation and the compiler developed for it. Some compilation techniques that are independent of the code generator are presented before a discussion of a typical 'naïve' code generator.

Chapter 6 continues by firstly exposing the problems with this naïve code generator, and then developing a novel code generation technique to produce code which does not suffer from these problems. Wider applications of the novel code generator are then investigated and the compilation of C [KR78] (a language very different from Smalltalk-80) which extends the technique to cover register allocation is discussed.

The final part of the thesis, chapters 7 and 8, presents the experimental results of the work, draws some conclusions about the success of the novel code generator, and presents ideas for future work.

## 1.3 Prerequisites

Throughout this thesis it is assumed that the reader has a basic knowledge of the Smalltalk-80 language and its concepts. A minimum requirement for non-Smalltalk-80 programmers would be familiarity with [GR83, part one].

Some knowledge of recursive descent compilation is also assumed; a suitable introduction to this can be found in [Bor79] and [ASU86]. Aspects of Smalltalk-80 and compilation techniques not covered in these works will be explained in the body of the thesis where necessary.

Familiarity with 68020 assembly language is necessary for the reader to make any sense of the chapters on compilation. The mnemonic instruction names and operand notation used are those defined by Sun Microsystems; these are explained in detail

in [Sun88], and those already familiar with a dialect of 68020 assembly language should have little difficulty in understanding the notation. A more thorough treatment of the operations and addressing capabilities of the 68020 can be found in Motorola's own documentation [Mot85], which uses the 'traditional' Motorola mnemonics and addressing mode notation. If in doubt, consult appendix C which summarizes the addressing mode notation used in this thesis, in both Sun and Motorola formats.

# Chapter 2

# Fundamentals

> Nor have we brought into this work any graces of rhetoric, any verbal ornateness, but have aimed simply at treating knotty questions about which little is known in such a style and in such terms as are needed to make what is said clearly intelligible. Therefore we sometimes employ words new and unheard-of, not (as alchemists are wont to do) in order to veil things with a pedantic terminology and to make them dark and obscure, but in order that hidden things which have no name and that have never come into notice, may be plainly and fully published…
>
> William Gilbert, *De Magnete*, 1600.

This chapter describes very briefly the salient features of Small- talk-80, highlighting the important differences between it and more conventional languages. Following this is a short review of compilation techniques.

## 2.1 Smalltalk-80

The following should not be taken as a complete description of Smalltalk-80, but rather as a concise and explicit declaration of the terminology that will be used to describe the language features that are relevant to its compilation. In particular, little of the concrete syntax of the language is presented — such details are explained in [GR83].

Some of Smalltalk-80's features have a profound effect on its performance and the techniques that can be applied in its compilation. Such relevant features are introduced here also, although implementation-specific details are ignored; these will be dealt with in a later chapter.

### 2.1.1 Objects, Classes, Messages and Inheritance

In Smalltalk-80 a unit of information is called an *object*. Everything from a simple number, through 2-dimensional points and arrays, to the structure of a paragraph with

font changes are all represented as objects. Each object belongs to a *class*, which describes the physical structure of its objects and their functionality. Since functionality is described by an object's class, all objects of a given class behave identically. We refer to an object of a particular class as an *instance* of that class.

Computation in Smalltalk-80 is effected by sending *messages* to objects. A message is a request to an object to perform a particular task, however simple. A simple message might request a 2-dimensional point (an instance of class Point[1]) to return its abscissa. The instance of Point that receives the message is called the *receiver*, and the name of the message is called the *selector*.

Classes contain a *dictionary* whose keys are the message selectors to which their instances will respond. The values are pieces of executable code called *methods*. When an object receives a message this *method dictionary* in the object's class is searched to find the method corresponding to the message selector, and the matching method is then executed. The set of messages to which an object can respond is called the *protocol* of the object.

Methods can refer to their receiver's state by one of two mechanisms. The fields of the object that contain its state are either *instance variables* which are referred to in a method by name, or *indexed variables* which are referred to by an offset. The instance variables of an object are called the *fixed fields*, and its indexed variables (if any) are called the *indexed fields*. By writing the name of an instance variable in a method, that variable of the receiver can be accessed.

Because instance variables can only be directly accessed by the methods of their own class and not from methods outside their class, objects provide *encapsulation* and *information hiding*. It is impossible (for example) for a method executing in class Rectangle to alter the abscissa of the Point representing the rectangle's origin without explicitly sending the point a message; this feature allows objects absolute control over the ways in which they can be manipulated. Indeed, in the case of Points, it also hides their implementation: they are in fact held as rectangular co-ordinates and provide for polar style access through more complicated methods that convert from the rectangular representation. They could equally well be implemented in polar co-ordinates directly, and have the normal rectangular accesses processed by more complicated methods that convert from the polar representation, without affecting the operation of the system at all.

Classes are arranged in a tree called the *class hierarchy* which provides both *structural inheritance* and *functional inheritance*. Given a class in this tree, the class immediately above it (from which it inherits) is its *superclass*; any classes immediately below it (which inherit from it) are its *subclasses*. Structural inheritance means that a subclass $B$ of a particular class $A$ inherits the instance variables of $A$; any instance variables defined in $B$ are appended to the variables of $A$. Similarly, functional inheritance means that messages defined for class $A$ can also be understood by instances of $B$. If $B$ defines any methods with the selectors in common with methods defined in $A$,

[1] This thesis follows the standard convention of setting Smalltalk-80 class names, message selectors, and pieces of Smalltalk-80 code itself in sans-serif type.

then these new definitions *override* the original definitions (figure 2.1).

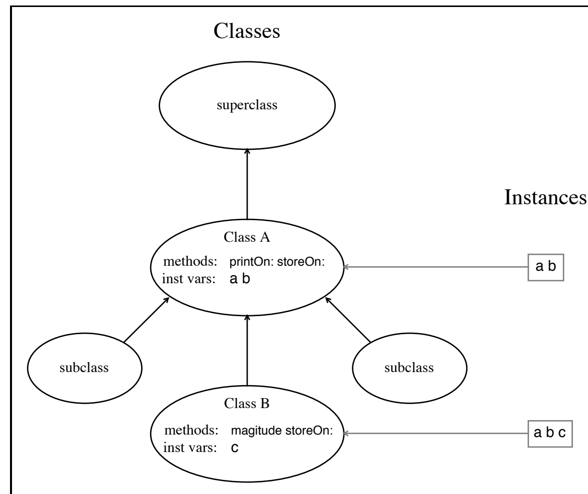


**Figure 2.1:** Structural and functional inheritance. Class *A* defines two instance variables 'a' and 'b'. These are inherited by class *B* which also adds a third instance variable 'c'.
Class *A* defines two methods, 'printOn:' and 'storeOn:'. Instances of *A* only respond to these two messages. Class *B* inherits these methods from *A*, but adds an extra method 'magnitude' and overrides the 'storeOn:' method. Instances of *B* respond to these two new methods in addition to the 'printOn:' method inherited from *A*.

---

Functional inheritance provides a simple and effective means of *code reuse*, an important consideration in a system as large as Smalltalk-80. The method that finds (for example) the maximum of a pair of Integers is exactly the same method as is invoked for finding the maximum of a pair of Dates or Times. Since these different classes of object are all kinds of Magnitude, they inherit the methods that rely solely on the magnitude of an object from their common superclass, thus obviating the need for duplicated code for finding maxima (and other operations based on some sort of ordering) in each of the separate classes.

Classes such as Magnitude are not designed to be instantiated. They provide the common functionality of a few distinct *concrete* subclasses that *can* be instantiated, but in themselves do not provide enough state and/or functionality to be useful. Such classes are called *abstract superclasses*. At the very top of the class hierarchy is an abstract superclass called Object that provides the protocol common to every object in the system.

When a message is sent to an object, the method dictionary in that object's class is interrogated to find the message selector. If the selector is found, the indicated method is executed. If the selector is not found then the search is restarted in the object's superclass; this mechanism, called *method lookup* implements functional inheritance. The process is continued in each class up the superclass chain until the relevant method

is found. If the method is not found in class Object, then the message is not understood by that object, and an error is reported by the system.

Methods can refer to several *pseudo variables* that are associated with special values. Within a method, the variable 'self' refers to the receiver of the message that caused the method to run. This allows messages to be sent to the receiver, or for the receiver to be passed as an argument. The message send is perfectly normal, with the method lookup beginning in the class of the receiver. The variable 'super' also refers to the receiver, and messages sent to this pseudo variable have the current receiver as the new receiver. However, this send differs from a send to 'self' in that the method lookup does not begin in the class of the receiver, rather it begins in the superclass of the class containing the method originating the send.[2]

Classes themselves are objects, and are therefore instances of a class. Each class has a *metaclass* that describes the class and the messages to which the class can respond. (Instantiating a metaclass yields a class, and there is only ever one instance of a particular metaclass [GR83, pages 269-272].) Many of these messages will be concerned with the creation of instances. To differentiate these methods from 'normal' methods, they are referred to as *class methods*; to reduce confusion further, 'normal' methods are sometimes called *instance methods* in situations where ambiguity could arise. As far as the compiler is concerned, there is no difference between compiling instance and class methods: instance methods are compiled 'in the context of' a class, and class methods are compiled in the context of the corresponding metaclass.

### 2.1.2 Deferred Execution: Blocks

Blocks are Smalltalk-80's version of closures, similar to Lisp's lambdas [McC60] [McC62]. They are first-class objects representing pieces of unevaluated code that can be executed at an arbitrary time.

Blocks can appear in any place in which an expression is valid. They are fully fledged objects, of class BlockContext, and can be the receivers of, or arguments to, messages. Syntactically they are a normal sequence of statements surrounded by square brackets; for example, a block that adds one to the variable 'a' is written

```
[a ← a + 1]
```

and executes when it receives the message 'value'.

Blocks can also have arguments. For example, a block to add an arbitrary value to the variable 'a' is written

```
[:increment | a ← a + increment]
```

and executes when it receives a message like 'value: 42'.

The Smalltalk-80 standard defines the arguments of a block as being shared with the method in which the block originates. This means that a temporary variable 'temp'

[2]Sends to 'super' are provided so that an overridden method can be called from within the overriding method.

in a method can be set to the value 42 by sending the block '[:temp]' the message 'value: 42' which is not as clean an implementation of closures as is found in languages such as scheme [Dyb87], although this 'feature' has been used in proprietary code. This problem has been fixed in versions 2.5 and later of the ParcPlace implementations, where space for arguments is private to the block itself.

### 2.1.3 Execution Flow Control

Smalltalk-80 provides two types of flow control: conditional execution and looping. Conditional execution is provided by complementary definitions in the two boolean classes, and looping is provided by a combination of conditional execution and recursion.

Conditional methods are defined in the two classes True and False which have identical protocols. Such methods expect blocks as arguments, since these represent deferred code that can be executed optionally. The definitions for a particular conditional message are complementary in that a method in True that causes its block argument to be executed has a corresponding method in False that returns 'nil' without executing its block argument. For example, in class True we have

```
ifTrue: aBlock
    ↑aBlock value
```

and in class False we have the complementary definition

```
ifTrue: aBlock
    ↑nil
```

Looping is provided by conditional recursion in messages understood by blocks. For example, a simple loop that executes ten times can be written

```
a ← 0.
[a < 10] whileTrue: [a ← a + 1]
```

using the method 'whileTrue:' which is implemented recursively in BlockContext as:

```
whileTrue: aBlock
    ↑self value
        ifTrue:
            [aBlock value.
            self whileTrue: aBlock].
```

## 2.2 Compilation

The task of a compiler is to translate a valid *sentence* of a *source grammar* into a (usually) executable form, whilest rejecting incorrect source sentences. We can consider a compiler as a program that recognizes correct sentences of its source grammar, producing executable code as a side effect of recognition.

### 2.2.1 Compiler Anatomy

Compilers are usually constructed from several modules, each of which performs one particular function. Modules are implemented as separate passes, with passes communicating through intermediate representations of the program, or as subroutines which other modules can call when a particular action is required.

The *lexical analyzer* (or *scanner*) is responsible for converting the plain text version of the program into discrete *lexemes*; these lexemes correspond to the *terminal symbols* in the source grammar.

The *syntax analyzer* (or *parser*) is responsible for assembling lexemes into sentential forms, and ultimately recognizing or rejecting (with suitable error reporting) the source program. As a side effect, the parser either builds a structure such as a *parse tree* which describes the syntactic form of the source, or makes calls on subroutines to output code directly. Producing a parse tree has the advantage of enabling high level (global) optimizations to be applied, as well as allowing the walk of the tree to be ordered by the Sethi-Ullman complexities [ASU86, page 561] of the subtrees at a node, which helps the code generator make the most efficient use of machine resources in the generated code.

The *code generator* (or *translator*) converts the (possibly optimized) parse tree representation of the program into either an executable or *intermediate* code. Executable output will be actual instructions for the target machine and are often produced in the human-readable form of assembly language statements; these are processed by an assembler to produce the final executable. Intermediate forms are sequences of instructions for hypothetical machines supporting operations similar to those of the target architecture. They are usually in the form of two-address or three-address code, which is processed further to produce the final executable. Producing intermediate code has the advantage of allowing low-level (local and peep-hole) optimizations to be applied more easily (these are discussed further in section 5.5.3).

### 2.2.2 Target Languages

A compiler writer has two options when designing the back end of a compiler: either to generate *native code* or *virtual code*.

If the source language can be mapped easily onto the architecture and resources of the hardware on which it is to run then real machine code can be generated. This machine code is called native code or *n-code*.

There is an alternative for languages that cannot be mapped easily onto real hardware, or languages in which code size must be minimized at all costs. In these cases, a hypothetical *virtual machine* (VM) is designed which implements an 'ideal' instruction set catering exactly for the requirements of the source language. The compiler generates code for this virtual machine, and can be much simpler since the semantics of the source and target languages are so closely matched. The VM is implemented on a real machine in software, and the code generated for such a virtual machine is called virtual code or *v-code*.

There are points both for and against each approach. In the case of n-code, a large semantic gap between the source and target languages can cause many machine instructions to be generated to perform a single source-level operation, which in turn can result in a relatively large amount of object code for a given program. With v-code, the compiled code size is usually minimal — the object code being a compact and concise encoding of the semantics of the original source program. However, v-code implementations usually suffer from overheads associated with the v-instruction 'fetch-decode-dispatch' loop which is written in software. A n-code compiler for the same language that simply inlines the runtime actions of the v-instructions that a v-code compiler would produce should be faster since this fetch-decode-dispatch is no longer required. In practice, the performance gains made by removing this overhead may be thwarted by the larger size of the object code which will cause increased paging in a virtual memory environment.

## 2.3 Summary

The Smalltalk language has many novel features that create difficulties for both the runtime system implementor and the compiler writer. Both the representation of data as objects, and the performance of computation by message passing, have a profound effect upon the design of the implementation. The situation is always a tradeoff, where complexity in the runtime system can be traded for complexity in the compiler.

Many Smalltalk systems choose to push most of the complexity into the runtime system, and have the compiler target to a virtual machine whose instruction set is designed to follow the semantics of the source language as closely as possible. Efficient implementation of the virtual instruction set can vastly increase the complexity of the runtime system, as we shall see in the next chapter.

# Chapter 3

# Related Work

> The dwarf sees further than the giant when he has the giant's shoulder to mount on.
>
> Samuel Taylor Coleridge, *The Friend*.

> To stand still on the summit of reflection is difficult, and in the natural course of things, who cannot go forward steps back.
>
> Gaius Velleius Paterculus (20 B.C. to 30 A.D.).

This chapter introduces two areas of relevant related work. First is the proprietary Smalltalk-80 system developed by the Software Concepts Group of the Xerox Palo Alto Research Center, Smalltalk-80 version 2.3.[1] After this, some related work on code generation will be described.

## 3.1 A Proprietary Smalltalk: ParcPlace PS2.3

Several factors influenced the choice of implementation model in the Xerox versions of Smalltalk-80. Early versions of the language were targeted to microcoded machines, so the obvious choice was to generate code for an ideal virtual machine which would be emulated either by microcode or a standard instruction set augmented with Smalltalk-80-specific microcoded instructions [Kra83, pages 114-117]. These machines had small (64K) address spaces, so providing a virtual instruction set was desirable to keep the size of the compiled methods to a minimum. Additionally, since runtime state

[1] The original work described in this thesis is based on version 2.3 of the ParcPlace Smalltalk-80 system which was the most up to date version available at the time the work was started. This version has since been superseded by version 2.5 and Release 4 which introduced many changes to the system and language, the impact of which were too great to be accommodated in the time available for this work.

could be represented as programmer-accessible Smalltalk-80 objects, debuggers and other support software could be written entirely in the Smalltalk-80 language.

The implementation is therefore split into two separate parts: the virtual machine which implements the machine-specific functionality, and the *virtual image* which contains the Smalltalk-80 objects (data and compiled methods). This organization also provides a high degree of portability: a virtual image containing compiled methods and user data can be interpreted by a variety of virtual machines implemented on different platforms. Smalltalk-80 virtual machine implementations that use identical image formats are currently available on a number of hardware platforms provided by different manufacturers.[2]

### 3.1.1 The Virtual Machine

The virtual machine provides several services: an evaluator for compiled methods, support for *primitive operations*, and the memory management system.

The instruction set is stack-based (for ease of code generation) and virtual instructions (called *bytecodes*) are executed by the evaluator from compiled methods held in the virtual image. The majority of bytecodes are associated with message sends and returns, and transferring variable values and object fields to and from the stack [GR83, page 596].

A set of primitive operations are provided by the VM to perform tasks that are either beyond the capability of the Smalltalk-80 system (such as machine-specific input/output), or that have a significant impact on execution speed (such as bitblt operations) [GR83, pages 612-615]. Many primitives supported for performance reasons are optional and are allowed to 'fail', in which case recovery is attempted by Smalltalk-80 code in the virtual image.

There are no bytecodes that directly invoke a primitive operation, instead each compiled method has a header which describes it. Primitives are invoked by normal message sends that cause the execution of a compiled method whose header identifies it as a primitive operation. If such a method has a body, then the body is invoked if the primitive fails for some reason.

Memory management is transparent to the virtual image. The VM maintains reference counts on all objects, and deallocates objects automatically when it can. A separate garbage collector is provided which can be invoked from Smalltalk-80 and is used to reclaim circular garbage which the reference counting system cannot cope with. Deferred reference counting [DB76] is used during method execution and eliminates about 85% [DS83] of normal reference counting operations. The details of the object allocation and reclamation strategies are not relevant here, but are described in [GR83, chapter 30] and [DB76].

An important principle underlying much of the VM is *dynamic change of representation*. The same information may be represented in two or more (structurally)

[2]Numerous UNIX platforms (Sun, HP, and so on), Apple Macintosh, and 386/486 PC to name just three.

different ways, being converted transparently to the most efficient representation on demand. The two major types of information treated in this way are compiled methods and their contexts.

#### 3.1.1.1 Dynamic Method Translation

The bytecodes of a compiled method are a compact representation of the source program's semantics. This representation is compact but inefficient for a straightforward evaluator to interpret. The ParcPlace Smalltalk-80 VM *dynamically translates* bytecode methods into native machine code, method by method, on demand. Translated methods are cached rather than paged; in other words, if the translated method cache fills up, methods are rejected from it and regenerated later if necessary.

Naïve method translation is similar to macro expansion: each bytecode expands to a sequence of n-code instructions that perform the required operations. However, during translation there is the opportunity to perform various peephole optimizations and even to map stack references onto register references.

Translating bytecode methods into n-code methods removes some interpreter overheads. Specifically, the v-instruction fetch-decode-dispatch overhead is removed completely. Peephole optimization also removes some reference counting operations: the overall reference count impact of a sequence of pushes and pops may be zero, and reference counts can be ignored in such cases.

It is possible to reduce the level of polymorphism in translated methods, which can be specialized on the class of the receiver. If a particular bytecode method is inherited, different translated versions will exist (if demanded) for its execution in the defining class and in each of the subclasses inheriting the method.

Translated n-code occupies about 5 times the space of v-code [DS83] and could place severe stress on a virtual memory system; the bytecode representation of methods has been retained for this reason, and for their usefulness in some debugging tasks.[3]

Some bookkeeping information must be kept at runtime to support access to a method's context and for determining the point of control during debugging. The largest structure needed is a map from n-PC addresses to offsets within bytecode methods, used to determine the point of send for messages when debugging and also for purging the inline cache (section 3.1.1.3) when a method is rejected from the n-code cache.

#### 3.1.1.2 Multiple Representation of Contexts

Contexts are Smalltalk-80's version of activation records. A context contains information about the caller (for returning), space for temporary variables and arguments, and some stack space for the method to use during execution. Because a block executes in the context of its defining method, and blocks can outlive their defining method, method contexts are conceptually allocated as a linked list in the heap. This

[3]The primary motivation for using dynamic translation is the possibility of using an inline cache in the final n-code (see section 3.1.1.3).

is clearly inefficient: a more conventional LIFO stack of activation records would be better. About 85% of contexts are 'well behaved' [DS83], and act as normal LIFO activations.

The ParcPlace VMs use three representations for contexts. A *volatile* context is allocated on the stack, and deallocated from there by the normal n-code return sequence when its method exits. Such contexts are optimized for execution, but contain some spare slots so they can (if necessary) imitate an object in a restricted fashion.

A *hybrid* context is a volatile context in which the spare slots have been filled to make it look partly like a real data object. A volatile context is converted to a hybrid context whenever a pointer is generated to it, usually caused by pushing thisContext onto the stack or exporting a reference to the context in a block context.

If a message is sent to a hybrid context, or if a hybrid context that may be referred to from a block context tries to return, it is converted into a *stable* context and moved from the stack to the heap. Stable contexts are fully-fledged data objects compliant with the virtual machine specification. Stable contexts are always converted back to hybrid contexts for execution.

Method contexts are born on the stack. Most of these remain volatile and will be deallocated in a LIFO fashion. On the other hand, block contexts are born stable, since they represent code that is arbitrarily deferred.

#### 3.1.1.3 Caches

Two types of cache are popular in Smalltalk-80 implementations. A global *method cache* retains <class, selector> pairs from message sends along with the address of the associated method. If another send of the same selector to the same class of object is encountered, the destination method is read from the cache without invoking a full lookup.

*Inline caches* exploit the dynamic locality of type usage in Smalltalk-80: at any given point of send, the class of the receiver tends to be the same as it was the previous time the send was made. Initially, a message send consists of a call to the full lookup routine. This routine determines the destination method and then patches the point of send with a direct call to this method. The next time the point of send is reached, the previously called method is invoked directly without a full lookup. Each method begins by checking that the class of the receiver is the same as the class of the previous receiver (stored at the point of send[4]), invoking a full lookup and repatching the send if not.

ParcPlace report an 85-90% hit rate with their method cache which improves performance by 20-30%, and a 95% hit rate with their inline cache which improves performance by 10%, although it increases the size of the translated methods by a factor of 5 [DS83].

[4]It is possible to store the previous receiver's class at the start of the method instead, but since there are more sends than methods a better hit rate can be achieved if classes are cached at the point of send.

### 3.1.2 The Compiler

The compiler performs several optimizations to increase performance, some of which compromise the object-oriented nature of the language in a controlled way.

A significant proportion of message sends are related to conditional execution and looping. The compiler gives special treatment to sends of ifTrue:, whileTrue, and the other similar control selectors; for such messages, the send is removed completely and the message is macro-expanded inline. An example is given in [GR83, page 550].

| | | | |
|---|---|---|---|
| † + | † - | † < | † > |
| † <= | † >= | † = | † = |
| † * | † / | † \ | @ |
| † bitShift: | † \\ | † bitAnd: | † bitOr: |
| at: | at:put: | size | next |
| nextPut: | atEnd | † == | † class |
| blockCopy: | † value | † value: | do: |
| new | new: | x | y |

**Figure 3.1:** The 32 special selectors. Those marked † are short-circuited to the appropriate primitive in PS2.3. The others invoke a normal lookup, but save space in the literal frame by having a "known" selector.

The compiler can generate 'special' sends for 32 frequently used selectors. Figure 3.1 shows these special sends, which are encoded in a single bytecode and include the 15 arithmetic and comparison selectors plus 17 other frequently used selectors. In the case of the arithmetic selectors the special send bytecode offers an improvement in speed by invoking the primitive associated with the selector directly, assuming the receiver to be of the most common class; if the primitive fails then a full send is performed. Of the other 16 selectors 'value', 'value:', 'class' and '==' are also treated in this manner, to improve performance in time.[5] The remaining 13 selectors are encoded as special sends with "known" selectors[6] that need not be entered into the literal frame for the method. Similarly, there are special bytecodes for pushing the seven most frequently used constants onto the stack.

When a method starts up, there is a considerable amount of overhead caused by the creation and initialization of the context in which it is to run. To reduce this overhead, the compiler does not generate bodies for some methods and the VM ensures that

[5] Both the short-circuiting of special arithmetic selectors, and the macro expansion of control constructs, break the semantics of the language. This is only barely justifiable because of the performance improvements that are possible, and by the observation that any changes to the behavior of these messages would damage the system beyond repair.

[6] The global array SpecialSelectors provides a means for Smalltalk to communicate to the runtime system which selectors are associated with the special bytecodes. When executing a special send bytecode, if the message is not short-circuited to a primitive then its selector is extracted from this array rather than the literal array of the method itself.

these methods are never started up in the usual fashion. Methods without arguments that simply return the receiver or an instance variable of the receiver are identified by their method header as a 'primitive return' method and when executed the evaluator pushes the receiver or instance variable onto the stack as appropriate, without incurring the overhead of a full entry to the compiled method.

## 3.2 Orthogonal Code Generation

Chapter 6 introduces a technique called *delayed code generation* (DCG). This technique is similar in some ways to one developed by Cordy, Holt and others [Hol87, CH90] in which runtime data is represented at compile time by 'data descriptors'. Although the two techniques are rather different, they exhibit some similarities which warrant a brief introduction to the work that has already been done on data descriptors.

Data descriptors were developed from the work of T. R. Wilcox [Wil71] on 'value descriptors' in the PL/C compiler for PL/I [CW73]. The motivation behind their development as a tool in code generation was the desire for small, highly retargetable code generators for languages such as Euclid.

### 3.2.1 Data Representation

Most modern (CISC) computer architectures provide a wide range of operand addressing modes. The choice of modes available to the programmer hasn't changed much since the days of the PDP-11, and the majority of the extra modes available only extend long established features. For example the 68020 offers many more addressing modes than the earlier members of the same family, but these additional modes simply provide an extra level of indirection. Indeed, it is in the treatment of indirection that data- and value-descriptors differ.

Data descriptors were designed to accommodate either the complete set of addressing modes available over a wide class of architectures, or at least the majority of the important modes. The essential features of the machines in the architecture class described by Holt are:

- a base address (normally the contents of a machine register) $b$,
- a displacement from the base (an integer constant) $d$,
- an index (a machine register) $i$, and finally
- an optional indirection level $k >= 0$.

A data descriptor can thus be written $@^k b.d.i$ with any null (zero) fields omitted for clarity. The 'value' of a data descriptor is the sum of $b$, $d$ and $i$, all taken $k$ levels indirect. Many types of high-level data can be represented using this notation, as is shown in figure 3.2.

| | value | descriptor | canonical form |
|---|---|---|---|
| literals | zero | $@^0 null.0.0$ | 0 |
| | constant N | $@^0 null.N.null$ | N |
| | address A | $@^0 null.A.null$ | A |
| registers | $R_i$ | $@^0 R_i.0.null$ | $R_i$ |
| memory | contents of A | $@^1 null.A.null$ | @A |
| | address of $A[R_i]$ | $@^0 null.A.R_i$ | $A.R_i$ |
| | contents of $A[R_i]$ | $@^1 null.A.R_i$ | $@A.R_i$ |

**Figure 3.2:** High-level values represented by data descriptors

### 3.2.2 Operations on Data Descriptors

A few commonly provided high-level language operations map directly onto operations on data descriptors. When processing these high-level operations a compiler need not generate any code, but can simply modify the data descriptor representing the data being operated upon.

For example, languages such as C and Pascal provide a pointer dereferencing operator (‘`*`’ in C, ‘↑’ in Pascal). When applied to a value, this operator causes an extra indirection which can be seen to correspond with the @ operation on a data descriptor. For a data descriptor D representing a value ‘`p`’, the result of ‘`p`↑’ is @D. Similarly, the ‘address of’ (‘`&`’) operator in C causes the operation $@^{-1}$ to be applied to a data descriptor.[7]

### 3.2.3 Data Descriptors

The work on data descriptors was extended by Cordy and Holt [CH90] into a complete machine-independent code generator. The major goals of the work were clarity and portability in the code generator.

In traditional compilers, ‘abstract operands’ (values of the source language such as structure fields and array elements) are often represented by sequences of machine operations using simple addressing modes. This is problematic for two reasons. First, the domain of addresses may be different from the domain of integers for normal arithmetic. Second, when generating the final machine code, sequences of arithmetic instructions involved in address calculation must be removed when the same address arithmetic can be performed by using the relevant addressing mode of the target architecture.

These problems were tackled by separating code generation into two independent

[7]In C only an ‘lvalue’ can have its address taken, and this check comes for free when using data descriptors. Since an indirection level of $k = -1$ has no reasonable interpretation, only data descriptors with $k >= 1$ can have their addresses taken.

tasks (which introduces the required clarity into the code generator). The source program is translated into a pseudo-code containing common generic operations (such as "add" and "move") with operands of arbitrary complexity represented by data descriptors. A two-stage process then maps this pseudo-code to code suitable for the target architecture. First *operator mapping* translates the abstract operations of the pseudo-code onto operations supported by the target architecture, still assuming arbitrarily complex abstract operands in any operand position. Next *operand mapping* translates abstract operands into the addressing structures supported by the target machine, correcting for the assumption made during operator mapping. Any operands that attempt to use an addressing mode not supported for a particular instruction are expanded into a minimal sequence of instructions, using legal addressing modes, having the desired overall effect.

The choices made during the mapping of operators and operands are based on sets of 'decision trees' which can be specified and generated for particular architectures in a machine-independent manner. This introduces the required portability into the technique.

Abstract operations can be generated using an elegant and general mechanism based on the manipulation of the abstract operands (data descriptors) at compile time. When a particular operation is required on one or more data descriptors (during the generation of code for an addition operation, say), the code generator attempts to perform the operation directly on the data descriptors concerned. For example, if the addition was between two data descriptors representing literals then the addition can be performed at compile time without the need to generate any code; the "result" of the addition simply being a data descriptor representing a literal whose value is the sum of the two operands.

| abstract operation | input 1 + input 2 | result |
|---|---|---|
| 3+4 | $@^0 null.3.null$ + $@^0 null.4.null$ | $@^0 null.7.null$ |
| $R_b[5]$ | $@^0 R_b.0.null$ + $@^0 null.5.null$ | $@^0 R_b.5.null$ |

**Figure 3.3:** Abstract addition of data descriptors

---

This technique is abstracted by Holt into a set of "super" operations corresponding to the abstract operations supported in the code generator. In the addition example, when translating an addition in the source program, the code generator calls "superAdd" with the two operands (represented by data descriptors) as the arguments. The result is a data descriptor representing the location of the result. For non-trivial arguments "superAdd" may generate some abstract operations leaving the result in a memory location or register; in such cases "superAdd" returns a data descriptor representing the whereabouts of the result. Figure 3.3 illustrates the technique.

This technique can be applied throughout the code generator not only for other

types of arithmetic operation (including array element and record field address calculations), but also for 'imperative' operations such as assignment.

### 3.2.4 Strengths and Weaknesses of Data Descriptor Techniques

Code generators based on data descriptors and "orthogonal" techniques offer many advantages over more traditional techniques, but suffer from some deficiencies due to the (necessarily) limited number of assumptions made about the capabilities of the target architecture. Data descriptor techniques are inherently portable across the class of architectures for which they were developed, and allow rapid retargeting of the code generator to other machines in the same architecture class. However, retargeting to machines with significantly different addressing models (stack-based processors for example) is much more difficult.

Generating operations as a side effect of performing "super" operations on data descriptors provides the opportunity for certain local optimizations. The "superAdd" example above illustrates that constant folding can be provided almost trivially.

Some serious deficiencies also arise from using data descriptors in the construction of code generators. Data descriptors were designed to accommodate the most common features found in the addressing modes of a wide variety of CISC processors. It is no surprise then that many machines provide architectural features that cannot be represented within a data descriptor. One important such feature is auto-increment/decrement which is provided by machines such as the PDP/11 and M68000 family.

Data descriptors are little more than a means of representing the location of a particular value. This is fine for concrete values (literals, variables, and so on) but is not sufficient for the representation of 'implicit' values. An 'implicit' value is one that does not actually exist in a concrete form, and may be something like the result of a relational expression whose 'value' is some interpretation of the bits within the processor's condition codes register.

Code generators based on data descriptors initially make the assumption that an arbitrarily complex data descriptor is legal as an operand in any position for any operation. This assumption is corrected during operand mapping where the code generator "forces addressability" by rewriting single operations involving illegal addressing modes as short sequences of operations involving only legal addressing modes. This rather narrow view, inspecting one instruction at a time, can lead to less than optimal code where knowledge of previous or subsequent instructions could have influenced the generation of better code.

Compilers using data-descriptor based code generators are invariably multi-pass. Apart from anything else, the operator- and operand-mapping phases of the code generator are implemented as distinct passes over the generated code. This limits the applicability of the orthogonal approach in recursive-descent compilers, which have many appealing attributes particularly if the language to be compiled is relatively small and compilation times must be kept to an absolute minimum.

## 3.3 Optimizing Compilers for Smalltalk-80

In this thesis Smalltalk-80 was chosen as the ideal language for investigation primarily for its simplicity, allowing the code generation technique described in chapter 6 to be developed in the environment of a fully functional compiler for a complete language. Nevertheless, it is worthwhile introducing some of the work that has been done in order to illustrate the kinds of problems facing Smalltalk-80 compilers, and the sorts of techniques that have to be adopted in order to overcome them.

A number of attempts have been made to produce optimizing compilers for Smalltalk that target native machine code, Hurricane [Atk86] and the SOAR compiler [CP83] [LB83] being some of the most successful. One system in particular, Typed Smalltalk (TS) [JGZ88], adopts some novel techniques in order to provide opportunities for optimization in the compiler.

### 3.3.1 Typed Smalltalk

Probably the most important feature of Smalltalk, as far as the implications for its compilation are concerned, is that it lacks any form of compile-time type checking. In the same way that many languages are described as "strongly typed", Smalltalk-80 could equally well be described as "strongly polymorphic". The nearest it comes to type checking is the failure of a message send at runtime, when a message is not understood by the object to which it is sent.

TS is similar to some of the other compilers mentioned above in that it approaches this problem by modifying the language to support type declarations. Armed with type declarations, the compiler can attempt to infer the set of classes to which an object belongs at compile time and then produce code optimized for receivers belonging to those classes. Knowledge of the class of the receiver of a message allows the compiler to uniquely identify which method will be invoked at runtime when the message is sent, allowing the message send to be reduced to a simple procedure call — a mechanism known as *static binding*.

If the set of possible classes of a receiver is sufficiently small, the parse tree representation of the message send is rewritten in the form of a case analysis on the class of the receiver. In conjunction with other optimizations such as the inlining of blocks that are explicitly sent the 'value' message (beta reduction), and direct substitution of statically-bound messages in the parse tree itself, the TS compiler can produce code that is closer in efficiency to languages such as C than it is to untyped implementations of Smalltalk-80.

The translation of the parse tree into executable code is accomplished by first generating a machine-independent intermediate representation expressed in a register transfer language (RTL). Peephole optimizations that would normally be carried out on the executable form are instead performed on the RTL, which has the advantage that the optimizer is not tied to any particular target architecture. Once optimized, the RTL is then translated into native code which requires no further optimization.

There are several problems with this approach. The introduction of explicit typing

into the language places an extra burden on the programmer, and forces program designers to predict the uses to which their code will be put. The designers of TS freely admit that not every legal Smalltalk-80 method will be successfully type-checked by their system. The additional overheads of the many passes of the TS compiler (parsing, type inferencing and checking, parse-tree optimization, conversion to RTL, RTL optimization, and final generation of native code) cause the compiler to run between 10 and 15 times more slowly than the compiler in systems such as PS2.3; this is disastrous in an EPE.[8]

More subtle problems also arise. Since the compiler makes heavy use of inlining based on both the functionality *and* the type signature of methods, if either of these is modified by the programmer a potentially large number of methods may need to be recompiled due to the invalidated assumptions that were made during their compilation. TS tackles this problem by maintaining dependency information between methods, allowing the set of affected methods to be determined easily should the definition of an inlined method change. To avoid long delays while methods are recompiled, fully polymorphic versions of any methods affected in this way (that make no assumptions about the types of their arguments) are used while the compiler rebuilds optimized versions of them as a background task.

## 3.4 Summary

Bytecode-based implementations of Smalltalk-80 present an execution model which makes compilation trivial but implementation of the runtime system very difficult. The latter must incorporate several sophisticated techniques to achieve an acceptable level of performance, including the dynamic (runtime) conversion of data between different representations according to the usage of that data. The dynamic binding of message sends to destination methods also adds significant overheads which are overcome using caching techniques where the results of previous method lookups are retained for later use.

Some previous work has investigated the possibilities of using compile-time representations of runtime quantities to aid the process of code generation. The major goal of the work was to produce code generators of modest complexity that were highly retargetable between machines with similar instruction sets. Some of the segregation between code generation phases implied by these goals, and the rich variety of addressing modes (some of which have side-effects on machine state), led to problems in generating high quality code. These code generators tended to be multi-pass, working with several different intermediate representations. This limits their usefulness in environments where small, fast compilers are essential.

[8]It is predicted that in a mature system using the TS compiler, the additional complexity of the compiler should be offset by the increased efficiency of the generated code. When the compiler can compile itself, it should be as efficient as compilers in commercially available systems such as PS2.3.

# Chapter 4

# Benchmarks

> It seems to be gratuitously courting disaster to expose our theories to conditions in which any slight weakness is likely to become magnified without limit. But that is just the principle of testing.
>
> Sir Arthur Eddington, *The Expanding Universe*.

A benchmark provides a meter for the absolute performance of a system, or the relative performance of several systems, for a selected task. The main reason for their use is in making comparisons between different systems, or for evaluating the effect of optimizations within a system. However, care must be taken to avoid benchmarks that show vast improvements in performance simply because they concentrate on those aspects of a system that are implemented efficiently. Benchmarks that concentrate on a single part of a system are only really valuable for determining the effectiveness of different implementation strategies affecting that particular part. In this thesis, benchmarks that concentrate on small aspects of a system are called *micro-benchmarks*. In Smalltalk-80, operations such as integer addition can benefit from alternative implementation strategies and so are legitimate subjects for micro-benchmarks.

More realistic measures of a system's performance can be gathered from *macro-benchmarks*, which aim to exercise large parts of a system, ideally those parts which users regularly exercise. In Smalltalk-80, subsystems such as the compiler or the user interface are good candidates for macro benchmarking. Macro benchmarks are more likely to reflect the performance as perceived by the user than are micro benchmarks.

Benchmarks provide a measure of the time performance of a system, but this is not always the only important factor to consider. The size of generated code can also be significant, especially in Smalltalk-80 where a large proportion of the image consists of compiled code.[1]

[1] For example, in the (PS2.3) image containing the Native Code Smalltalk-80 compiler there are 35980 objects, of which 16.5% are compiled methods. These account for 20% of the 1.43Mb total space allocated for object bodies. Methods are slightly larger than the average object too, being just over 48 bytes long compared to the average of just under 40 bytes per object.

This chapter presents briefly the benchmarks chosen to meter the performance of the various optimizations performed by the compilers described in the following chapters.

## 4.1 Benchmarking Smalltalk-80

The micro-benchmarks focus on several areas that impact the performance of most Smalltalk-80 applications: variable (instance, argument and global) access, arithmetic and comparison operations, conditional and looping operations, block context creation, method and block activation, and storage allocation. For those micro-benchmarks that are associated with a particular optimization in the compiler, only a single benchmark is used. For example, the performance of arithmetic operations performed on SmallInteger arguments can be improved by generating the code to perform them directly without a full message send. Even though all simple arithmetic operations benefit from this treatment, the performance gains will be comparable between them all. Therefore a single arbitrarily chosen operation (in the case of inlined arithmetic selectors, the operation is SmallInteger addition) is used as a representative for the entire group.

Larger areas of the system are also exercised. For example, much user activity relies on Smalltalk-80's text formatting, editing and display operations, so these activities are included in the set of macro benchmarks used. The maintenance of the Smalltalk programming environment relies heavily on the manipulation of complex data structures (lists, dictionaries, and so on) so some macro benchmarks that perform this kind of activity are also included.

### 4.1.1 Benchmark Framework

A total of 38 benchmarks were used to gather performance data. Some of these were drawn directly from the benchmarks developed at the Xerox Palo Alto Research Center [Kra83, chapter 9] for use in metering the performance of Smalltalk implementations. Others were developed either to fill in gaps left by the Xerox benchmarks, or to provide coverage of areas not adequately explored by the former (such as those that perform large amounts of numerical processing, implement highly recursive functions, and exercise the 'SystemTranscript').

The benchmark suite used was identical in both Native Code Smalltalk-80 and PS2.3. The timing information was gathered from within primitives rather than from within Smalltalk itself, partly to increase the accuracy of the timings[2] but also to gather other information regarding resource utilization.

In addition to the timing information itself, the Native Code Smalltalk-80 runtime system was instrumented in order to gather data about the behavior of the benchmarks

[2]Smalltalk can only provide a measure of elapsed "real" time, whereas a much more accurate measure of the amount of time a UNIX process spends doing a task can be determined by adding the "system" and "user" times for that process.

themselves. This was used both to validate the choice of benchmarks (after all, a benchmark meant to test addition would be worthless if it performed more comparisons than additions) and also to measure the impact of various optimizations on the dynamic behavior of the system — the most important consideration being the number of "expensive" runtime support operations, such as dynamic binding and object allocation, performed.

### 4.1.2 Micro-Benchmarks

The micro-benchmarks chosen to measure the relative performances of Native Code Smalltalk-80 and PS2.3 are mostly derived from the standard Xerox benchmarks, and are useful mainly to determine the effectiveness of optimizations in the compiler and runtime system. They can be divided into several categories according to the particular aspect of the system's implementation which they reflect:

- **Arithmetic and Comparison**
  The common operations performed on numeric quantities, and the other kinds of Magnitude in the case of comparisons. These are short-circuited to the appropriate primitive in PS2.3, and performed inline in Native Code Smalltalk-80. The associated selectors that are exercised are: '+', '<' (on SmallIntegers), '<' (on Strings), and '=='.

- **Conditional**
  Controlling conditional execution. These are macro-expanded inline in both PS2.3 and Native Code Smalltalk-80. The associated selectors are: 'ifTrue:', and 'ifTrue:ifFalse:'.

- **Looping**
  Controlling iterative execution. These are macro-expanded inline in both PS2.3 and Native Code Smalltalk-80. The associated selectors are: 'whileTrue', and 'whileTrue:'.

- **Variable and Literal Access**
  Both loading and storing temporary, receiver (instance), and global variables. Loading a literal quantity.

- **Activation**
  Rapid call to and return from methods and blocks.

- **Point Operations**
  Creation and instance variable access. Creation is short-circuited to a primitive in PS2.3, and performed inline in Native Code Smalltalk-80. Access to the fields is also inlined in Native Code Smalltalk-80. The associated selectors are: '@', 'x', and 'y'.

- **Memory Allocation**
  Rapid creation of an aggregate object, without initialization. The associated selector is 'new'.

- **"No Lookup" Operations**
  The non-arithmetic operations that are short-circuited directly to primitives. The messages 'class' and 'value' are short-circuited to a primitive in PS2.3. In Native Code Smalltalk-80, 'class' is performed inline and 'value' is actually sent (by the normal lookup mechanism). The associated selectors are: 'at:', 'at:put:', 'class', 'perform:with:', 'size' and 'value'.

- **Graphical Operations**
  Exercising the 16 combination rules of BitBlts. This benchmark is useful mainly to determine the match between the bitblt performance of Native Code Smalltalk-80 and PS2.3. The associated selector is 'copyBits'.

### 4.1.3 Macro-Benchmarks

These benchmarks exercise fairly large parts of the system, in both the user interface and invisible 'book-keeping' activities. Some are drawn from the Xerox benchmark suite, others were developed to exercise parts of the system otherwise ignored by the Xerox benchmarks. Again, they fall into several categories:

- **Data structure manipulation**
  Creating, initializing and walking a large binary tree. There is some crossover here with the "interface" benchmarks, which perform a non-trivial amount of data structure access.

- **Numerical**
  Evaluating the doubly recursive 'nfibs' and triply recursive 'Takeuchi' functions.

- **Interface**
  Writing messages on the Transcript, displaying plain text, parts of the class hierarchy, and class definitions in a text window. Note that these also perform a fair amount of data structure manipulation. Since much of the user interface uses text manipulation facilities, these benchmarks also include both formatting (inserting line breaks in) a long string and repeated selection, replacement, and redisplay of a long string in a text window.

Conspicuous for their absence are benchmarks that exercise the input side of the interface (keyboard and mouse) and external I/O (file access). These were omitted because they have little value in the determination of the quality of the code produced by the compiler. The output-oriented benchmarks were only included because it was relatively easy to make the comparisons between Native Code Smalltalk-80 and PS2.3 largely independent of the performance of the graphics primitives; this will be discussed further in section 4.3.

## 4.2 Benchmark Characteristics

Although each of the micro benchmarks targets one specific operation, it is impossible to avoid some interference in the timings caused by other operations necessary just to run the benchmark. The table in figure 4.1 shows which messages impact the performance of each of the benchmarks.

● targetted selectors
○ noise

| | + | - | < | <= | = | == | > | @ | at: | at:put: | class | ifFalse: | ifTrue: | ifTrue:ifFalse: | new | recur: | size | value | whileTrue: | x | y |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arithmetic Operations | ● | ● | ○ | | | ○ | | | | | | | | | | | | ○ | | | |
| Conditional Execution | | | | | | | | | | | | ● | ● | ● | | | | ○ | | | |
| Looping Operations | | ○ | | | | | ○ | | | | | ○ | ○ | ○ | | | | ○ | ● | | |
| Variable Access | | | | | | ○ | | | | | | | | | | | | ○ | | | |
| Method/Block Activation | | ○ | | | ○ | | | | | | | | ○ | | | ● | | ● | | | |
| Point Operations | | | | | | | | ● | | | | | | | | | | ○ | | ● | ● |
| Memory Allocation | | | | | | | | | | | | | | | ● | | | ○ | | | |
| No-Lookup Operations | ○ | | | ○ | | | | | ● | ● | ● | | ○ | | | | ● | ● | ○ | | |

**Figure 4.1:** Messages impacting the performance of each of the benchmarks. Those marked with a solid circle are specifically targeted in the benchmarks which they impact. Those marked with a hollow circle are 'noise' in that their performance will have a significant effect on the benchmark's performance even though those selectors are not specifically targeted by that benchmark.
Note that this table does not include the macro benchmarks which, due to their nature, spread themselves more thinly over a much larger set of selectors.

---

## 4.3 Fairness

Much important functionality in Smalltalk is not directly attributable to compiled methods. The virtual machine (or its equivalent) must provide support for the primitives invoked from compiled code, and also for 'invisible' operations such as dynamic binding, method cache mechanisms, and memory management. For reasons described below, the runtime support for Native Code Smalltalk-80 could not realistically match the performance of the PS2.3 virtual machine in several areas. In an attempt to make comparisons between the two systems fairer, reflecting the quality of the compiled code and its *direct* support (rather than the quality of the runtime features specifically provided as performance enhancers), some modification of the PS2.3 image and virtual machine was undertaken. Just what constitutes part of the implementation as opposed to a performance feature is an area of possible debate. For the purposes of gathering data for this thesis, the only parts of the runtime system deemed to be performance features are the 'optional' (and 'copyBits') primitives. The justification for giving special treatment to these parts of the runtime system are as follows…

PS2.3 specializes many operations for particular classes in order to improve performance. This is usually done by providing some particular behavior written in Smalltalk itself, and then overriding this behavior using 'optional' primitives in subclasses.

For example, many operations on the collection classes are written in Smalltalk in an abstract superclass. The message 'replaceFrom:to:with:startingAt:' is defined (in Smalltalk) in class OrderedCollection from where most subclasses inherit the relevant behavior. For performance reasons alone, the same behavior for byte-oriented classes (ByteArray and String) is defined as a primitive. These two classes override the 'replaceFrom...' message to invoke the primitive response instead. Optimizations such as this pervade the system, and it would be a non-trivial task to write all the optional primitives required by the set of benchmarks above. Apart from this consideration, the real point of the benchmarks is to measure the performance of compiled code in Native Code Smalltalk-80 against the performance of compiled code in PS2.3 and *not* the performance of the underlying runtime support. For these two reasons the optional primitives required for the benchmarks have been disabled in the PS2.3 image used to generate the statistics.

A special case of this argument concerns the 'copyBits' primitive in class BitBlt. The 'copyBits' primitive in Native Code Smalltalk-80 is based on a portable 16-bit bitblt [Wol84] written in C.[3] Contrast this with the implementation in PS2.3 which is hand-written in assembler and most likely uses a self-modifying inner loop to squeeze every last bit of performance out of the hardware. Implementing a bitblt operation from scratch that has the performance of the latter is certainly a nontrivial task, and far outside the scope of this thesis.

In order to make the comparison between the two systems fairer (by removing as far as possible any imbalance in the performance of the primitives), it is desirable to use bitblt operations of comparable performance in the two systems. Rather than expend large amounts of effort in developing a 'copyBits' that was competitive with the PS2.3 implementation, the indigenous 'copyBits' in the PS2.3 virtual machine was replaced with the same 16-bit implementation used in Native Code Smalltalk-80.

## 4.4 Summary

Benchmarks tend to come in two flavors. Micro benchmarks target small parts of a system and are useful in determining the effectiveness of changes to the implementation strategy for parts of a system. Macro benchmarks try to avoid concentrating too closely on a single aspect of a system and exercise complete, or significant portions of, typical nontrivial end-user activities. They are invaluable in determining the performance of a system as it is likely to be perceived by an end user.

Smalltalk-80 offers several prime candidates for micro and macro benchmarking.

[3]Saying that the implementation is '16-bit' means that the inner loop considers 16-bit words when block-transferring data. In general, the larger the number of bits considered atomically in the inner loop, the faster the transfer will be. A 32-bit implementation written in the same language should be just about twice as fast for large transfers of data.

The object-oriented nature of Smalltalk-80 puts great demands on several low-level operations, that must be implemented efficiently for the system to run with an acceptable level of performance. These operations are prime candidates for micro benchmarking. For more realistic assessment of the success of an implementation strategy, Smalltalk-80 provides an unusual opportunity to draw macro benchmarks directly from its own large subsystems, such as the user interface.

Some care has to be taken regarding optional primitives and the bitblt operation, and the benchmarking environments in the different implementations must be made as similar as possible for the results to be useful.

# Chapter 5

# 68020 Native Code Smalltalk-80

Nothing will ever be attempted, if all possible objections must first be overcome.

Samuel Johnson.

Native Code Smalltalk-80 is an implementation of Smalltalk-80 on MC68020-based workstations. The major difference between it and the ParcPlace implementation is that it does not use bytecode methods. Instead, methods are compiled directly into 68020 machine code.

Before developing a compiler for any language, it is necessary to design the environment in which the compiled code will run, and decide on the conventions to which the code will adhere. Consequently these two topics will be discussed before describing the compiler in any detail.

## 5.1 Runtime Environment and Conventions

Native Code Smalltalk-80 retains the distinction between image and runtime system. The image contains executable code, but in the form of 68020 instructions rather than bytecodes. For this reason it will be referred to simply as an 'image' rather than a '*virtual* image'. Similarly, the runtime system contains some of the support provided by a virtual machine (message lookup, a method cache, memory management, the primitives, and so on), but does not include an interpreter for a virtual instruction set. There are several benefits from following this approach. Firstly, incorporating the runtime support in the image itself would have increased its size, so several different images would all carry a large amount of common code; it is more desirable to have a runtime system with interchangeable images. Secondly, different runtime systems can be used to execute the same image. This was especially important in the benchmarking work where several runtime systems with different instrumentation were used. Lastly, incorporating the runtime system into the image itself means storing the primitives and other supporting routines as proper objects which would have made garbage collection

more complex, while keeping them separately in the runtime system in no way affects the performance.

Other approaches are possible, such as the use of dynamically linked libraries. Although these offer distinct benefits in some situations, there was nothing to be gained from using them to provide runtime support in a Smalltalk implementation.

### 5.1.1 Object Memory and Object Format

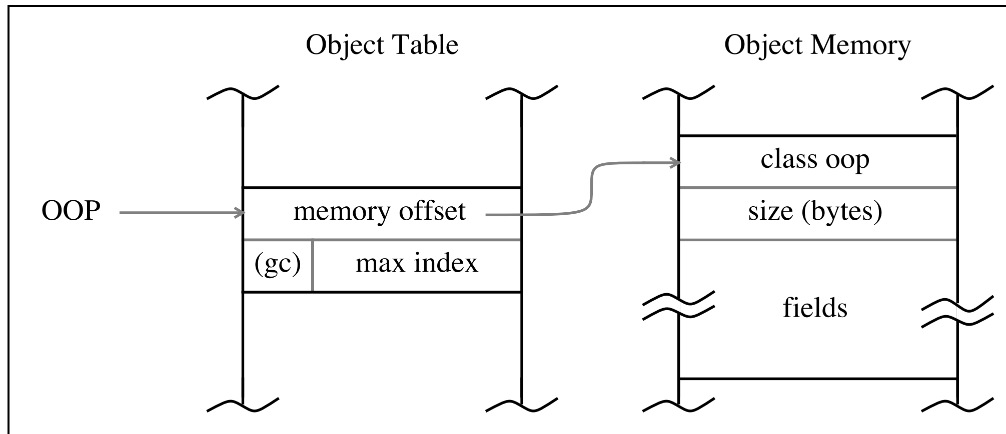


**Figure 5.1:** An arbitrary object pointer (OOP) is an offset into the object table. The object table contains an entry for each object giving the offset in the object memory at which the header begins, the number of indexable fields of the object, and some bits reserved for the garbage collector. Objects have a two word header giving the object's class and size, followed by the fields. (This object table format is equivalent to that used in PS2.3, but with minor changes to the locations of some of the fields.)

Memory is divided into two disjoint sections, the object memory (containing the actual object headers and bodies) and the object table (translating object pointers into object memory locations). For reasons of efficiency, the object table contains two entries per object giving the object memory location and the number of indexable fields of the object. For objects that contain 16- or 8-bit fields, this number will be twice or four times the number of object memory words used by the indexable fields. Keeping this information in the object table improves efficiency since the indexing primitives ('at:put:' for example) can perform a bounds check without having to follow the object's class pointer to obtain the format bits. The top byte is reserved for use during garbage collection.[1]

[1]This is only one of many object table formats possible, but is convenient for both bounds checking within primitives and for garbage collection. Superficially it may seem much more sensible to keep the class of the object in the object table to reduce the overheads of message sending by one memory reference, but considering the number of instructions and memory references executed as the result of a

This format has the disadvantage that every object access requires an extra indirection through the object table. However, the use of an object table simplifies both garbage collection and the process of saving and reloading an image. The 'become:' primitive is also much simpler when it merely swaps object table entries rather than having to scan the entire object memory, or allocate forwarding objects.[2] Apart from any other consideration, the use of an object table is in line with the implementation strategy of PS2.3.[3]

Objects allocated in the object memory have a two word header containing the object's class and its true size in bytes. Following these are the fixed and indexed fields of the object.

In accordance with every other Smalltalk-80 implementation, SmallIntegers are not allocated any space in the object table. Instead the value of a SmallInteger is encoded in its object pointer in such a way that it is distinguishable from a valid object pointer. Since true object pointers are zero-based byte offsets into the object table, they are all multiples of eight. The most sensible scheme for encoding SmallIntegers is therefore as twice their (signed) value with bit zero set, which causes SmallIntegers to have odd pointers and real objects to have even pointers.[4] The performance gains made from this approach more than compensate for the non-negligible increase in the overall complexity of the system.

Object pointers are indices into the object table rather than the actual address of the

---

message send this reorganization of the object table would have a negligible effect on performance (indeed, performance may even suffer due to extra overheads incurred during garbage collection). During method execution, the ratio of class fetches to size fetches is just over 5:1, so swapping the locations of the class and size fields would improve performance slightly. However, garbage collection accounts for a much larger number of accesses to both of these fields with a preponderance of fetches on the size field (the ratio is 1:4.7, class:size).

[2]One implementation strategy for 'become:' when direct pointers to objects are being used is to make a copy of the primitive's receiver and argument and insert references to these new objects in the original bodies, with the identities swapped. These "proxies" can be cleaned up during garbage collection (for example), but an additional overhead is incurred on each object access to check for the presence of a forwarding pointer.

[3]The design of the object table and object pointer formats can have a considerable impact on the performance of the system. In the absence of a processor data cache, indirections through an object table may be unacceptably inefficient. Also, the position of the tag bits identifying "immediates" such as SmallIntegers is important: in the absence of hardware support for tag bit checking and/or index register scaling, placing tag bits at the least significant end of the object pointers will usually be more efficient (because of sign-extension problems). A minor advantage of placing the tag bits at the most significant end is that checks for immediates can be reduced in many cases to a single 'bne' instruction.

[4]Since all OOPS are byte offsets into an object table, it would be possible to encode values for four classes directly in an OOP. (It may seem that seven immediate classes could be encoded in three tag bits, but one of these bits must be set for every immediate class if the inline checks are to remain efficient. Encoding more than one class in these bits also has the effect of making SmallInteger arithmetic more complex, and also reducing the range of SmallInteger values that can encoded in this way.) This encoding was only done for SmallIntegers since operations on these are so frequent. Increasing the number of directly encoded object types would increase the complexity of the compiler and runtime system for relatively little performance gain. However, ParcPlace have deemed it worthwhile to encode the value of Characters directly in their pointers in the more recent releases of Smalltalk-80 (2.4 and later).

object table entry. Likewise, the object table contains the offset from the start of the object memory at which the body can be found. By using offsets rather than absolute addresses in the object table, there is no need to convert from relative to absolute addresses and back again when saving or loading an image. The extra cost of adding the base addresses of the object table and memory is negligible, since these addresses are kept in two permanently assigned address registers and the calculation is performed at no extra cost (in either space or time) during normal effective-address calculation. The cost in terms of processor resources is also negligible, leaving sufficient registers free for other uses (see the next section).

### 5.1.2 Register Usage

| REGISTER | NAME | USE |
|---|---|---|
| `a7` | `sp` | stack pointer |
| `a6` | `frame` | frame pointer for contexts |
| `a5` | `obtab` | base of object table |
| `a4` | `obmem` | base of object memory |
| `a3` | `home` | home context for blocks |
| `a2` | | temp |
| `a1` | | temp, or cached base of 'self' |
| `a0` | | temp, or object base after allocation |

**Figure 5.2:** Address register usage and the mnemonic names of the five permanently assigned address registers.

---

The MC68020 provides eight general-purpose data registers (specialized for arithmetic and logical operations) known mnemonically as `d0` to `d7`, and seven general purpose address registers (optimized for use as base addresses and index registers) known as `a0` to `a6`.[5] Of these, four out of the seven available address registers have permanently assigned meanings in a Native Code Smalltalk environment, with two others being assigned particular meanings at certain times.

To facilitate access to objects, the bases of the object table and object memory are kept in address registers at all times. A frame pointer is used for access to temporary variables and for cleaning up the stack when a context exits, and an additional frame pointer is necessary which points to the home context during the execution of a block. These four permanently assigned address registers are normally referred to by their mnemonic names which are shown in figure 5.2.

The remaining three address registers are for general use, although some can contain standard values in certain circumstances. When the object allocator is called it returns the OOP of the allocated object in `d0`, and (for convenience) the object memory address of the new object's class field in `a0`. Finally, references to instance variables

[5] `a7` is dedicated for use as a stack pointer in all call and return operations, although architectures such as the PDP-11 have demonstrated the benefits of more flexibility in this respect.

require the method to calculate the base address of the receiver; this address is cached in a1 whenever possible.

The data registers are mainly free for use as temporary registers. Compiled code is at liberty to use d0 to d2 as temporary registers, the rest are reserved for use by the runtime system.

By convention, the result of any operation is left in d0. This includes 'out-of-line' operations such as message sends, object allocation, and primitives, and also 'inline' operations such as generating a constant, or accessing a variable. This is, to some extent, a similar approach to the common implementation strategy for languages such as C where the location of the result of a function call is standardized.

When performing a message send, the receiver must be in d0 and the OOP of the selector in d1. This is concomitant with the convention of d0 being the 'result' register, since computation normally proceeds as a stream of message sends to variables, constants, or the result of the previous message.

### 5.1.3 Stack Discipline

The maintenance of the stack is an important consideration in Smalltalk-80, since the stack contains a mixture of OOPs and non-OOPs and yet must be garbage-collected successfully. Not only are there mixed types of information in the stack, but some return addresses will be into primitives and others into compiled methods. This presents further problems when collecting garbage since compiled methods will move during compaction whereas primitives will not.

The stack is used mainly for the arguments (including the receiver) of a message, space for temporary variables, and for the return and linkage information related to method and block contexts; a typical stack frame is shown in figure 5.3.

To perform a message send, the receiver and arguments are pushed onto the stack, the message selector is loaded into d1, and the dynamic binder is called. By convention, the cleaning up of the stack is not performed by the called method, and must be done by the caller after the send returns.[6]

When a method starts up, the stack contains the receiver and arguments followed by the return address for the call. To these are added the linkage information (the frame and home pointers), the OOP of the method just begun, and as many references to 'nil' as necessary to form the space for temporary variables. The method's own OOP is pushed to simplify garbage collection, as will be explained in section 5.2.1.

[6] In languages such as C, the cleaning up of the stack is very difficult to perform inside the called function (just prior to returning) due to complications involving support for functions which accept variable numbers of arguments. In Smalltalk, the number of arguments accepted by a method is fixed and so the called method could clean up the stack before returning. The code would be smaller, since there are many more message sends in an image than there are methods. The code would be much slower, though, due to the position of the return address in the stack frame (it is *above* the arguments). It would be necessary to pop the return address, save it somewhere safe, modify the stack pointer to remove the arguments and then jump through the saved return address.

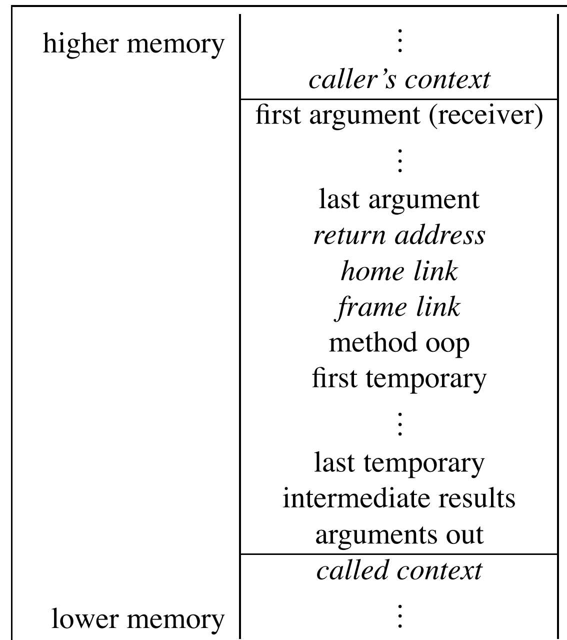


**Figure 5.3:** A typical stack frame. The non-pointer fields are in *italic* type.

The treatment of the stack is more critical in Smalltalk than in many other languages because not all contexts behave in a LIFO fashion. An interesting stack management technique for Smalltalk, where LIFO contexts are handled as efficiently as they would be if pushed on a stack and yet non-LIFO contexts can also be accommodated without having to copy them into the heap, is described in [Mos87].

### 5.1.4 Runtime Support

The image is no longer 'virtual' in the traditional sense, since it contains compiled methods in native machine code. These methods make calls on various parts of the runtime support for dynamic binding, object allocation, primitive dispatch, and so on. This means that some flexible but efficient mechanism for interfacing the runtime support to the compiled methods is required. The mechanism employed here uses the first entry in the object table (which is never allocated, corresponding to OOP 0) to hold the addresses of two dispatch tables containing the entry points to the required support routines, and the entry points to the primitive routines respectively. The code generator and the runtime system need only agree on the indices for the various routines in these tables. This organization allows different versions of the runtime system to be used with the same image without modifying the image in any way.

When the runtime system starts up an image, the first two words in the object table are filled with the addresses of the two tables. The first table (figure 5.4) contains

| ROUTINE NAME | INDEX | FUNCTION |
|---|---|---|
| `Alloc` | 0 | Object allocator |
| `Send` | 4 | Dynamic bind (normal) |
| `Super` | 8 | Dynamic bind (super) |
| `VALERR` | 12 | Value error |
| `NONBOOL` | 16 | Non-boolean receiver |

**Figure 5.4:** Jump table entries for runtime support.

---

dispatch addresses for the object allocator, the message binder, a routine that deals with value errors from block activations (by failing the 'value' primitive), and a routine that deals with non-Boolean receivers encountered in control constructs. The second table contains dispatch addresses for the primitives (indexed by primitive number). All requests on the runtime system are made by calling through one of these dispatch tables:[7]

```
jbsr obtab@(0)@(0..16)   for Alloc, Send, Super, etc.
jbsr obtab@(4)@(N*4)     for primitive number N.
```

For convenience, the *operands* corresponding to `Alloc`, `Send`, `Super`, `VALERR` and `NONBOOL` are defined using these names, which are replaced during a preprocessing pass immediately prior to assembly.[8] Similarly, the appropriate operand for a primitive dispatch can be specified using the `PRIM(N)` macro.

The jump tables themselves are in the runtime system, and the only action required when Native Code Smalltalk-80 starts up is to place the relevant two base addresses in the first two words of the object table.

On calling `Alloc`, `d0` must contain the number of (long) words required for the new object. `Alloc` exits with `d0` containing the OOP of the empty object and `a0` containing the address of the first field of the object in the memory (this will be the class field).[9] Some care must be taken since the garbage collector may be invoked as a result of a call to `Alloc`, so any cached absolute addresses must be invalidated after calling it.

`Send` expects `d0` to contain the OOP of the receiver and `d1` to contain the selector. A dynamic bind is performed and the destination method started up. By convention, the called method leaves its result in `d0`.

[7] All example sequences of 68020 code are written using the mnemonics and operand notation defined by Sun Microsystems in [Sun88], rather than those defined by Motorola in [Mot85].

[8] The assembler source is piped through cpp before assembly, which allows constants and macros to be defined in included files. This makes the assembler form of the compiled image both more digestible to humans (important during debugging) and more compact (many complex operands, for dispatch to runtime support for example, are defined as much shorter mnemonics or macros).

[9] The fact that `a0` contains a pointer to the first word in the body of the newly allocated object was originally merely a coincidence of the design of the object allocator, but turns out to be quite convenient (and is indeed used during BlockContext and Point creation).

Super is almost identical to Send, except that the method lookup begins in a class other than the class of the receiver. For this reason calls to Super must also specify in a0 the class in which to start searching for the method. Since both selector and initial class for the method lookup are known at compile time, it should be possible to statically bind sends to 'super'. However, doing this requires a large amount of support in the image to keep track of dependencies due to the possibility of statically bound methods being overridden at a later time. The performance gains are not likely to be great either, since the method cache is very effective (a dynamic bind that hits the cache will be only a few cycles less efficient than a statically-bound send). The frequency of 'supered' sends is fairly low too, accounting for only 1.6% (statically) and an insignificant 0.075% (dynamically) of sends, averaged over all the benchmarks (a total of 8.2 million message sends).

VALERR is called from within a block if the number of actual arguments does not match the number of expected arguments. It is explained as part of the discussion of blocks in general, in section 5.4.8. NONBOOL is called from within any inlined control constructs ('ifTrue:', 'whileFalse', and their relatives) if the receiver of a conditional is anything other than 'true' or 'false'. The runtime system recovers from this situation by invisibly sending a 'mustBeBoolean' message to the errant value.

### 5.1.5 Omissions in the Runtime System

Several nontrivial pieces of runtime support have *not* been implemented since they were unnecessary for an investigation into the efficiency of the code produced by the compiler. They would, however, be vital in a production system.

The two largest topics ignored by the runtime system are support for Processes and support for some infrequently encountered situations involving blocks. Both of these involve the promotion of contexts held in the machine's hardware stack into full objects held in the object memory, and vice-versa. This becomes necessary when processes are suspended and resumed, a reference to 'thisContext' is encountered, or when a block misbehaves by attempting to return to (or access variables in) a deallocated context. The process-related problems have been ignored completely (although they are the easier of the two to cope with, involving mainly the wholesale conversion of the stack into a linked list of objects, and back again later). Some, but by no means all, of the problems related to blocks have been addressed (section 5.4.9). A thorough treatment of blocks involves the handling of some relatively subtle situations which are described, with some suggestions for solutions, in section 5.4.9. The solutions adopted by ParcPlace in their implementation are described in [DS83].

Many of the essential primitives are not implemented. Since the benchmarks make no use of the keyboard (to choose an arbitrary example), the primitives associated with it are not implemented. Other areas that have been neglected, yet would require runtime support in a full implementation, include Semaphores (which are used heavily by the input event handling mechanism), and operating system interfaces (to the filesystem, for example).

One final area that has been ignored is the very important rôle of the compiler

in debugging. The debugger and compiler have a very close relationship, with the compiler providing the debugger with access to context information and 'reverse maps' of program counter values onto source-level constructs.

## 5.2 Garbage Collection

The garbage collector is of the "standard" compacting mark-sweep type, with nothing particularly remarkable about it at all. The collector is triggered either by executing the 'primGarbageCollect' primitive, or from within `Alloc` if either the object memory or object table overflow; in either case a full mark-sweep-compact cycle is performed. In an attempt to make garbage collection as effective as possible, the allocation ratio of object pointers to object memory words was determined by instrumenting the runtime system and exercising the allocator with a wide variety of activities. This ratio (about 1:4) was used to determine the best split of available memory between the object table and object memory, so that a garbage collection triggered by either object table or object memory overflow occurs very near to an overflow of the other.

The only aspect of garbage collection worthy of further discussion is the treatment of objects referenced from the stack, and the manipulation of absolute addresses within the stack.

### 5.2.1 Garbage Collection and the Stack

The stack presents several problems regarding the collection of garbage. First, many objects, such as temporary variables and arguments to blocks or methods, will be referenced only from within the stack. It is therefore vital to include all object references from within the stack during the marking phase of garbage collection. Second, the stack contains a mixture of object pointers and absolute addresses, which must be treated separately. Since the format of a stack frame is well known (figure 5.3) it is relatively easy to mark the real object references within the stack, ignoring the other information contained in the frames. Third, the stack contains real addresses for linkage between stack frames, and also for return addresses into both methods and primitives. This last point merits further consideration.

Return addresses into primitives can be ignored, since primitives reside in the text segment along with the rest of the runtime system. The type (primitive or compiled method) of any return address can be found easily, since UNIX data segment addresses are guaranteed to be larger than text segment addresses. Comparing a return address against the start address of the object memory gives an indication as to the type of the address.

Return addresses into compiled methods are more problematic since the methods themselves reside in the object memory, and so will move when the memory is compacted at the end of the garbage collection cycle. The solution to this problem is as follows. During the marking phase, whilest object references from within the stack are being considered, the return address in each frame is inspected. Recall that the

OOP of the method forms part of the stack frame (section 5.1.3). Knowing the OOP of the method allows the method's object memory offset to be found. The start address of the method is subtracted from the return address into that method, and this offset stored in place of the original address. When reverse-mapping these addresses, it must be possible to differentiate between offsets into compiled methods and primitive return addresses, so bit zero is set to indicate an offset (the offset is guaranteed to be even since all 68020 instructions must start at an even address). Later, after the object memory has been swept and compacted, the stack frames are inspected once more and the offsets into methods turned back into real return addresses using the new start addresses of the corresponding methods in the object table.

## 5.3 The Compiler Front End

Having fixed the runtime environment and conventions that compiled methods are subject to, we can discuss the compiler itself.

### 5.3.1 Scanning and Parsing

Smalltalk-80 is a fairly simple language to parse. Only 5 token types are needed for scanning, and parse trees are made up of 15 different types of node.[10] The entire scanner and parser for Native Code Smalltalk-80 is fewer than 600 lines of Smalltalk-80 code.

The overall structure of the compiler front-end is shown in figure 5.5. A scanner breaks the input into a stream of tokens that are in turn used by the parser to construct a parse tree. During parsing, a symbol table is created and maintained by a *Mapper*, the main function of which is to translate variable identifiers into parse tree nodes representing the type and location of the variable.[11] The mapper is also required to keep a set of independent parse trees for the block bodies in a method so that code for these can easily be generated out-of-line,[12] as will be discussed in sections 5.4.6 and 5.4.8.

### 5.3.2 The Parse Tree

The result of parsing a method is a parse tree. The root of the tree is a MethodNode which describes many aspects of the method. Each MethodNode contains an array of arbitrary nodes (the roots of subtrees) representing the statements of the method. In

[10] This is more than the number of types of node used in the ParcPlace bytecode compiler since separate nodes are used, for example, for different types of message send.

[11] Mappers have a similar function in Native Code Smalltalk-80 to that of Encoders in PS2.3.

[12] The ParcPlace compilers generate the code for block bodies inline in the method at a point corresponding to the location of the block in the source. They must therefore generate a jump instruction in the body of the method to skip over this code. By saving the parse trees for block bodies separately, and generating code for them at the end of the method, this extra jump instruction is avoided. Versions 2.4 and later of the ParcPlace implementations treat blocks similarly.

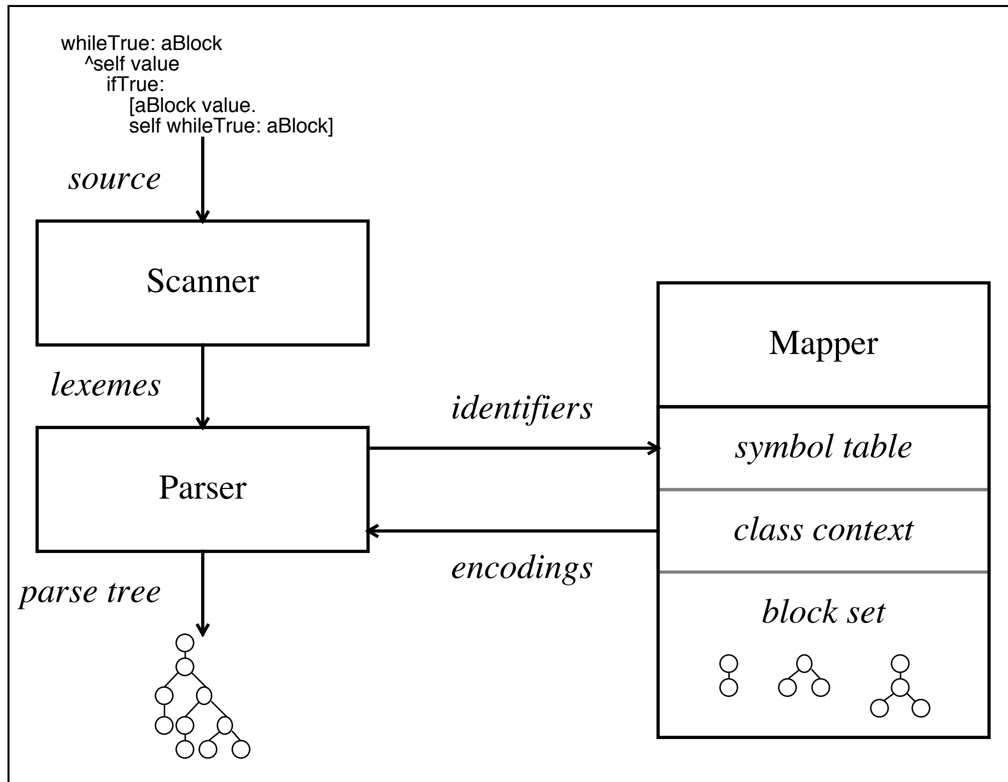


**Figure 5.5:** The scanner, parser and mapper. The mapper maintains a symbol table which it uses with a knowledge of the compilation context to map variable identifiers onto parse tree nodes representing their type and location. The mapper also keeps track of the nesting of blocks, and retains the bodies of out-of-line blocks to be appended to the method during code generation.

---

addition to these, the method node retains a reference to the Mapper that was used during parsing, so that the information about temporary variables and arguments that was gathered during parsing can be used during code generation. The partial parse trees generated for the bodies of blocks that are not expanded inline are not (logically) part of the main parse tree; instead, these partial trees are kept by the Mapper, and are retrieved from there during code generation.

A detailed description of the structure of parse tree nodes can be found in appendix A.

### 5.3.3 Optimizing the Parse Tree

Very little can be done to optimize a Smalltalk parse tree: the popular parse tree optimizations such as common subexpression elimination cannot be safely applied. About

the only thing that can be done is constant folding, where a node for a simple arithmetic operator with both arguments kinds of Number is replaced by a LiteralNode representing the result of evaluating the expression at compile time.[13] This can only be justified by the fact that the operation would have been inlined in the final generated code, so any changes made by the user in the definitions of the arithmetic selectors would not have had any effect anyway.[14] In the next chapter we will see how even optimizations on the parse tree such as constant folding can be performed implicitly during code generation.

## 5.4 Code Generation

We are now in a position to develop a naïve code generator which will produce correct code for any Smalltalk-80 construct. It should be stressed that the code generator will initially be as simple as possible, with some optimizations and enhancements added later. The main purpose of this simple code generator is to provide a reference point for the development of a more sophisticated code generator which will be the subject of chapter 6.

### 5.4.1 Overview of Code Generation

Each parse tree node responds to the message 'generate'. Leaf nodes will respond by placing code to generate the quantity they represent on a 'code stream'. Non-leaf nodes will respond similarly, but will also propagate the 'generate' to their descendants at the appropriate time. Code generation for the whole method is thus initiated by sending a single 'generate' message to the root of the tree (a MethodNode).

Parse tree nodes do not explicitly place instructions on the code stream, but send 'emit' messages to an object representing the machine for which code is to be generated. By standardizing the interface between the parse tree nodes and the machine object, different back ends can be used with the same front end to generate code for different architectures. This means that optimizations performed on the parse tree itself are independent of the final target architecture. The interface is designed to follow closely the operations supported by most CISC architectures. For example, the message 'emitMove: *source* to: *dest*' causes an instance of M68000 to append a '`movl` *source*`,` *dest*' to its instruction stream.

Just as the parse tree in a recursive descent compiler can be implicit in the call graph of the compiler itself, so the sequence of 'emit' messages sent from the parse

[13]Since the expression is evaluated using exactly the same semantics as are in force at run time, any overflow results in a LiteralNode containing a LargeInteger. Operations such as division could equally well result in a literal Fraction or Float.

Constant expressions are fairly uncommon in typical Smalltalk images. Out of all methods in the image used to cross-compile the benchmarks, only seven constant expressions were folded into literals.

[14]Just as in the ParcPlace compiler, it is possible to change the set of selectors that are subject to inlining to cater for the rare cases where non-standard behavior is required. In the native code Smalltalk compiler, the set of operations that are considered during constant folding can also be modified easily.

tree to the machine object represents an implicit intermediate representation of the compiled code.

For convenience we will discuss the critical code fragments generated by the compiler for each node type in turn starting with leaf nodes for constants and variables, then moving on to assignment and messages sends. Last, the required method entry and exit sequences generated by MethodNodes, along with the treatment of block bodies, are explained. Each node type sends a handful of different ‘emit’ messages to the machine, most of these being specific to a particular type of node. Only a few very common operations (emitting a label or loading ‘nil’, for example) are sent from more than one node type.

Most of the complexity is associated with code fragments generated for optimized constructs such as inlined control structures. These will be dealt with later, as each optimization is introduced. First though, a brief description of the workings of the ‘machine’ object to which the parse nodes send the ‘emit’ messages.

### 5.4.2 Machine Objects: the M68000

Machine objects encapsulate all of the machine-dependent information required for code generation. Parse tree nodes do not generate code directly, but instruct an instance of a particular class of machine to generate the code on their behalf. By standardizing the interface to machine objects it is possible to plug different back ends into the same compiler front end, for example

```
aParseTree generate: aMachineClass
```

causes ‘aParseTree’ to send a stream of ‘emit’ messages to the instance of ‘aMachineClass’, thereby translating the tree into code suitable for that machine. The machine class of interest in the present discussion is the M68000 which responds to ‘emit’ messages by generating instructions for the 68020.[15]

A M68000 machine object has three main tasks. It must create and maintain a M68000CodeStream which is similar to an InstructionStream but specialized for 68020 instructions and operations thereon. It must also keep track of certain state information such as the number of items on the stack below the frame pointer for the last instruction generated (information such as this is used, for example, during the calculation of a stack-pointer relative address when retrieving the receiver from further up the stack during a message send), and the identity of the next temporary label to generate (for use as branch destinations in loops and conditionals). Finally, it must implement all the ‘emit’ messages that can be sent from parse tree nodes to append representations of 68020 instructions to the instruction stream.

In addition to these responsibilities, the machine object in a naïve compiler is responsible for implementing any peephole optimizations that may be required.

[15] A machine class called ‘VM’ also exists which responds to ‘emit’ messages by generating bytecoded methods suitable for interpretation by a Blue Book virtual machine.

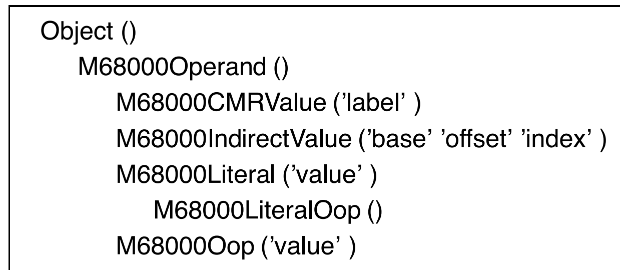


**Figure 5.6:** The class hierarchy for 68000 operands. See text for details.

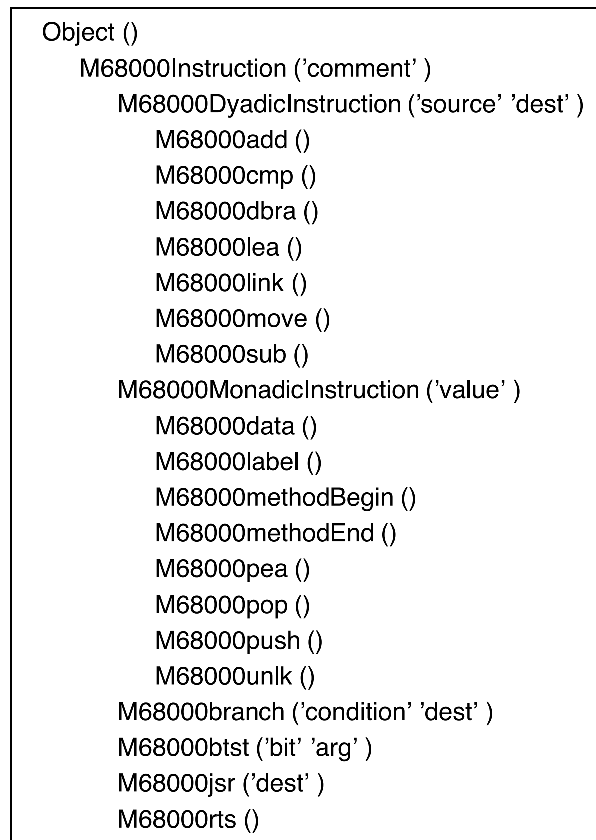


**Figure 5.7:** Class hierarchy for 68000 operations. See text for details.

The 68020 instructions themselves are contained in two small hierarchies, one formed from classes representing instructions, rooted at M68000Instruction, and the

other formed from classes representing operands, rooted at M68000Operand. These hierarchies are shown in figures 5.6 and 5.7, and explained below.

CMRValues represent offests relative to the start of a compiled method, so they are associated with a particular 'label'. They are used mainly to represent the start of a block body within a method. IndirectValues represent memory locations whose address is in some 'base' register, with optional 'index' register and integer 'offset'. These represent variable locations within the stack, where the 'base' would be the `frame` pointer with 'offset' corresponding to the required field within the stack frame, and locations within the object table or object memory (global variables, for example). LiteralValues represent 'immediate' integer values corresponding mainly to integer literals in the Smalltalk program. LiteralOops represent 'immediate' object pointer values, being used mainly to identify classes and selectors where necessary. Oops represent non-immediate object pointers and usually occur as the 'offset' field of an IndirectValue during an operation on a global variable.

M68000Instruction is the root for all instructions, providing default behavior for messages such as 'isBranch' in the same manner that Object does for messages like 'isNil' in the image as a whole. It also provides support for comments which are included in the assembler representation when present. (Instructions that refer to a selector or variable usually place the associated identifier in the 'comment' field to make the compiled version more comprehensible to human readers.) Its subclasses are more or less self-explanatory and most fields in the computational and data movement instructions can be any kind of M68000Operand. M68000data represents literal data in the instruction stream (used to insert the method OOP into a compiled method). M68000label represents a label: its 'value' is an integer identifying the label (labels are local to each compiled method, so the label allocator can start anew for each method). M68000methodBegin and M68000methodEnd are pseudo-instructions used simply as a shorthand notation for the method entry and exit sequences; their 'value' is the selector of the method. M68000branch represents both conditional and unconditional branches: if the 'condition' field contains a symbol representing a condition code ('#eq' for example) then the branch is conditional, otherwise it is unconditional. M68000btst tests a single bit in a value. The 'bit' field specifies the bit (zero is the least significant), with 'arg' being any M68000Operand.

### 5.4.3 Code for Leaf Nodes

Leaf nodes represent either literal constants or variables. Constants are encoded in an instance of LiteralNode, which contains the constant itself. When a M68000 is sent an 'emitLiteral:' message, with the relevant literal as the argument, it will generate a '`move`' instruction to set `d0` (the result register) to the value of the literal. Two types of literal are differentiated by the code generator, SmallInteger constants and other (object pointer) constants:[16]

[16]The use of the '`I()`' and '`O()`' macros allows changes to be made to the representation of SmallIntegers and object pointers trivially. In the 68020 implementation, '`I(N)`' expands to '`(N*2+1)`' (SmallIntegers are twice their value with the bottom bit set) and '`O(P)`' to '`(8*P)`' (object pointers are byte

```
movl #I(INT), d0   | for SmallIntegers
movl #O(OOP), d0   | for object numbers
```

Arguments and temporaries are represented by instances of ArgumentNode and TemporaryNode. Each of these contain an index identifying the variable. By convention, the receiver has index 0, the first argument has index 1 and so on. Similarly, the first temporary has index 1, the second 2, and so forth. These indices are used as offsets from the frame pointer to address a word in the stack, either above the frame pointer (for arguments) or below the frame pointer (for temporaries) (stack frames are explained in section 5.1.3). Nodes for arguments and temporaries respond to the 'generate' message by sending an 'emitArgument:' or 'emitTemporary:' message to the machine, with the argument or temporary node itself as the argument. An M68000 responds to these by placing one of

```
movl frame@(8+4N), d0    | for argument N
movl frame@(-4-4N), d0   | for temporary N
```

on the instruction stream, as appropriate. Nodes representing 'self' and 'super' are treated as if they were ArgumentNodes with index zero (argument zero is always the receiver).

In the case of instance variables, InstVarNodes carry an index (as for arguments and temporaries), which specifies which 'field' of the object is occupied by that variable. When asked to 'generate' code, InstVarNodes send an 'emitInstVar:' to the machine (with the node as the argument, as before) which then generates instructions to load the variable into the result register.

When accessing an instance variable it is first necessary to load the OOP of the receiver into a register, use this to fetch the object memory offset of the receiver into another register, and finally to use this offset as the base from which to access the required field of the receiver. For example, Points have two instance variables, 'x' and 'y'. Allowing for the two word object header, we can fetch 'x' from a Point as follows:

```
movl frame@(16), d1        | receiver
movl obtab@(d1:l), a1      | start of receiver in memory
movl obmem@(8,a1:l), d0    | 'x' is first inst. var, offset 8
```

When generating the code for the body of a block, the `frame` pointer can no longer be used to access the arguments (including the receiver) and temporaries of the home context since it will have been set to point to the start of the active *block* context by the 'value' primitive to allow access to the arguments of the block itself. The compiler must instead generate sequences that use the home pointer, which will be a copy of the `frame` pointer that was in effect at the time the block was created. The Mapper (which is responsible for maintaining the 'context' in which the method is compiled) keeps track of whether the compiler is currently generating code for the body of a method or block, the code generator checking with the Mapper when generating code to access

offsets into the object table, so must be multiples of 8).

temporary or argument variables. This is explained in full in section 5.4.8. Apart from this one exception, the sequences generated for leaf nodes within blocks are the same as those generated within methods.

Nodes for global variables contain a pointer to the Association corresponding to the variable; as might be expected, they generate code for themselves by sending the machine an 'emitGlobal:' message which generates a short instruction sequence to extract the value field of the association. This is done in two instructions: the object memory offset of the association is first loaded into `a0`, then the second instance variable of the association (the 'value' field, offset 12 from the start of the association) can be loaded into `d0`:

```
movl obtab@(O(1622)), a0   | assoc. for Rectangle, OOP 1622
movl obmem@(12,a0:l), d0   | fetch value field
```

### 5.4.4 Assignment Statements

Having seen how code generation works for simple exterior nodes, it is time to tackle the simplest complete statement in Smalltalk – assignment.

AssignmentNodes contain two subtrees: one each for the left- and right-hand sides of the assignment (destination and source, respectively). The right hand side is first asked to 'generate' code for itself. The result will be left in `d0` irrespective of the type or complexity of the expression involved. The machine is then asked to 'emitAssignTo:' with the destination as the argument. Since the destination *must* be a kind of VariableNode, code very similar to that given above for leaf nodes can be produced, except that the last line

`movl` *someLocation*`, d0`

must be reversed to

`movl d0,` *someLocation*

(Note that the code fragments for variable access do not use `d0` as a temporary 'scratch' register; its contents will consequently be preserved during an assignment to any type of variable, remaining available as the result of the assignment.)

### 5.4.5 Message Sends

A message send is effected by pushing the receiver followed by the arguments (if any) onto the stack, calling `Send` or `Super`, and cleaning up the stack by popping the arguments (section 5.1.3 includes a diagram of a typical stack frame due to a message send). The three types of message node (UnaryNode, BinaryNode and KeywordNode) respond to 'generate' messages by asking the machine to 'emitUnary:', 'emitBinary:' or 'emitKeyword:' as appropriate. As usual, the argument is the node sending the 'emit' which gives the machine complete control over any optimizations that may be applied to specific selectors.

Ignoring optimizations for the moment, the machine responds to these 'emit' messages by sending a 'generate' message to the receiver, and each of the arguments in turn, pushing each result (which the reader will have noticed by now is always returned in d0) onto the stack in preparation for the impending dynamic bind. Once receiver and arguments are pushed on the stack, the OOP of the selector is extracted from the message node, loaded into d1, and a dynamic bind performed by calling 'Send' or 'Super' as appropriate.[17] If the message is a send to 'super', implying that the binder to be used is Super, then one extra instruction is emitted to load the OOP of the initial class for the method lookup into a0. The only thing left to do after returning from the called method is to pop the arguments off the stack by adding a constant to the stack pointer.

The only difference in the code generated between the three different types of message node is the number of arguments pushed onto the stack. In general a message send looks like:[18]

```
generate receiver in d0
movl d0, sp@-

generate argument in d0     | repeated for
movl d0, sp@-               | each argument

movl sp@(4*nArgs), d0       | recover receiver

movl #classOop, a0          | if sending to Super only!

jbsr Send                   | or Super
addw #4*nArgs, sp           | pop args
result in d0
```

All types of message node (UnaryNode, BinaryNode and KeywordNode) use a common method within the code generator to effect a message send, one parameter of which is the number of arguments in the message send. For unary and binary nodes, the number of arguments is implicit in the type of the node (zero and one respectively). KeywordNodes keep an (ordered) collection of subtrees representing their arguments. The size of this collection is passed to the code generator to indicate the required number of arguments in the message send.

Cascaded message sends are very similar to normal sends. The code produced to generate the receiver, initial arguments and first message send is identical to that just given. The first difference appears in the tidying up of the stack after the initial send, which pops one fewer words than normal, leaving the receiver on the stack as

[17]The correct binder is determined by inspecting the receiver: if it is a reference to 'self', *and* the 'supered' flag is true, then the bind is performed through 'Super', otherwise through 'Send'. These are two entry points to the same routine that simply pick a different class in which to start the method lookup.

[18]This scheme does not make the number of arguments in the message send explicit, which could cause problems if the receiver does not understand the message. See appendix E for a more detailed discussion of this problem.

the recipient of the subsequent messages in the cascade. The arguments to the second send in the cascade are then pushed, the receiver retrieved from further up the stack, and a dynamic bind performed again. This pattern is repeated until the last message has been sent, at which time its arguments *and* the receiver are popped of the stack.[19] This is so straightforward that an example should not be necessary.

### 5.4.6 Method Entry and Exit

Several operations are required at the entry and exit points of methods in order to maintain the execution environment introduced in section 5.1. MethodNodes communicate the required actions to the machine object via four 'emit' messages. The first is concerned with the method entry sequence:

emitMethod: aClass selector: aSymbol

> generates the method entry sequence, including linkage for the `frame`, `home` and stack pointers, and initialization (to 'nil') of the temporary local variables on the stack.[20]

As described earlier, to send a message a method must push the receiver followed by the arguments, load `d0` with the receiver, load `d1` with the selector, and then call `Send` (or `Super`). This leaves the arguments followed by a return address on the stack when the destination method is entered. It is the responsibility of the called method to build and initialize a new stack frame, and update the frame and stack pointers appropriately.

Regardless of any requirements for temporary variables, all methods must save the two frame pointers (`frame` and `home`) before they are changed to refer to the new context. The garbage collector expects the next word on the stack to contain the OOP of the method (section 5.2.1), so this is pushed next. After this the method must reserve space in the stack for any temporary variables that it requires. This can be done neatly in parallel with the initialization of these variables by simply pushing the required number of pointers to 'nil' on the stack. The method entry sequence therefore consists of:

```
pea    home@        | save home pointer
pea    frame@       | save frame pointer
movl   sp, frame    | new frame pointer
pea    0(36265)     | method OOP
pea    0(1)         | zero or more pushes of NIL
                    |   to reserve temporaries
```

Using this scheme, arguments are accessed by positive offsets from the frame pointer, and temporaries by negative offsets.

[19]The receiver is protected while on the stack since assignments to 'self' are outlawed during syntax analysis.

[20]The Mapper keeps track of the number of temporary variables required.

The body of a method is represented as an ordered collection of nodes, one per statement. Sending each node a ‘generate’ message causes code for the body of the method to be generated.

Having generated code for the body of the method, it may be necessary to supply an exit sequence (this will be the case if no explicit return statement was present):

emitLocalReturn
: generates a code sequence to perform the implicit ‘local’ return at the end of a block or method.

The exit sequence for a method, whether the return is requested explicitly with a return statement or an implicit return of ‘self’, is generated by sending the machine an ‘emitLocalReturn’ which restores the previous stack frame and resumes execution of the previously active method (this will be done by sending the ‘generate’ message to the return node if an explicit return statement ended the method). This is done by unlinking the frame pointer in the usual way, and then popping the home pointer. After this a normal ‘`rts`’ will return control to the caller. The sequence is therefore:

```
unlk   frame        | deallocate stack frame
movl   sp@+, home   | recover home pointer
rts                 | return to caller
```

In section 5.3.1 it was mentioned that the bodies of blocks are not compiled inline, but added to the end of the method. One more message is used by MethodNodes during the generation of code for these blocks:

emitBlockPrelude: aBlockNode
: generates the code to check the number of arguments sent to a block, copy the arguments into the home context, and then rebind ‘self’ and ‘home’ to the values they had in the home context.

Code for blocks is generated by sending the machine an ‘emitBlockPrelude:’ message to generate code for argument checking and context binding, sending the BlockNode a ‘generateBody’ message to generate the code for the block’s statements, and then sending the machine an ‘emitLocalReturn’ to terminate the block if no explicit return was provided.[21] Blocks will be discussed in much more detail in section 5.4.8.

### 5.4.7 Primitive Methods

PrimitiveNodes make just one request to the code generator:

emitPrimitive: primIndex
: generates the primitive dispatch code described earlier in section 5.1.4

[21]The generation of code for block bodies is split in this way for the benefit of the macro selectors that inline their block arguments: the natural response of a BlockNode to a ‘generateBody’ message must be to generate code for the statements of the block, and nothing more.

To invoke a primitive, a call through a dispatch table at the start of the object table is made. If the primitive fails, it returns with a normal '`rts`', to the instruction following the dispatch, which allows the fail case code to immediately follow the call to the primitive. If the primitive is successful it will pop the topmost stack frame before returning, thereby transferring control back the the method that sent the message which invoked the primitive:

```
jbsr PRIM(N)    | primitive number N
```
*fail case code....*

This is illustrated graphically in figure 5.8.

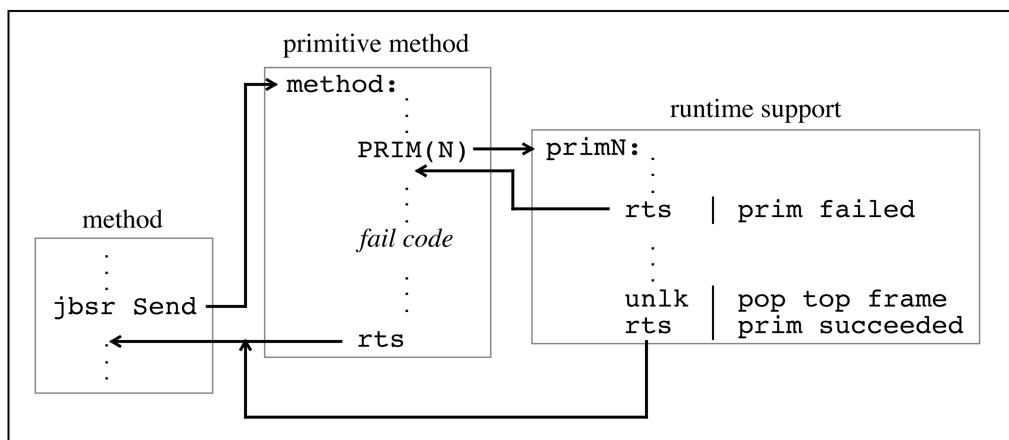


**Figure 5.8:** A primitive method begins with a call through the primitive dispatch table. Primitive failure is indicated by a normal return that continues with the fail case code, whereas a successful primitive pops the topmost stack frame and returns to the sender of the primitive message.

---

### 5.4.8 Code for Blocks

Some blocks that appear in the source (as arguments to the control selectors for example) are optimized into inline statements; these will be discussed along with other optimizations a little later (section 5.5.2). Of more immediate concern is the code generated within the body of a method corresponding to a block in the source, and (completely separately) the code generated around the statements that form the body of the block. These block bodies are kept separate from the main body of the method, and are appended to the generated code when the compilation of the method itself has finished. Efficiency is the only motivation for this, since the normal jump around the body of a block (necessary in bytecoded methods) is no longer required.

When a BlockNode is encountered, during the parse tree walk for the generation of code for a method, it will be sent a 'generate' message just like any other kind of node.

In response to this the BlockNode must ask the machine to generate code to create a BlockContext inline in the method:

emitBlock: aBlockNode
: generates code to create a BlockContext inline and initialize the home context, parent method, and initial-offset fields of the new BlockContext.

Having done this the BlockNode enters itself into the Mapper's block set, and in doing so ensures that it will eventually be asked to generate code for its body (by being sent a 'generateBody' message) to be appended to the method (see the discussion of MethodNodes, section 5.4.6).

Before discussing the actual code generated to implement blocks, a short digression is necessary to explain the mechanism used for accessing the arguments passed to a block. In section 2.1.2 it was explained that the arguments to a block, as well as any temporaries named inside the block, are held in the block's home context (recall the example of setting the value of a method temporary by invoking a single-argument block with no body). Since a block can be activated at an arbitrary time in an arbitrary context, the usual mechanism for accessing temporaries using the `frame` pointer will not produce the correct results. It is therefore necessary for a BlockContext (the object that represents the block, and which responds to the 'value' messages) to keep a copy of the `frame` pointer that was active at the moment the block was created. When the block is later activated, this value is used to initialize the home pointer, through which the arguments and temporaries within the block are accessed.[22]

In addition to a copy of the active `frame` pointer, a block context must retain two other pieces of information: the OOP of the method in which the block originated (since this method contains the code for the block's body), and the offset into this method at which the body begins.

The sequence generated to create a BlockContext thus creates a new object (by calling `Alloc` with `d0` containing the size of a block context), and then initializes the class of the object and the three 'instance variable' fields with the active frame pointer, the originating method, and the offset into the method ('initial PC') at which the body of the block begins:

```
movl #24, d0               | size of BlockContext
jbsr Alloc                 | create a new BlockContext
movl #BlockContext, a0(4)  | class field
movl frame, a0@(8)         | home context
movl #methodOop, a0@(12)   | originating method
movl #bodyOffset, a0@(16)  | initial PC
```
*OOP of block context left in d0*

[22]It is entirely possible that the context to which the home pointer refers has already been deallocated when the block is started up (section 5.4.9). The runtime system makes no attempt to detect and cope with this situation (section 5.1.5), although the remedies are well known and thoroughly documented ([Mir87] and [DS83] for example). Such remedies do however complicate accesses to state within non-LIFO contexts, which might have to be performed using an out-of-line routine with considerable performance overheads. See appendix E for a more detailed discussion of this problem.

Note that for nested blocks, the home context for a new inner block is *not* the active context, but the home context of the outer originating block. Therefore, in these situations the home context is initialized slightly differently:

```
movl home, a0@(8)    | home context
```

(the code generator can easily determine which sequence to generate by asking the Mapper whether the code being generated is within a method or a block, just as it does when deciding which frame pointer to use for temporary variable access – see below).

Each method is bound to a unique selector which implicitly determines the number of arguments expected, so it is impossible to invoke any method with the wrong number of arguments. This is not true in the case of blocks, since they are essentially 'anonymous' methods with no selector to imply the number of arguments expected. It is therefore necessary to explicitly check the number of arguments passed to a block before the body can be executed. The number of actual arguments passed to a block is determined by the message used to activate it (a block activated with 'value' must have no arguments, a block activated with 'value:value:' must have two, and so on). These (primitive) 'value' methods load `d1` with the number of actual arguments, which is checked during the prelude to the block body.[23] If the number of actual arguments does not match the expected number of arguments, then the 'value' primitive that invoked the block is failed (it simply returns without unlinking the topmost stack frame) by transferring control to the `VALERR` routine.[24]

The last action that must be performed by these primitives is to load the `home` pointer from the saved `frame` pointer in the BlockContext.

One last detail must be taken care of before code for the statements of the block body can be generated. Since the arguments will reside in the context of the method from which the block was activated, these must be copied back into the space allocated for them in the home context (the context of the method in which the block was created).

After this block body prelude, the BlockNode can be sent a 'generateBody' message which will cause it to generate code for the statements that make up its body. Argument and temporary variable accesses must be performed through the `home` pointer rather than the `frame` pointer, since these variables reside in the context of the defining method rather than the context of the block's activation. The machine determines which is appropriate by checking with the Mapper whether the current statement was located in a method or block body.

Blocks exit by one of two mechanisms: they either provide an implicit return (of the value of the last statement executed within them) to the context in which they were activated (a 'local' return), or an explicit return to the parent of the context in which they were created (the notorious 'non-local' return).

---

[23] Another approach would be to store the expected number of arguments in the BlockContext itself, and have the 'value' primitives check this before activating the block.

[24] This is more efficient in both space and time than branching over an inline '`rts`' if the argument count is correct.

The code generated for a local return is identical to that generated for a return at the end of a method (section 5.4.6). For non-local returns, the situation is slightly more complex. The home pointer will contain the frame pointer for the context in which the block was defined, so a return can be accomplished by moving the home pointer into the frame pointer and performing a normal return sequence. This effectively extends the current stack frame down to the one in which the block was defined. The non-local return sequence is as follows:

```
movl home, frame   | extend stack frame to defining method
unlk frame         | unlink and return as normal...
movl sp@+, home
rts
```

So, the code for the body of a block follows this pattern:

```
cmpw #numArgs, d1
jne VALERR                     | fail if value error
movl sp@(8), home@(Arg1)       | copy first arg
.                              | copy
.                              | intermediate
.                              | args
movl sp@(4+4N), home@(ArgN)    | copy last arg
block body
movl home, frame               | for non-local returns ONLY!
unlk frame
movl sp@+, home
rts
```

### 5.4.9 Block Problems

In both PS2.3 and Native Code Smalltalk-80, block contexts do not retain their creation-time environment but rely on their home context (the context of the method in which they were created) for this. However, since blocks are first-class objects they can be stored for activation at an arbitrary time, possibly even after the nominal demise of their home context. This is not a problem in implementations based faithfully on the Blue Book, since all contexts (for both methods and blocks) are created as fully-fledged objects which are reference-counted in the normal fashion. A method context will *not* be garbage collected while one or more block contexts continue to exist which have it as their home.

This is all well and good in theory, but most processors are designed to work well with contexts held on a strictly LIFO stack, suffering crippling losses in performance if they are forced to work with contexts held in a heap (such as an object memory).

Although a few very early experimental Smalltalk implementations followed the Blue Book virtual machine definition faithfully (such as [Wol84] and the implementations described in [Kra83]), the performance losses associated with heap-based contexts quickly influenced the development of schemes that kept contexts on the real

hardware stack, where they rightfully belong [Mir87] [DS83].[25] These machine-friendly contexts are then promoted into full objects only when absolutely necessary. Section 3.1.1.2 describes the ParcPlace approach in some detail.

Although far from insurmountable, these block problems were not addressed in Native Code Smalltalk-80 which always keeps contexts in an execution-oriented form on the hardware stack. Since none of the benchmarks used suffer from unsavory behavior by blocks, there was little point in implementing any mechanisms for coping with object-based representations of contexts.[26] Cures for this problem invariably rely on the use of multiple representations for contexts, the representation at any time depending on the manner of use of that context at that time. The usual conversion would be to promote contexts from an execution-oriented form into a full object for retention in the object memory. The mechanisms required are well understood and are tedious rather than difficult to implement. For this reason, their implementation was considered beyond the scope of the work undertaken for this thesis. Most of the information required about the behavior of an arbitrary BlockContext is available at compile time, so arranging for contexts to be promoted into full objects when they exit is simply a matter of augmenting the return sequences when relevant.

Another problem with blocks is related to non-local returns, which (for any given block) always return control to the same context (the 'sender' context of the context in which the block was created). If this context has already been returned from (which will be the case if the block is stored and executed more than once, or even if it is executed just once after the sender has exited through a normal series of method returns), there will be no context for the block to return to. In Blue Book implementations this condition is detected by storing 'nil' in the instruction pointer of the active context during a return, and then checking for a 'nil' instruction pointer in any context being returned to. Since it is not at all clear how this situation should be recovered from, Smalltalk simply raises a 'cannotReturn' error and leaves the user to sort out the mess.

This problem would be a little more tricky to sort out in Native Code Smalltalk-80 since it is impossible to predict at compile time which contexts will be on the receiving end of these badly behaved non-local returns. A simple but effective solution would be to arrange for any methods that export blocks which perform non-local returns to mark their 'sender' contexts at runtime, promoting these to full objects (and marking them as having already been returned from) when they exit.

---

[25]The Blue Book implementation described in [Wol84] only ever keeps contexts in the heap, and approximately 25% of the virtual machine's time is spent solely in reference counting these contexts. This may seem quite high to begin with, but this figure ignores the considerable memory-management overheads associated with the immense rate of creation and destruction of these objects.

[26]This is not strictly true. An obscure problem can occur with SortedCollections which keep a reference to a 'sortBlock' for use in ordering the contents of the collection. Due to the nature of BlockContexts, simply creating such a block and storing it for later use will result in a circular chain of contexts, which in some cases could be quite large. These will not be garbage collected by the normal reference counting mechanism, being discarded only during a full mark-sweep garbage collection. For this (and a few other similar cases) the BlockContext is sent the 'fixTemps' message which makes a copy of the block's 'home' context (thereby 'fixing' the values of any shared temporaries and arguments, hence the name) and then 'nil's out its 'sender' field, breaking the circularity.

The solutions to both of these problems require contexts to exist in two distinct forms: in a machine-oriented form on the hardware stack for use during execution, and in the form of a fully-fledged object in the object memory for use when a deviation from strictly LIFO behavior is required. Various mechanisms have been developed to cope with this duality (triplicity in the case of PS2.3, which uses three representations for contexts) and any of them could be adopted by Native Code Smalltalk-80. Again, the details are tedious rather than novel and so were considered beyond the scope of the work undertaken for this thesis.

A more radical attack on these problems is probably relevant in this enlightened era. Blocks should be implemented as full closures (this has been done in versions 2.4 and later of ParcPlace distributions, and a few notable independent implementations such as BrouHaHa [Mir87]). This brings other benefits including the separation of block and method temporaries and arguments that happen to share names, and the ability to provide block temporaries that obey lexical scoping rules. The problem of multiple returns to the same context can (and should) be tackled by applying continuation semantics to non-local returns, with enough runtime state being preserved to allow these returns to be activated an unlimited number of times.[27] The use of continuation semantics in association with non-local block returns in a Smalltalk-based language is currently being investigated; some conclusions can be found in [Wol88].

## 5.5 Optimizations

Apart from the usual trivial optimizations performed on the parse tree, the important optimizations in Smalltalk are the inlining of the 'special selectors', the macro-expansion of control selectors with the consequent inlining of their block arguments (which is performed during code generation), and peephole optimizations performed on the code after it has been generated.

### 5.5.1 Inlining Special Selectors

Some gains are made in bytecoded Blue Book implementations of Smalltalk by the special treatment of 32 messages that have a large impact on overall system performance (section 3.1.2). These 32 'special selectors' include 16 arithmetic operations, testing for equivalence ('=='), fetching the class of an object, the 'blockCopy:' message, two of the block activation messages 'value' and 'value:', and eleven commonly sent messages (collection accessing messages, Point creation and accessing, and so on). The default set of special selectors is shown in figure 3.1.

[27] Since contexts are first-class objects in Smalltalk, it is a trivial matter to implement explicitly-managed continuations (created and invoked in a manner similar to that used in Scheme [Dyb87]) by manually copying each context in the 'sender' chain on the stack. By making further copies of this chain as necessary, the topmost context can be restarted as many times as required. This can all be implemented directly in Smalltalk, in a handful of very short methods.

In PS2.3 these messages are not sent using the normal send bytecodes. Instead the compiler emits one of 32 'special send' bytecodes which the virtual machine recognizes as a send of one of the special selectors. When the virtual machine encounters a special send of one of the 16 arithmetic selectors it invokes the relevant primitive in class SmallInteger directly (thus avoiding the overhead of a normal method lookup) assuming the common case which is that the arguments are SmallIntegers. If this assumption is incorrect then the primitive fails and a normal lookup is performed to complete the send. Literature on Smalltalk commonly refers to sends of this type as being 'short-circuited' to a primitive.

Five out of the remaining 16 selectors are treated in a similar fashion, causing the appropriate primitive to be invoked after a few checks on the class of the receiver (these checks are fairly straightforward, and are shown in the definition of 'commonSelectorPrimitive' on page 619 of [GR83]).

The remaining eleven special sends cause a normal message lookup and serve only to save space in the method's literal frame, and to reduce the size of the method by one byte per special send. Consequently they are used for the eleven most commonly sent messages. Since these sends serve only to specify the selector implicitly, it is trivial to change the set of common selectors treated in this way.[28]

A similar optimization can be applied in a native-code Smalltalk implementation, where the back-end is at liberty to produce whatever code it sees fit in order to implement the operation requested when it receives an 'emit' message.[29] The code to perform the action associated with a special selector is compiled inline, with some suitable class check on the receiver and arguments (if any). The selectors chosen for this treatment in native-code Smalltalk are the arithmetic operations and comparisons on SmallInteger quantities. The code produced for all arithmetic operations follows the same pattern:

```
           receiver on stack, argument in d0
           btst   #0, d0          | SmallInteger receiver?
           jne    fullSend
           btst   #0, sp@(3)      | SmallInteger argument?
           jne    fullSend
           perform operation, popping stack, result in d0
           jra    continue
fullSend:  movl   d0, sp@-
           movl   sp@(4), d0
           movl   #O(selector), d1
           jbsr   Send
           addql  #8, sp
continue:
```

[28]The compiler and runtime system communicate through an array with a well-known OOP (the class variable 'SpecialSelectors' in SystemDictionary) which holds pairs of Symbols and SmallIntegers representing the special selectors themselves and the number of arguments they require.

[29]The code generator is the correct place to choose implementation strategies for any particular operation, including the inlining of special selectors, since it is the code generator that has the intimate knowledge of the target architecture's capabilities necessary for this choice.

Bit zero of receiver and argument will be set if they are SmallIntegers. If either argument is not a SmallInteger, then the inlined version is abandoned and a full send performed. If both argument and receiver are SmallIntegers, then the relevant operation is performed inline.[30]

The code produced for the comparisons is similar, but the inlined operation itself must result in 'true' or 'false' being left in d0:

```
            receiver on stack, argument in d0
            btst  #0, d0
            jeq   fullSend
            btst  #0, sp@(3)
            jeq   fullSend
            cmpl  sp@+, d0
            s??   d0           | see section 6.5.1.2
            andw  #8, d0
            addw  #TRUE, d0
            extl  d0
            jra   continue
fullSend:   perform full send as before, result comes back in d0
continue:
```

The only other selectors considered for inlining are those for the creation of Points, and the two accessing messages 'x' and 'y'.[31] Point creation is straightforward:

```
abscissa second on stack, ordinate on top of stack
movl  #16, d0          | size of a Point
jbsr  Alloc            | allocate object
movl  #POINT, a0@(4)   | class field
movl  sp@+, a0@(12)    | ordinate
movl  sp@+, a0@(8)     | abscissa
new point in d0
```

The class of the receiver should first be checked, since '@' is only defined for Numbers. However, this check is *not* performed by the ParcPlace virtual machine which will quite happily construct a Point with arbitrary objects as the abscissa and ordinate, so Native Code Smalltalk-80 does not perform the check either.[32]

[30]Although this may appear to break the semantics of the language, in that redefining the behavior of a special arithmetic operation in class SmallInteger will have no effect at all at runtime, it is completely commensurate with the language definition: any true Blue Book implementation would behave this way, due to the short-circuiting of the special arithmetic selectors.

Note that the inline code shown in the text does not check for overflow; this could be done in software (a branch to the full send) or in hardware (the 68000 series supports an overflow trap triggered by the 'trapv' instruction).

[31]These three selectors were targeted in both the ParcPlace implementations and in Native Code Smalltalk-80 due to the importance of Points in the system. Points pervade the entire user interface and are used heavily during any graphic, character scanning, or bit block-transfer operation.

[32]This has been corrected in more recent versions of Smalltalk-80.

Simulating a send of 'x' or 'y' to a Point requires a check on the class of the receiver with the relevant field being extracted explicitly if the check succeeds. As with all other inlined operations and comparisons, if the the class check fails then a full send is performed:

```
            point in d0, perform 'aPoint x'
            btst    #0, d0                  | SmallInteger receiver?
            jne     fullSend
            movl    obtab@(d0:l), a0        | memory offset
            cmpl    #POINT, obmem@(4,a0:l)  | class == Point?
            jne     fullSend                | no, do full send
            movl    obmem@(8,a0:l), d0      | yes, get abscissa
            jra     continue
fullSend:   movl    d0, sp@-
            movl    #O(selector), d1        | 'x'
            jbsr    Send
            addql   #4, sp
continue:
```

### 5.5.2 Inlining Control Selectors

The special treatment of the control selectors is motivated by two considerations. Firstly, the 'pure' implementation of the looping constructs ('whileTrue' and related selectors) uses recursion as a means of iteration. While conceptually very clean, this approach is inefficient due to both the unnecessary overhead of a message send (of the control selector itself) on each iteration of the loop, and the unnecessary use of stack space which increases linearly with the number of iterations performed. This last point is especially important in the parts of the system (such as the ControlManager) which rely on infinitely repeating loops.

The second consideration is the frequency with which the conditional constructs ('ifTrue:' and similar selectors) are used. Not only is conditional execution common in its own right, but it is also used to terminate the recursion of the looping constructs. While the latter point may seem to be obviated by the macro-expansion of loops, this is not always the case since control selectors (both looping and conditionals) can only be inlined safely when their arguments are *literal* blocks; the use of control constructs with variable arguments will thwart the macro-expansion mechanism, causing loops to be executed recursively and conditionals to execute methods in True or False.[33]

The code fragments produced for all the control selectors are obvious, but are included here for completeness. Sending the 'whileTrue' message to a block causes that block to be executed until its value is 'false':

[33]In a standard image (with none of the macro selectors disabled in the compiler) this recursion will *never* happen, regardless of the nature of the arguments. This is achieved by resisting any temptation to pass block variables directly as arguments to a control selector. Instead the block is explicitly sent a 'value' message from inside a *literal* block argument, triggering the macro expansion mechanism. Inspecting a standard image for all senders of 'whileTrue:' will convince even the most skeptical reader of this fact.

```
start:  code for body of literal block
        cmpl   #TRUE, d0
        jeq    start       | repeat while result 'true'
        cmpl   #FALSE, d0
        jne    NONBOOL     | non-boolean receiver!
        moveq  #NIL, d0    | result is always nil
```

Similarly for 'whileTrue:', which executes its block argument until its receiver evaluates to 'false':

```
start:  code for body of receiver block
        cmpl   #TRUE, d0
        jne    done
        cmpl   #FALSE, d0
        jne    NONBOOL
        code for body of argument block
        jra    start
done:   moveq  #NIL, d0    | result is always nil
```

The code for 'whileFalse' and 'whileFalse:' is similar, with the TRUE changed to FALSE in the comparison.

Sending the 'ifTrue:' message to a Boolean causes the block argument to execute if the receiver is 'true', returning the value of the block as the result. Otherwise the block is not executed and the result is 'nil':

```
        generate receiver in d0
        cmpl   #TRUE, d0
        jne    fail
        cmpl   #FALSE, d0
        jne    NONBOOL
        code for body of argument
        jra    done
fail:   moveq  #NIL, d0
done:
```

The code for 'ifTrue:ifFalse:' is similar, with the loading of 'nil' into d0 replaced by the code to execute the second argument:

```
        generate receiver in d0
        cmpl  #TRUE, d0
        jne   fail
        cmpl  #FALSE, d0
        jne   NONBOOL
        code for body of first argument
        jra   done
fail:   code for body of second argument
done:
```

Again, for the inverted cases ('ifFalse:' and 'ifFalse:ifTrue:'), the comparisons are made against FALSE instead.

### 5.5.3 Peephole Optimizations

Once code has been generated for the entire method (including any appended block bodies), a pass is made over the code to remove several unpleasant relics of the rather naïve convention of leaving the results of all expressions (no matter how trivial) in a standard place. There are two such cases to be dealt with: move chains occur where a value (a variable, for example) is moved into the 'result' register `d0`, only to be moved immediately to another location (perhaps during an assignment). The raw generated code fragment is

```
movl fromLocation, d0
movl d0, toLocation
```

The second case is similar, but arises when the second of the instructions is a 'push'. This case is common (for example) when pushing simple values (literals or variables) onto the stack as arguments for a message send. The raw generated code is

```
movl fromLocation, d0
movl d0, sp@-
```

Since the second situation is just a special case of the first, both cases can be dealt with simultaneously by transforming the general case

```
movl fromLocation, d0
movl d0, toLocation
```

into the equivalent

```
movl fromLocation, toLocation
```

There is one more frequent inefficiency which arises, most often in unary message sends within the code that sets up the receiver in both the topmost stack location and the register `d0`. Idle moves (moves that overwrite a value with itself)

```
movl aLocation, aLocation
```

are removed entirely from the generated code.

These three cases account for all of the inefficiencies in the generated code that can be dealt with easily by peephole optimization. More global 'long-range' optimizations are possible to improve the code further, but these were felt to be outside the scope of a code generator that was purposefully kept as simple as (reasonably) possible.

## 5.6 Summary

Smalltalk-80 is a fairly simple language to compile. Parse trees consist of only a (relatively) few types of node, and a naïve code generator can be constructed with little difficulty.

Due to the message passing semantics of the Smalltalk-80 language, many of the standard parse tree optimization tricks cannot be applied. For example, it is almost impossible to apply common subexpression elimination in Smalltalk-80 due to the possibility that the types (classes) of the variables in each of the subexpressions may change unpredictably.[34]

This places much of the responsibility for producing efficient code on the code generator, which cannot rely on the usual high-level optimizations for help. The naïve code generator described in this chapter pays little attention to this requirement beyond trying to cache a couple of useful pointers in registers when possible. Even this is of limited use due to the dynamically allocated object memory used by Smalltalk-80; the code generator must make the pessimistic assumption that the address of every object could change each time a new object is allocated or a message sent.

The naïve generator of this chapter also makes use of a peephole optimization pass to remove the usual types of redundancies found in code generated by simple one- or two-pass recursive-descent compilers. This optimization accounts for a surprising proportion of the average compilation time for a method (section 7.3), and while improving the quality of the code substantially it still leaves the code far from that which should be achievable.

[34]Some work has been done on producing type inference systems for Smalltalk-80 [JGZ88] [ST84] and Smalltalk-like languages [CU89] [CUL89] that allow this kind of optimization to be used, but a practical and mature system has yet to emerge. Type inferencing is beyond the scope of this thesis and will not be considered further.

# Chapter 6

# Delayed Code Generation

> I have a great subject to write upon, but feel keenly my literary incapacity to make it easily intelligible without sacrificing accuracy and thoroughness.
>
> Sir Francis Galton.

The previous chapter described a compiler and runtime environment for a Smalltalk implementation in which methods are compiled into 68020 machine code for direct execution on stock hardware, rather than into bytecodes for interpretation by a virtual machine. Two purposes were served in the process: a runtime environment was described which will continue to be used (unmodified) in this chapter, and a simple compiler and code generator were described, illustrating (rather naïvely) the compilation of Smalltalk for that environment.

In this chapter we will develop a more sophisticated 'delayed' code generator for Smalltalk. Smalltalk is an ideal vehicle for the introduction of delayed code generation since it is such a small language, making it feasible to describe the workings of an entire code generator in the space available, yet the compiled code produced by a naïve approach exhibits many inefficiencies that are common to many languages.

This chapter begins with a review of the flaws inherent in the raw code produced by the naïve code generator, followed by a short analysis of the *primum mobile* of these deficiencies in order to throw some light on possible methods with which to counter them. The strengths and weaknesses of these solutions are considered to identify the most promising, which will be developed into a code generator in the major central portion of the chapter.

This code generator will present an interface to the parse tree nodes which is very similar to that described in the previous chapter, the major difference being that an 'emit' message sent to the machine object will return a value which affects the future course of code generation. Although the parse tree nodes will still determine the order and behavior of the operations performed by the generated code, by sending the appropriate 'emit' messages to a machine object in the necessary sequence, they will never directly manipulate or have any knowledge of the content or structure of these

values.[1] The nodes themselves need only have contact with them in a capacity as 'value holders' on behalf of the code generator, which alone will be responsible for interpreting and manipulating them. The 'generate' messages in the parse tree nodes will need slight modification before they can be used with a delayed code generator, but the changes are in the fine details rather than in the overall logical structure. This organization provides a very clean split between the front-end (in effect, the parse tree itself) and the back-end (the machine object) of the compiler, allowing back ends for various architectures to be plugged into the same front end with immense ease.[2]

The main part of this chapter begins with a discussion of these mysterious values, and what they represent, in the context of code generation for Smalltalk-80 targeted to the M68020. Following the format of the previous chapter, the generation of code for values represented by leaf nodes will be explained before moving on to operations performed on these values which are the province of the interior nodes of the tree. The last of the sections dealing exclusively with Smalltalk explains the generation of code for the optimized messages: the arithmetic and other special selectors, and the control constructs.

After a thorough treatment of delayed code generation for Smalltalk, the scope of the discussion will be expanded for a brief introduction to the use of the technique in the compilation of C. C will provide a vehicle for the discussion of delayed code generation in the context of a much wider class of languages, permitting a discussion of its relationship to language features and resource management tasks that are not inherent in the compilation of a language as simple as Smalltalk.

## 6.1 Shortcomings of the Naïve Code Generator

The 'naïve' code generator of the previous chapter performs poorly in many circumstances for a variety of reasons. Peephole optimization is necessary to eliminate inefficiencies such as move chains and redundant moves, during an assignment for example:

a ← 42

```
movl  #42, d0          | generated for '42'
movl  d0, frame@(-16)  | generated for 'a ←'
```

The generated code also contains many other inefficiencies (due to the committal of values to physical locations, usually a register, at the time of parsing the values) which are not realistically removable by peephole optimization. Not only is the code still far from ideal, but the process of peephole optimization itself slows the compiler considerably.[3]

---

[1] This kind of value is often referred to as a 'cookie'.

[2] To generate code for different architectures from the same parse tree, all that needs to be changed is the (machine object) argument to the initial 'generate' message sent to the MethodNode.

[3] On average, 61% of the total compilation time (of which 71% is taken by code generation) is devoted to peephole optimization in the naïve compiler (see chapter 7).

The code fragments for inlined arithmetic and comparison operations can also be improved. These take their inputs from the top of stack and a register, leaving the result in the register. Even if one operand is a SmallInteger literal, a test is still generated on its class since information regarding the class of the value will be lost before code for the operation itself is generated.

Even where the value of a comparison is not used as a real object (in the case of a loop iteration test, for example) a proper object is generated, tested against 'true' or 'false', and then immediately discarded. It is not possible to plant a 'compare and branch' sequence since the code generator has no knowledge of the final use for a value (even a logical one), so must plant code to construct a real object regardless of its use:

(a == b) ifTrue: [doSomething]

```
       movl  frame@(-8), d0    | 'a'
       movl  d0, sp@-
       movl  frame@(-12), d0   | 'b'
       cmpl  d0, sp@+          | '==' (inlined)
       seq   d0                | [-1B 0B] if [== ~=]
       andw  #8, d0            | [8W 0W] if [== ~=]
       addw  #TRUE, d0         | [16W 8W] if [== ~=]
       extl  d0                | [TRUE FALSE] if [== ~=]
       end of comparison, start of 'ifTrue:'
       cmpl  #FALSE, d0
       jeq   FAIL
       cmpl  #TRUE, d0
       jne   NONBOOL
       doSomething
       jra   CONT
FAIL:  movl  #NIL, d0
CONT:  end of 'ifTrue:'
```

Even after peephole optimization, assuming that the value of the 'ifTrue:' is unused, and that the contents of `d0` are not required beyond the test, and that the peephole optimizer is capable of dealing with this situation, the best that a naïve code generator could produce is:

```
        movl  frame@(-8), sp@-
        movl  frame@(-12), d0
        cmpl  d0, sp@+
        seq   d0
        andw  #8, d0
        addw  #TRUE, d0
        extl  d0
        cmpl  #TRUE, d0
        jeq   FAIL
        cmpl  #FALSE, d0
        jne   NONBOOL
        doSomething
        jra   CONT
FAIL:   movl  #NIL, d0
CONT:   end of 'ifTrue:'
```

Over half of this code is redundant! If it were possible to generate a 'compare and branch' sequence, then the eight instructions between the '`seq`' and the '`jne NONBOOL`' could be replaced by a single '`jne FAIL`' instruction. The resulting code is

```
        movl  frame@(-8), sp@-
        movl  frame@(-12), d0
        cmpl  d0, sp@+
        jne   FAIL
        doSomething
        jra   CONT
FAIL:   movl  #NIL, d0
CONT:   end of 'ifTrue:'
```

which is only one instruction away from being acceptable (the three instruction compare sequence could be reduced to two).

The delayed code generator described in this chapter addresses and solves these, and similar, problems using a general technique that improves all generated code, rather than attempting to special-case a handful of situations in a peephole optimizer; in fact, the need for peephole optimization on the generated code is removed entirely. Code fragments can also be generated to take advantage of the final use of a value. For example, a comparison will only create a reference to a real object if the actual physical value of the comparison is needed later. Inlined operations benefit greatly from the technique described here since information regarding the class of a value is preserved for use in the generation of any code that refers to it (this has an enormous impact on the efficiency of the special arithmetic selectors). It must be stressed that the obviation of peephole optimization is by no means the only important benefit of the technique.

## 6.2 Classifying the Problem

The problems described above can be viewed in several different ways; each way of viewing the problem suggests a different solution.

In many of the cases described, the generated code was less than ideal because the future uses of a value were not known at the time the node representing the value was visited during code generation. Since code generation proceeds as a side-effect of visiting the node, and must be completed before leaving the node, the most general case has to be accommodated every time. This means that even for literals we must generate a move instruction to place them in the 'expression result' register.

The solution in this case is to pass information from a particular node down the parse tree to inform its children of the intended future use of their values. This information could be passed either by decorating the tree with inherited attributes, or in a more object-oriented style by generating code in the children by sending them different messages depending on the desired use of the generated values. This is not an ideal solution for several reasons. In the case of literals which could be used as immediate operands, it would still be necessary to ask a child to place the value of the literal in some particular physical location, such as a register. Also, a child might have no choice but to generate code to place a value in a location which differs from the one requested, and in doing so cause extra instructions to be generated to move the value into the requested location even though the original location for that value may have been just as useful a place to leave it.

An alternative view of the problem is that the code associated with a value is generated before the 'optimal time', thereby committing that value to a register or stack location where it may never be required to reside. The 'optimal time' for committing a value to a location will be the first use of that value, when information regarding the manner of its use is available. In contrast to the approach above, information regarding the values associated with child nodes is passed up the parse tree in the form of synthesized attributes describing the location (if any) and other attributes of the values. Nodes higher up in the tree receive this information and can modify the instructions they generate accordingly. For example, by deferring the code to move a literal value into the expression result register, it may be possible to use the value of the literal directly as an immediate operand in a generated instruction without ever moving it into a register or onto the stack. Also, since most machine-level instructions can accommodate a variety of operand locations, the children can be left to produce their results in the most 'natural' place. In some situations there are artificial constraints placed on the range of locations in which a value can be generated.[4] These valid locations will more than likely be suitable operand locations for operations generated by nodes higher up in the parse tree, so the choice of which location to use should be left to the children rather than the parents.

[4]When performing a division on the PDP-11 the quotient can only be generated in a register with an even number (`r0`, `r2`, `r4` or `r6`). Non-orthogonalities in the instruction set of the MC68020 also constrain some operands to be in registers rather than in a memory location.

The code generator developed in this chapter follows the second of these alternatives, and this technique alone should be sufficient for many languages. In the case of Smalltalk, however, where logical values (the results of relational expressions) may be used either as values in their own right or merely as a condition for a test in a control constructs, it is necessary to use a small amount of inherited information regarding the future use of a logical value.[5] If recovery from inlined relational operations on non-SmallIntegers (by performing a full message send for non-numeric Magnitudes) were not necessary, the use of synthesized attributes alone would suffice.[6]

## 6.3 Operand Descriptors

The rather poor quality of the code generated using the straightforward techniques of the previous chapter necessitated peephole optimization to remove inefficiencies such as move chains and dead code. These deficiencies were introduced because the code generator had no choice but to generate code to place the 'result' from each node into a known physical location before code generation could continue in that node's parent.[7] In the case of the naïve code generator described in the previous chapter, this known location was the register `d0`.

The time at which code is generated in a single-pass compiler is usually the time at which a node is visited in a parse-tree walk, or the time at which a procedure associated with the parsing of a particular terminal or non-terminal symbol is executed in single pass recursive descent compiler. What is required is a technique for delaying the generation of code associated with fetching a value until that value is actually used in some way, at which time we can match the location of the value to an operand addressing mode. This mechanism is implemented by passing *operand descriptors* up the tree from leaf nodes, each operand descriptor representing a particular value or location for an operand. This can be thought of as partial code generation, where the code associated with a single operand (rather than a whole instruction) is generated as an operand descriptor, and passed up the tree as a synthesized attribute to a parent node which will use the operand when generating code for an entire instruction.

So, an operand descriptor is a compile-time representation for some particular legal machine operand. These operands may be addressable quantities such as memory or register locations, literal quantities such as integers, or even abstract quantities such as the representation for truth values within the processor's condition codes (flags, or

[5]The ParcPlace compilers make a similar distinction between expressions whose value will be required later and statements whose value is not required; parse nodes are sent either 'emitForValue' or 'emitForEffect' accordingly. This is normally only used to generate bytecodes that implicitly 'pop' the stack after performing some operation such as a message send.

[6]To compile languages such as C as efficiently as is possible with a single-pass recursive descent compiler or tree-walking code generator, inherited information is not needed at all, as will be seen in section 6.6.

[7]This is true for both interior and exterior (leaf) nodes of the tree, although it is mostly the code generated at leaf nodes which is the cause of move chains and other artifacts which require removal by peephole optimization.

status) register.[8] Typically, operand descriptors will be created at leaf nodes and passed up the parse tree (whether it be explicit or implicit) to interior nodes that will use them as operands; these nodes in turn will pass an operand descriptor up the parse tree as their result. The operand descriptors returned from interior nodes will be either one of their original 'synthesized' operand descriptors returned by a child (for example the right- or left-hand operand of an assignment), a newly created operand descriptor (maybe one representing the register destination of an operation), or (as we shall see later) an operand from a child node modified in some suitable way. In the simple case of exterior nodes, the operand descriptors returned as a result of code generation in these nodes are simply representations of the operands that would have been moved into `d0` as the last step in code generation at those nodes in the naïve code generator.

In some circumstances it may be necessary to decide between two possible operand descriptors which represent the same value. Such 'aliasing' situations can occur, for example, after an assignment when both the left- and right-hand operand descriptors represent the same value. Since the operand descriptor representing the result of this assignment may be used in the generation of another instruction later on, it makes sense to ensure that the one chosen to be returned to the parent node corresponds to the operand with the lowest overhead when used in an instruction.[9] A 'cost' is therefore associated with each particular type of operand descriptor. In situations where a choice must be made between several aliased operand descriptors, the one with the lowest cost wins. For example, the cost of using a literal or register is usually lower than the cost of using a memory location, and the cost of using an absolute location is lower than using one with an indexed or indirect addressing mode. A 'league table' of operand costs is easy to construct based on information supplied by manufacturers for their microprocessors.

### 6.3.1 Representation of Operand Descriptors

The naïve code generator of the previous chapter made use of two hierarchies, classes from one representing the instructions available in the target architecture and the other representing operands within those instructions. Since operand descriptors represent legal machine operands, the classes in the hierarchy rooted at M68Operand will serve perfectly well as operand descriptors for the purposes of code generation in a Smalltalk compiler. A corollary to this is that operand descriptors can be used directly to represent the operands of instructions in the internal representation of the generated code.

Since operand descriptors always represent a legal machine operand, they will be

[8] Some languages, C in particular, associate logical meaning to physical values and vice versa. Rather than complicating the situation, this dual interpretation of each and every value simplifies delayed code generation, as will be discussed in section 6.6.3.

[9] In both Smalltalk and C, statements are merely expressions whose result is discarded. An assignment can therefore form part of larger enclosing expression, and so must have an overall value. In both of these languages the value of an assignment is the value of either the right- or left-hand sides *after* the assignment has taken place.

written simply as the operand itself enclosed in square brackets. A valid operand descriptor could thus be written:

```
[a1@(16,d1:l)]
```

## 6.4 Code Generation with Operand Descriptors

In the development of the delayed code generator, we will proceed in a manner similar to that adopted for the naïve code generator of the previous chapter, by considering the abstractions provided by the machine object for the parse nodes for each of the operations supported in Smalltalk. Code generation at leaf nodes, which will be presented first, is trivial since these nodes have no descendents and are consequently only sources of operand descriptors; it is not until interior nodes are considered, which need to manipulate 'synthesized' operand descriptors passed up from their descendants, that the technique becomes interesting.

Interior nodes are associated with assignment, message sending, BlockContext creation, and returns from either methods or blocks. These will be dealt with in that order, ignoring inlined and macro-expanded selectors. The last section that deals exclusively with Smalltalk will discuss code generation for inlined special selectors and control constructs, for which very efficient code can be produced with relatively little effort.

### 6.4.1 Operand Descriptors generated at Leaf Nodes

We defer the generation of code for operands wherever possible by encoding values as operand descriptors, which represent either the value itself (in the case of literals) or the physical location (register number, memory address, etc.) in which the value can be found. Following this convention, figure 6.1 shows the output for leaf nodes from the naïve code generator of the previous chapter and from the delayed code generator. Each entry in this table is explained in a little more detail below.

At a leaf node representing a literal, code is generated by sending the machine object an 'emitLiteral:' message with the literal itself as the argument. In the naïve code generator, code was generated on the instruction stream to load the literal into the expression result register. Rather than appending an entire instruction to the code stream, the delayed code generator returns the appropriate operand descriptor to the parse tree node which sent the 'emitLiteral' message:

```
M68000>>emitLiteral: aLiteral
    ↑M68LiteralOop value: aLiteral
```

The generated code will be an empty sequence of instructions returning an operand descriptor that encodes the value of the literal itself:

[*#literal*]

| Node | Code Generator Naïve | Code Generator Delayed |
|---|---|---|
| Literal | `movl` *#literal*`, d0` | [*#literal*] |
| Argument | `movl frame@(8+4N), d0` | `[frame@(8+4N)]` |
| Temporary | `movl frame@(-4-4N), d0` | `[frame@(-4-4N)]` |
| Instance | `movl frame@(16), d1`<br>`movl obtab@(d1:l), a1`<br>`movl obmem@(8,a1:l), d0` | `movl frame@(16), d1`<br>`movl obtab@(d1:l), a1`<br>`[obmem@(8,a1:l)]` |
| Global | `movl obtab@(O(1622)), a0`<br>`movl obmem@(12,a0:l), d0` | `movl obtab@(O(1622)), a0`<br>`[obmem@(12,a0:l)]` |

**Figure 6.1:** Result of generating code at leaf nodes for both the naïve code generator of the previous chapter, and the delayed code generator of this chapter.

Similarly, at a leaf node representing a temporary (or argument) variable it is not necessary to generate code to load the value of the variable into the result register, instead an operand descriptor is created representing the location of the variable and returned to the parent node:

```
M68000>>emitTemporary: index
    ↑M68IndirectValue
        base: (mapper inBlock ifTrue: [HomePointer] ifFalse: [FramePointer])
        offset: index + 1 * -4
```

The argument to the 'emitTemporary:' message is the index of the variable, which will be, for all intents and purposes, independent of the architecture for which code is being generated.

A typical descriptor representing, for example, the third temporary variable would be:

```
[frame@(-16)]
```

This pair of examples, for generating operand descriptors for literals and temporary variables, illustrate the relationship between the parse tree nodes and the machine object. The parse nodes use the semantic information in the parse tree to decide which operations to request from the code generator, in the form of 'emit' messages sent to the machine object with appropriate arguments taken from the state held within the nodes

themselves. The code generator in turn translates these 'emit' requests into machine-dependent operand descriptors, extracting the necessary information from the 'emit' arguments.[10] Again, 'self' and 'super' are treated as any other argument variable (as in section 5.4.3). The definitions of the 'emit' messages called from the other types of leaf node will not be shown, since they follow a very similar pattern to those above.

Operand descriptors for instance variables are generated by sending the machine object an 'emitInstVar:' message, with the index of the instance variable as the argument. Instance variables are a little more complicated than temporaries since they require the object memory address of the receiver to be in a suitable base register. The code to initialize this register is generated (if necessary — see below) before returning an operand descriptor representing the physical location of the variable as an offset into the receiver:

```
movl  frame@(16), d1     | receiver
movl  obtab@(d1:l), a1   | start of receiver in memory
[obmem@(8,a1:l)]         | first inst. var, offset 8
```

After the first reference to an instance variable, the base register `a1` can be treated as a cached pointer to the start of the receiver in memory, provided the receiver does not move due to garbage collection. (The machine object itself is responsible for maintaining a flag reflecting the state of this cache.) This is easy enough to arrange, since the code generator knows precisely at which points in the generated code potential garbage collection can occur.

The garbage collector only ever runs during an attempt to allocate an object. The object allocator is called in three situations: when a BlockContext or Point is created in line, or when an object is created as the result of a 'new' message. Invalidating the cached copy of the receiver in the former case is trivial. The latter is also trivial in the absence of any 'inter-procedural' optimizations since any message send potentially involves the creation of a BlockContext or a new object, so the cache should be invalidated on every message send. Once invalidated, the receiver base register must be regenerated during the next access to an instance variable.[11]

An example, although prematurely demonstrating the code produced by the delayed code generator, will make this clearer:

instVarA ← instVarB size

[10]Note that this mechanism also removes the need for special treatment of the left-hand side of an assignment, since the result of generating code for a variable is not to load that variable's value, but to create and return an operand descriptor representing the variable's location.

[11]This was the meaning of the 'cached base of 'self'' in figure 5.2 of section 5.1.2.

```
movl   frame@(12), d1          | oop of receiver
movl   obtab@(d1:l), a1        | pointer to receiver
movl   obmem@(12,a1:l), sp@-   | 'instVarB'
movl   sp@, d0                 | receiver
movl   #O(91), d1              | 'size'
jbsr   Send                    | base in a1 invalidated
addql  #4, sp
movl   frame@(12), d1          | regenerate receiver
movl   obtab@(d1:l), a1        | pointer in a1
movl   d0, obmem@(8,a1:l)      | instVarA ←
[d0]                           | result of assignment
```

The cached base of 'self' in `a1` may be destroyed during the message send, so it must be regenerated during the second instance variable access (to 'instVarA').

Global variables are similar to instance variables, being represented as an offset from a pointer into the object memory. However, there is no advantage in caching this pointer so it might as well be generated by the leaf node itself, which subsequently returns an operand descriptor based on the pointer:[12]

```
movl  obtab@(O(associationOop)), a0
[obmem@(12,a0:l)]                      | value field
```

### 6.4.2 Assignment

The simplest operation in Smalltalk is assignment, so we will use this as the first example of the generation of code for an actual operation using operand descriptors.

Consider the statement 'tempN ← 42'. The parse tree consists of an AssignmentNode with two children: a TemporaryNode for the left- and a LiteralNode for the right-hand sides of the assignment.

As was explained earlier, an operand descriptor represents either a literal value or the physical location in which to find the value it represents. Thus for assignment it is simply necessary to generate operand descriptors for the right and left hand sides of the assignment (the source and destination values, respectively) and then emit a '`move`' instruction to copy the value represented by the source operand descriptor into the location represented by the destination operand descriptor.

Upon receipt of a 'generate' message, an AssignmentNode will send another 'generate' to the right hand side which will (no matter how complex the expression involved) return an operand descriptor representing the result (or more precisely, the location of the result); the node stores this operand descriptor (implicitly, on the stack, in this particular case) for later use. The left hand side is then also sent a 'generate' message which returns an operand descriptor representing the location to be assigned to. The

[12]In a typical Smalltalk image only 0.6% of global variable accesses would benefit from a cached pointer to the association, compared with the 25.7% of instance variable accesses which benefit from the technique. Pointers to globals are therefore not cached in the same manner as pointers to the receiver since doing so would have an essentially undetectable effect on performance. With one global reference every two methods on average, the space that would be saved by caching pointers is also negligible.

only remaining thing to do is to ask the machine object to emit a 'move' instruction to complete the assignment. Again, the definition of the 'generate' message will be shown for this operation (to illustrate the technique as fully as possible), but not for later operations for which code is generated in a similar manner.

```
AssignmentNode>>generate: aMachine
    | source destination |
    ↑aMachine
        emitMove: (rhs generate: aMachine)
        to: (lhs generate: aMachine)
```

It would appear that the aliasing situation which occurs as the result of any assignment is ignored here, but this is not the case since the 'emitMove' message must return one of its arguments as its overall result, so the choice of which argument (source or destination) to return to the parent is made by the machine while completing the 'emitMove':

```
M68000>>emitMove: src to: dst
    src = dst
        ifFalse:
            [codeStream nextPut: M68move source: src dest: dst].
    ↑src cost < dst cost ifTrue: [src] ifFalse: [dst]
```

which will append a move instruction to the code stream:

```
move  #I(42), frame@(-4-4N)
```

The last statement in the 'emitMove' method chooses which of the two possible operand descriptors to return as the result of generating code for the assignment. The choice is based on the 'cost' associated with each type of operand descriptor. In this case the operand descriptor for the assignment as a whole should be [#42] since an immediate quantity involves less overhead when used as an operand compared to a frame-pointer relative memory location.

Figure 6.2 shows the parse tree, operand descriptors and generated code for this example.

Comparing the code produced by the delayed code generator, even for this trivial example, to that which the naïve compiler would have produced, it should be apparent that the move chain which would have required removal by peephole optimization is no longer present in the final code.[13] The leaf node no longer needs to generate an entire instruction, and can generate just the required operand and pass that up the parse tree to be used later by the assignment node which has much more information available regarding what to do with it.

[13]The more alert reader will probably have been slightly suspicious at the 'src = dst' test in the 'emitMove' method, which is the point in the delayed code generator at which a 'virtual' peephole optimization is performed, removing the possibility of a redundant move due to the assignment of a value to itself.

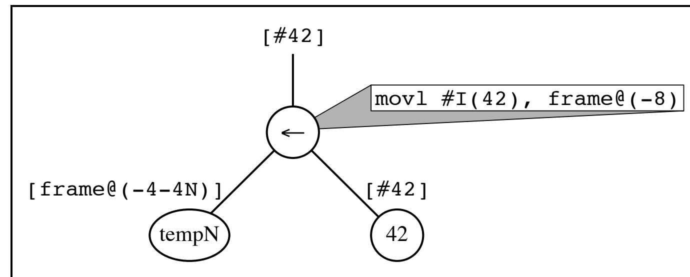


**Figure 6.2:** Parse tree, operand descriptors, and generated code for the assignment ‘tempN ← 42’

### 6.4.3 Message Sends

The code produced for normal dynamically-bound message sends is comparable with that produced by the naïve compiler of the previous chapter. The most significant benefits are only seen when the inlining of the special selectors is considered. This will be discussed in a short while.

The parse tree and code generator enjoy a very close, almost synergistic, relationship. The code generator has intimate knowledge of the capabilities of the target architecture and can therefore choose the best implementation strategy for any particular operation requested of it by a parse tree node via an ‘emit’ message. Conversely the parse tree has an enormous amount of semantic knowledge regarding the values used in the program, both the ways in which they are to be used and in the flow of data through the program. It is the successful, and intimate, marriage of these two very rich sources of information that gives the delayed code generator the ability to generate surprisingly efficient code with relatively little effort.

Message sending is a prime example of this relationship. Message nodes effect message sends by sending the machine object the appropriate ‘emit’ message: ‘emitUnary’, ‘emitBinary’ or ‘emitKeyword’.[14] The arguments to these ‘emit’ messages are the nodes themselves; just as the code generator entrusts the parse nodes with operand descriptors so that they can draw on their semantic knowledge to use the operand descriptors in the most beneficial manner, so the parse nodes sometimes entrust themselves to the code generator so that it can decide the best time to forward the ‘generate’ messages to any descendants that those nodes might have. In the case of message sends, those descendants are the message’s arguments which will receive ‘generate’ messages from the code generator rather than from their parent node directly.

The principles for code generation for message sends are the same in each of the three cases, so we will concentrate on keyword messages (an arbitrary choice). The KeywordNode responds to the ‘generate’ message by sending the machine object an

[14]Splitting the generation of code for message sends into three operations, depending on the type of message being sent, is merely a convenience to avoid some rather unpleasant ‘procedural-style’ case analysis in the code generator.

'emitKeyword:' message with itself (the KeywordNode) as an argument. As before, the 'generate' method is almost trivial:

```
KeywordNode>>generate: machine
    ↑machine emitKeyword: self
```

Dynamically-bound message sends are still constrained by the runtime conventions described in section 5.1.3. Arguments must still be placed on the stack, and the result of a message send is still expected in the usual `d0`. Assuming the message selector and arguments do not make the send suitable for optimization (the message is not, for example, a control selector with block arguments), the code generator responds to the 'emit' message by sending a 'generate' message to the receiver and arguments in turn, pushing each result on the stack in preparation for the dynamic bind. The send is completed in exactly the same manner as was described in the previous chapter (section 5.4.5):[15]

```
M68000>>emitKeyword: message
    self push: (message receiver generate: self).
    message arguments do: [:arg | self push: (arg generate: self)].
    ↑self
        emitDynamicSend: message selector
        size: 1 + message arguments size
        supered: message receiver isSuper
```

Code for cascaded message sends is implemented in the same manner as for the naïve compiler. A CascadeNode contains the initial receiver, and an OrderedCollection of message nodes forming the cascade. The receivers in these message nodes are ignored for the purposes of code generation.

A CascadeNode responds to the 'generate' message by asking the machine object to 'emitCascade:' with the CascadeNode itself as the argument. The machine generates code for the receiver, and then propagates the 'generate' message to each message of the cascade in turn. The result of the last of these propagated 'generate's is returned from the CascadeNode as its overall result.

### 6.4.4 Method Entry and Exit

The code sequences produced for method entry and exit by the 'emitMethod:selector:' message, and the code for block preludes produced by the 'emitBlockPrelude:' message are identical to that given in section 5.4.6. There really is only one way to perform these operations, which have no intrinsic result and are therefore not affected by the

[15]The 'push: something' message is defined for convenience to append a '`move` *something*`, sp@-`' onto the code stream. The 'emitDynamicSend:size:supered:' message loads `d0` with the receiver (from further up the stack using the 'size' argument) and `d1` with the selector, then calls `Send` or `Super` as appropriate and cleans up the stack, as explained in section 5.4.5.

use of operand descriptors or delayed code generation. The associated 'emit' messages therefore do not return an operand descriptor as their result.

The 'emitLocalReturn:' method now takes a single argument representing the value to be returned. Before generating identical code to that given in section 5.4.6, a 'move' instruction is generated (if necessary) to place the result in d0.

### 6.4.5 Primitives and Blocks

Since the runtime support is unchanged, the interface to the primitives is the same for code generated by both the naïve and delayed code generators. Primitives are explained fully in section 5.4.7.

The code generated for BlockContext creation in response to an 'emitBlock' message is also as given in section 5.4.8. In the case of blocks, however, their treatment in conjunction with the inlining of the control constructs delivers some impressive results. This rather more involved area is the subject of the next section.

Both primitives and blocks have an associated return value. The associated 'emit' methods are altered slightly to return an operand descriptor representing the location of any returned value that construct might have. For example, the 'emitBlock' message returns the descriptor [d0], since the OOP of the newly created BlockContext is left in this location by the generated code fragment.

## 6.5 Optimizations

The generation of code for the control constructs and special arithmetic selectors is where the delayed code generator begins to offer some tangible improvements in the compiled code. The quality of the final code is much higher than could be realized with any amount of peephole optimization, and is possibly beyond the reach of any optimizations operating solely on the 'raw' generated code of the naïve code generator.

This section deals with the most involved area of the code generator: the mechanisms for inlining the special selectors, and macro-expanding the control constructs. It starts by explaining a mechanism for the treatment of inlined arithmetic selectors, introducing the concept of the deferred message send; the effort of understanding the inlining of arithmetic selectors is worth expending since the explanations of the other inlined operations will be much easier to follow once the arithmetic selectors are understood. Inlined arithmetic selectors have much in common with inlined arithmetic comparisons, so these will be covered next in conjunction with a new type of operand descriptor representing logical (rather than physical) values which are bound to the state in the machine's condition codes register.

Although the use of operand descriptors to delay the generation of code associated with values improves the code produced for assignment, message sending and blocks by eliminating move chains and similar deficiencies that are trivial to remove by peephole optimization, it is not until attention is given to the special selectors (the arithmetic selectors for example) and macro-expanded selectors (the conditional and

looping constructs), that the full benefits of the application of delayed code generation to Smalltalk can be realized.

Most of the power associated with delayed code generation with respect to control constructs comes from the treatment of relational operators and logical values. By introducing the notion of a conditional 2-way branch based on the logical 'truth' of an operand descriptor, and then passing this information *down* the parse tree to inform a child of the intended use of its value (which can be thought of as an inherited attribute for the child), very efficient code can be produced for all of the control constructs.

## 6.5.1 Inlined Special Selectors

The treatment of inlined selectors can be much more effective with delayed code generation. In cases where there are literal arguments to inlined selectors, there is no need to test the class of those arguments in the generated code; if the compile-time check succeeds then there is no need for a check at run-time, and if the compile-time check fails then the inlined send can be abandoned completely and code for a normal dynamically bound send generated. This is especially important in the arithmetic and relational operations involving SmallInteger arguments. The generic cases shown in section 5.5.1 test the class of receiver and argument for the common case, and code to perform the operation was emitted inline whenever possible. If either the receiver or argument failed the class check then a full message send was performed for the operation.

### 6.5.1.1 Arithmetic Operations

In the naïve code generator, by the time the compiler had generated code for the receiver and argument, all that was known about them was their locations. For a unary message the receiver was in `d0`; for binary messages the argument was in `d0` and the receiver on the top of the stack. Code had to be generated to check the class of both the receiver and any argument at runtime before an inlined operation could be performed. In the delayed code generator, operands are represented by operand descriptors which either represent a literal or some machine location in which the value resides. When compiling code for inlined selectors, it is easy to check the class of literal receivers and arguments at compile time, obviating the need for the check at run time. For example, the inlined version of the addition operation generates operand descriptors (and maybe some code) for both receiver and argument, checking for the possibility of the receiver being corrupted by the evaluation of the argument and taking action if necessary, and only bothering to generate code to check the classes of non-literals. If either argument is a non-SmallInteger literal then the inlined send can be abandoned completely and a full send used in place. If either argument is a SmallInteger literal, then there is no need to check its class a run time.

A simple example will make this clear. Consider the statement

```
M68020>>emitInlinedBinary: message
    | rpushed rcv arg fail needsend cont |
    rpushed ← false.
    (rcv ← message receiver generate: self) isInteger
        ifFalse:
            [message argument isSend
                ifTrue:
                    [rpushed ← true.
                    self push: rcv]].
    (arg ← message argument generate: self) isInteger
        ifTrue:
            [rcv isInteger
                ifTrue:
                    [↑M68Literal value:
                        (rcv value
                            perform: message selector
                            with: arg value)]]
        ifFalse: [arg ← self inD1: arg].
    rcv isInteger ifFalse: [rpushed
                ifTrue: [rcv ← self pop: d0]
                ifFalse: [rcv ← self inD0: rcv]].
    self
        emitBranchNotInteger: rcv and: arg to: (fail ← self newLabel);
        emitInlinedOp: message selector rcv: rcv arg: arg;
        emitLabel: (cont ← self newLabel);
        defer: message rcv: rcv arg: arg start: fail resume: cont.
    selfCached ← false.
    ↑d0
```

**Figure 6.3:** The full definition of ‘emitInlinedBinary:’. See the text for a detailed description of its operation.

---

```
| a b |
⋮
a ← a + 1
```

The BinaryNode will send an ‘emitBinary:’ message to the code generator which checks if the selector is in the set of selectors suitable for inlining, and that neither receiver nor argument is a non-SmallInteger literal. If these conditions are not met, a normal dynamic bind is performed.[16] Otherwise the code generator emits code for the inlined

[16]If either the receiver or argument is a non-SmallInteger literal then the class check would fail on every occasion, so the inlining can be abandoned even before it is begun.

version by sending itself the 'emitInlinedBinary:' message. This method is long and complicated. The full definition is given in figure 6.3, and explained in detail below.

Once the code generator decides to go ahead and inline an arithmetic message, the situation becomes quite complicated. The receiver is sent a 'generate' message, which returns an operand descriptor representing its location or value. If the receiver is not a simple literal, and the argument is a kind of message send, then the receiver must be considered volatile; it may depend on machine a resource (such as a base register) that could be corrupted during the evaluation of the argument. In such cases the receiver must be pushed onto the stack to protect it while the argument is evaluated.

The argument can now be sent a 'generate' message, and if it does not represent a literal it is moved into `d1`. If both receiver and argument are literals, then the required operation could be performed at compile time and the result returned in an operand descriptor as the result of code generation for the message send. However, this assumes that the compile time and run time meanings of the message are the same, which may not be a reasonable assumption.[17]

The receiver (whether pushed or not) is moved into `d0` which gives us a consistent position whatever the types of the operands: receiver in `d0` and argument in `d1`. The code generator next emits code to perform class checks for the receiver and argument by sending itself 'emitBranchNotInteger:and:to:' with the two operand descriptors and a new unique label as arguments; the class check is omitted for one of these if it happens to be a SmallInteger literal. The label ('fail') is used for the destination of the branch in the class checks if the argument is not a SmallInteger. The code to perform the operation is appended to the code stream (by sending 'self' the 'emitInlinedOp:' message), leaving only the recovery (from class check failure) to be handled.

The 'emitInlinedOp:' method generates the instructions necessary to perform the inlined operation, based on the selector and the operand descriptors for the receiver and argument. It is the last step in the generation of code for the inlined case so the result is left in `d0`, which is necessary since this is where the full send will leave the result if a class check fails. This method is simple, and the details of its operation are not particularly interesting. For completeness, its definition is shown in figure 6.4.

Rather than generate code for the full send along with the code for the inlined version, it will be deferred until the end of the method. This is done for several reasons. Not only does it make the code much easier to read, it removes a branch instruction that would otherwise have been necessary after the inlined operation, to skip over the full send. As we shall see later, the provision of 'deferred sends' for recovery from inappropriate classes reaching inlined operations provides a powerful mechanism when used during the inlining of relational operations for control constructs.

The label 'fail' was already created to tag the first instruction in the full send. We now need to register a 'deferred' send which will be compiled at the end of the method, along with the bodies of blocks. This will perform a full dynamic bind, returning the

[17]If the message is a candidate for inlining, and both receiver and argument are SmallInteger literals, then it *is* a reasonable assumption since the operation will be inlined at runtime anyway. In this case performing the operation at compile time implements constant folding during code generation, and so is indeed done.

```
M68020>>emitInlinedOp: op rcv: rcv arg: arg
        ⋮
    op == #+ ifTrue:
        [self inD0: (rcv isInteger ifTrue: [rcv noTagBit] ifFalse: [rcv]).
        arg isInteger ifTrue:
            [codeStream nextPut: (M68add source: arg noTagBit dest: d0).
            ↑d0].
        codeStream nextPut: (M68add source: arg dest: d0).
        rcv isInteger ifFalse:
            [codeStream nextPut: (M68sub
                        source: (M68Literal value: 1)
                        dest: d0).
            ↑d0].
        ⋮
```

**Figure 6.4:** The part of 'emitInlinedOp' that deals with inlined addition. The receiver is moved to d0 (if it wasn't already there), stripping the tag bit if it was a SmallInteger. If the argument is a SmallInteger then the receiver was a non-literal, so the argument is added to the receiver directly, without the tag bit, the result of the addition being left in d0 with the tag bit (inherited from the receiver) in place. Otherwise an 'add' instruction is generated to perform the addition, and if both receiver and argument were non-literals, the excess due to the addition of the two tag bits is subtracted. Note that overflow handling has again been ignored.

---

result in d0 as usual. Obviously, an extra label will be required to which control will be transferred after this send has completed, to rejoin the method immediately after the inlined operation. This label ('cont') is created and appended to the code stream. Deferred message sends will assume an important rôle in the generation of code for the conditional constructs, as we shall see in section 6.5.2.2

It may seem from the above that efficiency could be improved by not insisting that the result of the inlined operation be returned in d0. However, the result must be left in this location since the operation may require a full send if the argument and receiver are not both SmallIntegers.

The final generated code for inlined operations will be as follows:

```
        arguments in d0 and d1
        btst   #0, d0         | SmallInteger?
        jeq    FULL           | no
        btst   #0, d1         | SmallInteger?
        jeq    FULL           | no
        perform inlined operation, result in d0
CONT:   rest of method...

        deferred sends...
FULL:   movl   d0, sp@-
        movl   #I(1), sp@-
        movl   sp@(4), d0
        movl   #selector, d1
        jbsr   Send           | perform full send
        addql  #8, sp
        jra    CONT           | rejoin method body
```

#### 6.5.1.2 Relational Operations

When supplied with SmallInteger arguments, the relational (comparison) operations can be inlined in much the same manner as the arithmetic operations. Having performed the inlined operation, a small additional code sequence must be generated to convert the result of the operation (which is implicit in the machine's condition codes register) into a proper object, 'true' or 'false', as follows:

```
result of comparison in condition codes
scc   d0          | [-1B 0B] if [true false]
andw  #8, d0      | [8W 0W] if [true false]
addw  #TRUE, d0   | [16W 8W] if [true false]
extl  d0          | [TRUE FALSE] if [true false]
```

This sequence makes use of the fact that the object pointers for 'true' and 'false' are immutable, well-known and consecutive, as follows. The 's*cc*' instruction either sets its (byte-sized) argument to 0 or −1 depending on the condition *cc* specified. If the condition is satisfied then the lowest eight bits of `d0` will be set, otherwise cleared. '`and`'ing this with 8 gives either zero or the difference between consecutive object pointers in `d0`. Adding '`TRUE`' to this leaves the object pointer to either 'true' or 'false' in `d0`, since 'false' immediately follows 'true' in the object table. The final step extends this from a word quantity into a long, massaging the result into a valid object pointer.

Apart from these four extra instructions, only one other minor difference from the inlining of arithmetic operations is needed. When registering the deferred message send, a slightly different version of the 'defer' message is used in order to perform the necessary check for non-Boolean results. However, to avoid overcomplicating the present discussion, the explanation will be left until section 6.5.2.2.

### 6.5.2 Inlined Control Constructs

The messages implementing conditional execution and looping are the only constructs in Smalltalk for which nontrivial alterations to the 'generate' messages in the parse nodes themselves are required. For these, a small amount of inherited information will be passed down the parse tree to guide the generation of code for literals and relational (comparison) operations. This is necessary for the inlining of comparison operations since a distinction must be made between the use of their results as values (to be assigned to a variable or passed as an argument, for example) or as conditions in a control construct (where the value is not explicitly required).

#### 6.5.2.1 Control Flow Forking

A single model will suffice for the generation of code for all the control constructs — that of the two-way fork in the flow of control through the code. Using the concept of a fork, both conditional execution and looping can be modeled (see figure 6.5).

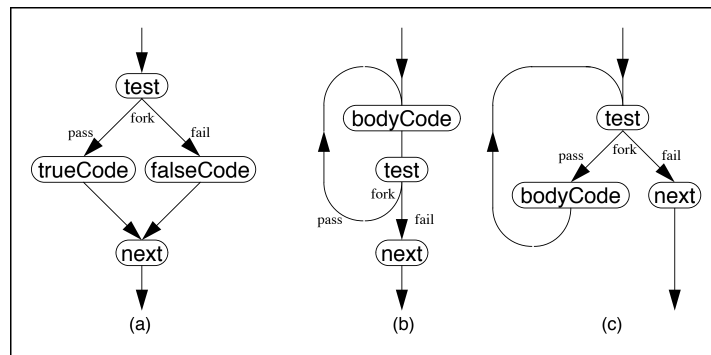


**Figure 6.5:** Implementing conditional execution and looping via a fork in the flow of control. **(a)** Forking to one of two possible code fragments to implement conditional execution. **(b, c)** Forking backwards to a point in the code already executed to implement looping, both 'repeat' and 'while' loops respectively.

---

Forking is implemented by adding two new 'generate' message to parse nodes, 'generateForkTrue: aLabel for: aMachine' and 'generateForkFalse: aLabel for: aMachine'. Most nodes will inherit default definitions from ParseNode, as follows:

```
ParseNode>>generateForkTrue: destination for: machine
    ↑machine
        emitBranchTrue: (self generate: machine)
        to: destination
```

The definition of 'emitBranchTrue: value to: label' is almost obvious, and causes the following code to be placed on the code stream, unless the first argument is an operand descriptor representing a literal:

```
cmpl  #TRUE,value
jeq   label
cmpl  #FALSE,value
jne   NONBOOL
```

In the case where the 'emitBranch:' argument is an operand descriptor representing a literal, the appropriate branch of the fork can be determined at compile-time. If the literal meets the forking condition than an unconditional branch to the specified label is generated. If the literal is non-Boolean then a compile-time 'mustBeBoolean' error is raised due to the illegal receiver.

A similar default definition is also present for 'generateForkFalse:for:', sending 'emitBranchFalse:to:' to the machine instead. These defaults simply propagate a 'generate:' message to the receiver, and then cause the machine to emit code to check the result against 'true' and 'false'. The result is either a branch to the specified label if the 'forking' condition is met, a branch to 'NONBOOL' if the value is non-Boolean, or no branch at all.

These new 'generate' methods are overridden in two classes. For BinaryNodes, which implement relational (comparison) operations, it may be possible to inline the required operation. Rather than sending the usual 'emitBinary:' message, the code generator is sent an 'emitForkTrue:' (or 'emitForkFalse:' as appropriate) with the BinaryNode itself as the first argument. This leaves any decisions about inlining to the code generator, which is only required to implement the desired forking behavior based on the result of the binary operation.

BlockNodes are used extensively as the receivers in 'while' constructs. On receiving a 'generateFork' message they will propagate a normal 'generate:' message to each of their statements except the last, to which they will propagate the 'generateFork' message originally received. There are two points to note here. Firstly that this generates the code for the block's body inline, rather than deferring it for compilation after the body of the method. This is acceptable since blocks will only ever be sent a 'generateFork' message when they are used to control loop iteration. Secondly, and for the same reason, the final statement in the block body will most likely be a relational operation. Passing the original 'generateFork' message on to this statement will cause the comparison and forking operations to be inlined if at all possible, as described in the preceding paragraph.

#### 6.5.2.2 Deferred Message Sends Revisited

Deferred message sends were introduced in section 6.5.1.1 as a mechanism for storing dynamic binds for code generation at a later date. There was no particular reason to introduce this mechanism (apart from the aesthetic consideration mentioned) at the time, but we will now see how this mechanism can help in the recovery from inlined comparison operations used as the condition in some control construct.

When used as the test in a conditional construct, an inlined comparison operation should be able to make use of a 'compare and branch' sequence. It may seem, in the light of possible class check failures, that this is impossible since the result returned

would be a real object in `d0` if the full send were invoked. However, we can extend the deferred send mechanism slightly to take advantage of the situation, and allow the inlined version of any comparison operation to use a compare and branch sequence.

The vital observation is that any conditional construct splits the flow of control between two possible paths, depending on some logical value. If an inlined comparison is possible then this value will be in the machine's condition codes register; if a full send is required then the value will be 'true' or 'false' returned in `d0`. In either case, there is a point in the code corresponding to the test passing, and another corresponding to the test failing. In a compare and branch sequence, the branch will transfer control to one of these points or fall through to the other, which will be immediately after the branch instruction. If we record labels for both of these points while registering the deferred send it will be possible to generate code to test `d0` against 'true' and 'false' (and perform the non-Boolean receiver check) as part of the deferred send, transferring control to the appropriate point in the body of the method.

This is accomplished by expanding the 'resume:' argument of the message deferral mechanism (see figure 6.3) into two arguments, one being the label connected to the code associated with the test passing and the other with the test failing:

```
machine
    defer: message
    rcv: receiver
    arg: argument
    true: passLabel
    false: failLabel
```

From here, it is a trivial step to see how the code generator produces the following code for inlined comparison operations and their recovery from class check failure:

```
        perform test, branch to FULL on class check failure
        otherwise result implicit in condition codes register
        j??   FAIL        | fork
PASS:   success code
        jbra  CONT
FAIL:   failure code
CONT:   rest of method
          ⋮
FULL:   perform full send for recovery, result in d0
        cmpl  #TRUE, d0
        jeq   PASS
        cmpl  #FALSE, d0
        jeq   FAIL
        jbra  NONBOOL
```

The next sections explain how the these forking mechanisms are used during the generation of code for the conditional and looping constructs.

In section 6.5.1.2 it was mentioned that inlined relational operations used a slightly different version of the 'defer' method than that used by inlined arithmetic operations; one which also provided a check for nonBoolean results before continuing the execution of the method after recovery from a class-check failure. The actual version used is that just given, but with both 'true:' and 'false:' arguments being the same 'resume' label.

#### 6.5.2.3 Code for Conditionals

As with previous examples, we will concentrate on the optimization of one particular message from the class of optimizations under consideration. In this case, the inlining of the message

```
a == 1 ifTrue: [a ← 0]
```

will be explained.

A 'generate' message arriving at a KeywordNode causes the code generator to receive an 'emitKeyword:' message with the KeywordNode as the argument. The code generator will inspect the selector, receiver and argument to determine if the message is a candidate for inlining. In this case it will find the selector is 'ifTrue:' and the argument a literal block, and will therefore inline the operation by sending itself an 'emitIfTrue' message with the KeywordNode as the argument.

```
M68000>>emitIfTrue: message
    "Generate inlined code for 'ifTrue:'; message argument is a
    literal block. Return value of block if executed, otherwise nil."
    | else cont |
```

Figure 6.5(a) shows how conditional execution can be accomplished using the forking mechanism described in the preceding section. The first thing to be done is to generate a new label (for the 'fail' branch) and to ask the receiver to 'generateForkFalse' to this label (the receiver plays the part of the 'test' expression in figure 6.5(a)):

```
message receiver generateForkFalse: (else ← self newLabel) for: self.
```

This will take care of generating the inlined code to perform the comparison, branching to 'else' if the test condition fails. The body of the block can now be compiled inline, moving the result of the last statement into d0 for any enclosing expression to use:

```
self inD0: (message arguments first generateBody: self).
```

Assuming the test failed and the block was not executed, it will be necessary to return 'nil' as the result of the 'ifTrue:', so an instruction must be generated to accomplish this. If the last statement in the block was not a return, then a branch will be needed to skip over this instruction:

```
message arguments first returns
    ifFalse: [self emitBranchTo: (cont ← self newLabel)].
self
    emitLabel: else;
    inD0: (self emitNil).
cont isNil ifFalse: [self emitLabel: cont].
```

Note that the branch after the block body over the ‘else’ case is not needed if the body of the block finished with a return statement. Sending a BlockNode the message ‘returns’ checks for such a return statement. The ‘emitNil’ message does not generate any code, but merely returns an operand descriptor representing the constant ‘nil’.

All that remains to be done is to return an operand descriptor to whoever sent the KeywordNode the original ‘generate’ message. Since both branches of the conditional return their result in d0, this is straightforward:

```
↑d0
```

noindent The full definition of the ‘emitIfTrue:’ method is given in figure 6.6.

```
M68000>>emitIfTrue: message
    "Generate inlined code for ‘ifTrue:’; message argument is a
    literal block. Return value of block if executed, otherwise nil."
    | else cont |
    message receiver generateForkFalse: (else ← self newLabel) for: self.
    self inD0: (message arguments first generateBody: self).
    message arguments first returns
        ifFalse: [self emitBranchTo: (cont ← self newLabel)].
    self
        emitLabel: else;
        inD0: (self emitNil).
    cont isNil ifFalse: [self emitLabel: cont].
    ↑d0
```

**Figure 6.6:** The full definition of the ‘emitIfTrue’ method. See the text for a description of its operation.

---

The code generator methods associated with the other conditional constructs are very similar. For ‘emitIfTrueIfFalse’, the only change is that two literal blocks are expected as the message arguments, and the body of the second is generated inline rather than the single instruction to move ‘nil’ into d0. The two other cases with the test reversed for ‘ifFalse:’ and ‘ifFalse:ifTrue:’ are identical, but with the forking condition reversed (in other words, the ‘generateForkFalse’ is changed to a ‘generateForkTrue’).

#### 6.5.2.4 Code for Loops

There are four looping constructs: ‘whileTrue’ and ‘whileFalse’ for repeat style loops, and ‘whileTrue:’ and ‘whileFalse:’ for while (and ‘$n\frac{1}{2}$’) style loops.[18]

The former pair are detected by the code generator in response to the ‘emitUnary:’ message and, like the conditional constructs, depend on the selector and a literal block receiver for inlining to be attempted. The latter pair are detected, like the conditionals, in response to ‘emitKeyword:’ and require both receiver and argument to be literal blocks.

The inlining of ‘whileTrue’ is straightforward. First, a label is generated to mark the entry point to the loop:

```
emitUnaryWhileTrue: message
    | top |
    self emitLabel: (top ← self newLabel)
```

The receiver (a literal block) is then sent a ‘generateForkTrue’ message, which causes it to generate code for the statements of its body, looping back to the ‘top’ if the result of the last is ‘true’:

```
message receiver generateForkTrue: top for: self
```

The last thing to be done is to return an operand descriptor for the result, which is always ‘nil’ for a loop:

```
↑self emitNil
```

The generation of code for ‘whileFalse’ is similar, with the ‘emitForkTrue’ changed to an ‘emitForkFalse’.

The generation of code for ‘whileTrue:’ follows a similar pattern, but with the existence of a loop body (the block argument) taken into consideration. Again, the first thing to do is to generate a label to mark the top of the loop followed by the body of the test block, branching out of the loop if the condition is not met:

```
emitKeywordWhileTrue: message
    | top exit |
    self emitLabel: (top ← self newLabel).
    message receiver generateForkFalse: (exit ← self newLabel) for: self.
```

Next to be generated is the body of the loop, followed by a branch back to the ‘top’, the label marking the end of the loop, and finally the return value:

[18] For those who never had the privilege of attending a lecture course by Charles Lindsey, an ‘$n\frac{1}{2}$’ loop is one whose exit is in some place other than the beginning or end of the loop body.

```
message arguments first generateBodyFor: self.
↑self
    emitBranch: top;
    emitLabel: exit;
    emitNil
```

Changing the ‘generateForkFalse’ to a ‘generateForkTrue’ produces the definition of ‘emitKeywordWhileFalse’.

### 6.5.3 Other Optimizations

As with the naïve code generator, the only other operations considered for inlining are the two Point accessing messages ‘x’ and ‘y’, and the Point creation message ‘@’. The code generated for inlined Point creation is identical to that given in section 5.5.1. The code generated for inlined sends of ‘x’ and ‘y’ is similar to that given in the same section, but now makes use of the deferred send mechanism.

Section D.4 illustrates the code produced by the delayed code generator described in this chapter.

## 6.6 Code Generation for Other Languages

The remainder of this chapter will introduce delayed code generation techniques in the compilation of a more ‘traditional’ procedural language: C. A full and general treatment of delayed code generation applied to the compilation of C is far beyond the scope of this thesis, so the discussion will be limited to a brief description of the application of the technique with a few examples.

### 6.6.1 Operand Descriptors for Other Addressing Modes

C is a relatively low-level language, allowing the programmer to express quantities and operations that have much in common with the capabilities of the majority of complex instruction set computers. For this reason it generally exercises most of the available addressing modes and instructions provided by architectures such as the MC68020.

The work done by Cordy, Holt and others on data descriptors [Hol87] [CH90] attempts to generalize the representation of operands to cover almost every addressing mode provided by a wide range of CISC architectures. Although code generation based on this approach has some benefits, it suffers from an inherently multi-pass nature in which the last few passes translate the highly generic operations and operands into real instructions and simpler operands, legal for the instructions to which they are attached. In many cases this involves introducing extra instructions to explicitly perform ‘effective address’ calculations that are implied by the generic operands but not supported by the hardware directly.

A delayed code generator constrains all operand descriptors to be within the range of valid operands for the instructions that they are associated with. In doing so it

removes the need for these later passes to the extent where a single-pass compiler is possible. This has a further important consequence in that the semantic gap between the parser and code generator is closed immensely, allowing a much greater amount of 'high level' semantic information to be used to provide simple but effective solutions to resource management problems such as register allocation.

### 6.6.2 Operand Sizing

The 68020 supports four operand sizes: 8-, 16-, 32- and 64-bit words. The actual size of the quantity encoded in an operand descriptor is not needed in Smalltalk since all values (both object pointers and SmallInteger literals) are implicitly 32-bit words. In C however, such information is essential if an operand descriptor is to be treated correctly. We therefore tag each operand descriptor with its size in bytes, so a long-word variable would be represented as:

$$[_a]_4$$

The size information is used by most instructions, which come in several variations depending on the size of their operand(s). The situation is slightly more complicated when it comes to dyadic operations (arithmetic or move instructions) which have to take appropriate action if the operands do not have equal sizes. For example, moving a byte quantity from a register into a long-word variable would necessitate an extra '`extbl`' instruction to extend the byte quantity into a long-word quantity before the '`movl`' instruction could be generated. The details of these conversions are intricate but not difficult to derive, and so will not be discussed further.

### 6.6.3 Logical Values

In the Smalltalk-80 code generator presented in the first half of this chapter, all values were manifest in some physical location. Operations such as 'forking' to implement the control constructs (which rely on values implicit in the condition codes register) were achieved atomically with a single message sent to the code generator, and resulted in a concrete value being returned (either the value of the last statement in a block, or 'nil'). It was not necessary to represent logical values implied by the condition codes in an operand descriptor.

Languages such as C associate both a physical (numerical) *and* a logical interpretation with all values. It is therefore necessary to extend operand descriptors to cater for both of these interpretations simultaneously. This is done by adding an attribute representing a logical condition to each operand descriptor. When an instruction is generated that will set the condition codes according to its result, the operand descriptor is tagged with the condition which would imply the result was 'true'.

For example, an assignment in C yields a true result if the quantity moved is non-zero. Since the '`move`' instruction used to effect the assignment will set the condition codes to reflect its source operand, we can tag the returned data descriptor with the condition '`ne`' (a non-zero result implying 'true'). This tells the code generator that

emitting a '`jne`' instruction will cause a branch if the result of the assignment was 'true'. More usefully, for the results of comparisons we can build an operand descriptor that has no concrete value but instead represents the result in the condition codes of the comparison operation. For example, the comparison '`a < b`' would generate the following code:

```
movl   _a, d0
cmpl   _b, d0
[]lt
```

If used as the test in an '`if`' statement, the next instruction would branch on the logical inverse of the condition in the operand descriptor:

```
jge   failLabel
```

This mechanism has several rather appealing consequences. The logical 'not' operator can be applied to an operand descriptor by simply inverting its associated logical condition. Literals whose operand descriptors are generated in leaf nodes automatically have their logical condition set to '`t`' for non-zero and '`f`' for zero quantities, making the generation of optimal code for constructs such as

`while(1)`*doSomething*`;`

quite straightforward. (In this case the test would generate no code at all, and the loop body would terminate with an unconditional jump back to the empty test.)

Depending on the type of physical value represented, an operand descriptor may not have an associated logical meaning. For example, at a leaf node representing the variable '`a`', the logical interpretation can only be determined at run-time. Whenever a branch instruction is required based on an operand descriptor with no logical interpretation attached to it, an explicit test instruction must be generated. For example, the code produced in response to

aMachine emitBranchTrue: [`_a`]

would be:

```
tstl   _a            | [_a] → [_a]ne
jne    destination
```

Conversely, when a physical interpretation is required of a purely logical value, an instruction must be generated to coerce the implied value into an explicit value. For example, '`a= (b < c);`' will end by sending the code generator the message

aMachine move: $[]^{lt}$ to: [`_a`]

which generates:

```
sge     d0      | set (byte) on inverse of condition
extbl   d0      | set (long) on inverse of condition
addql   #1, d0  | set (long) on condition
movl    d0, _a  | do the assignment
[d0]^ne
```

leaving 0 or 1 in '_a' if the condition was false or true, respectively. The coercion of logical into physical values in this manner must be performed by the code generator on any operand descriptor used in a source position.

### 6.6.4 Coercion of Operand Types

Most CISC architectures do not support fully orthogonal addressability of operands.[19] The MC68020 is no different to the majority in this respect, with all dyadic instructions placing some limitations on the combination of operands which can be used. The 'move' instruction is the most general, allowing any addressing mode in the source field but only "data alterable" modes in the destination field.[20] All other dyadic instructions impose a much stricter set of constraints, such as only allowing a general addressing mode in one position if the other position contains a data register.

It is therefore inevitable that there will be occasions when an attempt will be made to generate an instruction with an illegal combination of operand types. These situations must be resolved by the code generator in a systematic and sensible manner. The obvious place for the information regarding the legal combinations of operand types is in the class of each particular instruction. The best time at which to perform any coercion of operand types from an incompatible set into a compatible set is less certain, but a good choice would seem to be at the time of the instantiation of the instruction itself. This can be illustrated with an example based on the 'add' instruction.

The 'add' instruction allows only the following combinations of operand types:

| *source* | *destination* |
|---|---|
| <ea> | dN |
| dN | <ea> |
| <ea> | aN |
| #<data> | <ea> |

This is typical of many MC68020 dyadic instructions. It is clear that if the operands presented for addition are not of a legal combination, then moving the source into a data register will certainly remove the conflict.

To show the operation of this abstraction mechanism in practice, we shall work through a short example. Consider the statement 'a= (b+= c)'. The right hand side, 'b+= c', will be compiled first, by sending the code generator the message

[19]Again, the wonderfully notable exception to this is the PDP-11 which even allows immediate operands in both source and destination positions (this actually has some practical uses!).

[20]See [Mot85, table B-1 and page B-101].

aMachine add: $[_c]_4$ to: $[_b]_4$

The abstract addition operation is defined in M68000 as:[21]

```
M68000>>add: src to: dst
   | s |
   s ← ((src isImmediate or: [dst isReg])
           ifTrue: [src]
           ifFalse: [self inDReg: src]).
   codeStream nextPut: (M68add source: s release dest: dst).
   ↑dst fix
```

The combination of operand types presented for addition is not legal, so an instruction must be generated to coerce the source into an acceptable operand type before generating the actual 'add' instruction. This is achieved by the 'inDReg' message to the machine, passing the unacceptable operand as the argument. The 'inDReg' message is defined as:[22]

```
M68000>>inDReg: operand
   | dReg |
   operand isDReg ifTrue: [↑operand].
   codeStream nextPut:
      (M68move source: operand release dest: (dReg ← self newDReg).
   ↑dReg
```

Once the operand is in the data register, the 'add' instruction can be generated from within the 'add:to:' method, and the destination operand returned as the result:

```
movl   _c, d0
addl   d0, _b
```
$[_b]_4^{ne}$

The result of the addition is $[_b]_4^{ne}$, which forms the source for the assignment:

aMachine move: $[_b]_4^{ne}$ to: $[_a]_4$

which causes the instruction

```
movl _b, _a
```
$[_a]_4^{ne}$

to be generated.

Although trivial, this example illustrates the approach to the generation of code by abstract operations. Abstract operations defined for other instructions, which *will* have to constrain the operand types in the final generated code, will follow a similar format.

[21] The meaning of the 'fix' and 'release' messages will be explained in sections 6.6.5 and 6.6.7.1 respectively.

[22] The 'newDReg' message will be discussed in section 6.6.7.1.

### 6.6.5 Operands with Side Effects

Quite a few CISC architectures support addressing modes which have side effects when used. The MC68020 provides two such modes, address register indirect with post-increment and pre-decrement. After using one of these modes, the associated address register will be either incremented or decremented by the size of the data referred to. Operand descriptors representing these modes that have been used in a generated instruction cannot be returned as the result of an operation since they will not refer to the same location once their side effect has been incurred. It is therefore necessary to 'fix' any side effects associated with operand descriptors before returning them for further use.

For example, an operand descriptor representing a long-word quantity at the address held in `a3` with an implied post-increment would be represented as:

$$[\texttt{a3@+}]_4$$

Once used in an instruction, the operand descriptor cannot be used as a reference to the same value. By returning instead the result of sending it the 'fix' message, it has a chance to return another operand descriptor that does represent the original location. In this example, the result of sending the 'fix' message would be

$$[\texttt{a3@(-4)}]_4$$

which represents the original value.

### 6.6.6 Argument Order

Some language implementations (including most implementations of C) require the arguments to functions to be pushed in reverse order, starting with the last argument and working towards the first. In C this is usually necessary since functions support variable numbers of arguments ('`printf()`' for example), and pushing the arguments in the actual order they occur in a function call can create severe implementation difficulties and/or inefficiencies.[23]

If an explicit parse tree is available then this is not a problem, since the branches representing the arguments are simply visited in reverse order. However, in a single-pass compiler this could cause problems, since the arguments must be parsed strictly left to right. Delayed code generation offers an effective solution to this problem.

When parsing and generating code for a function call, each argument will ultimately be represented as an operand descriptor. These operand descriptors not only represent the value of the argument, but also own and protect any resources (such as

[23] This concerns the problem of finding the address of the first function call argument in the stack. This is trivial if the arguments are pushed in reverse order, but which otherwise depends on the number of actual arguments at runtime. Since the number of actual arguments cannot be determined at compile-time, some additional mechanism (such as the passing of an invisible last argument containing the argument count) is required.

registers) that the value depends upon. By pushing the operand descriptors for the arguments onto a (compile-time) stack, resources required for the earlier arguments are automatically protected while code is generated for the later arguments. When code for all the arguments has been generated, their operand descriptors can be popped off the compiler's stack, pushed onto the (run-time) stack in the required reverse order, and their resources released. No run-time performance penalties at all are incurred due to the reversal of the argument order using this mechanism.

Section D.3 gives a worked example of the technique.

### 6.6.7 Register Allocation

Register allocation fits neatly into the delayed code generation model using operand descriptors. The allocation strategy described below is much simpler than register allocators using graph coloring techniques yet produces code that is almost as efficient in space and time, and more efficient in machine resource usage.

Graph-coloring allocators generally aim to improve the efficiency of generated code by minimizing the movement of values between memory (where most data resides) and the machine registers (where computation takes place most efficiently). Not only is the use of data from memory slower than using cached copies from registers, but register to register instructions are usually shorter than memory to memory, or memory to register, instructions. Graph coloring techniques are most valuable in RISC environments where all computation takes place in machine registers, and where the overheads of moving data between registers and memory are high.

The allocator described in this section aims to minimize the number of registers used for a given expression. For many languages (including C) implemented on conventional processors, each function must save and restore any registers it modifies. Traditionally 'good' programming practice leans towards many small functions each performing a specific task, in which case the overheads associated with saving and restoring the register context during function entry and exit can become significant.

The allocation strategy used is similar to the strategy used in [ASU86, section 5.8] for the allocation of space at compile-time for attribute values during syntax-directed translation.

#### 6.6.7.1 Machine Models and Allocation Strategy

Operand descriptors model the capabilities of the target architecture in terms of the legal addressing modes that can be used in the instructions of the generated code. A useful technique, for register-oriented execution models, is to model the use of the registers of the target architecture at compile time.

The available registers of the target machine will be modeled as a set of free registers, the 'register pool'. When an instruction is generated that requires an unused register for its destination, an operand descriptor will be created to represent the register location. The register is removed from the pool, possibly by simply setting a flag in the pool to indicate the register is in use, and the operand descriptor initialized with

the appropriate register number (figure 6.7a). Alternatively, in a compiler written in a more object-oriented style, the object representing the register could be physically removed from the pool and stored within the operand descriptor as the appropriate part of the operand. For architectures such as the MC68020 which have different register classes optimized for either addressing or arithmetic, it is convenient to maintain a separate pool of unused registers for each register class. The required class of register will be specified when the code generator requests an operand descriptor which refers to and depends upon a machine register.

In the delayed code generator under discussion, a pool of free data registers is held in an instance variable of the machine object called 'dRegs' (a SortedCollection, sorted by register number). A register can be allocated from this pool by requesting a 'newDReg':

```
M68000>>newDReg
    dRegs isEmpty ifTrue: [self error: 'expression too complex'].
    ↑dRegs removeFirst
```

A similar method exists for the allocation of address registers.

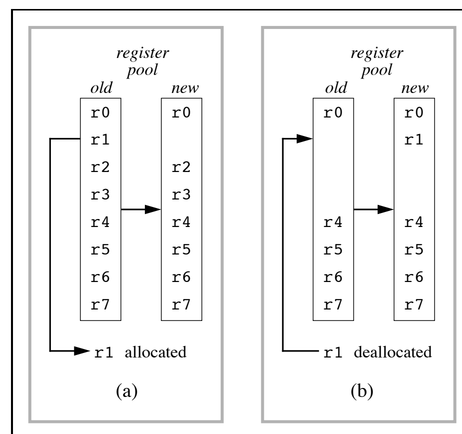


**Figure 6.7:** Register allocation and deallocation using a pool of free registers. A register allocated to an operand descriptor for use as a destination in a generated instruction (a) must later be returned to the pool from whence it came (b). Identifying the exact moment at which an operand descriptor, and hence any registers that are intrinsically part of the location it specifies, 'dies' is straightforward given the amount of semantic information available to the code generator (see text).

---

Immediately after the last use of any operand descriptor which refers to a register, and which contains that register as part of the operand that it represents, that register must be returned to the pool for reuse. Responsibility for the allocation and deallocation of registers is shared between the code generator (machine object) and

the parse tree nodes. The former must sometimes allocate registers while (for example) coercing operands into legal types, as in the 'M68000>>inDReg' method defined on page 104. For the same reason it must also deallocate resources no longer needed in 'stale' operand descriptors. Parse nodes must deallocate resources held in operand descriptors when the value they represent is no longer required for future computation. Deallocation is performed by asking an operand descriptor to 'release' any resources it owns. For example, a descriptor representing a data register responds as follows:

```
M68DReg>>release
    pool add: self
```

where 'pool' is an instance variable containing the register pool appropriate for this particular kind of register. Operand descriptors that do not own resources (such as immediate quantities or absolute addresses) inherit a default definition of 'release' that returns the receiver without performing any other action.

Resource management by this technique is particularly effective due, as has been stressed before, to the breadth of information available: from low-level machine dependent information known to only the code generator during (for example) operand coercion, through to high-level semantic information regarding the lifetimes of values known only to the parse tree nodes (or the syntax analyzer in a single-pass compiler).

Most procedural languages reserve certain registers for implementation uses (such as stack and frame pointers) or for 'register variables'. Such registers should not be allocated for use as temporaries, and so are omitted from the relevant pool during initialization. It is also trivial to arrange for them not to return themselves to the pool when released, most easily by building a dummy pool (distinct from that which holds the available resources) to which they 'add:' themselves when released. Resources required for register variables can simply be removed from the pool before, and returned to the pool after, generating code for the body of the function.

#### 6.6.7.2 Register Allocator Performance

As an example of this register allocation strategy, the code for a simple expression produced by both a delayed code generator using operand descriptors, and a graph coloring register allocator, will be compared.

The graph coloring algorithm used is based on that given in [ASU86, section 9.7], although no heuristic is given there for which node to remove from the interference graph while finding an ordering of the nodes to use when coloring the graph. The simple heuristic used in the examples given in this thesis chooses to remove the node with the highest number of incident arcs, thereby maximizing the number of other nodes whose connectivities are reduced by its removal.

Graph coloring allocators usually work with a 3-address intermediate code. The 3-address code for an arbitrarily chosen expression and a linearized representation of its associated interference graph are shown in figure 6.8. The parse tree for the same expression, showing the operand descriptors and the register allocator activity, appears in figure 6.9.

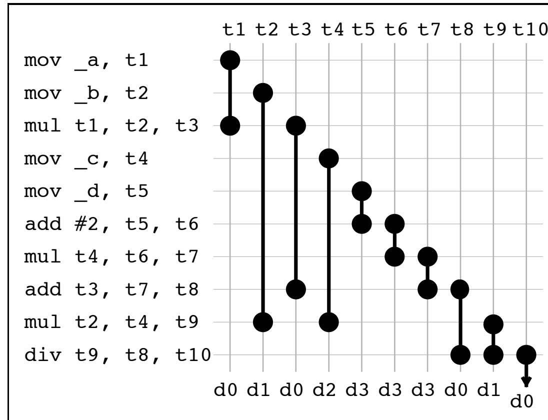


**Figure 6.8:** 3-address intermediate code and the associated graph coloring for the expression '`(a*b+c*(d+2))/(b*c)`'. Unique temporary variables used in the intermediate representation are assigned to actual physical registers during the mapping of the generic operations onto actual machine instructions. The final result appears in `d0`.

---

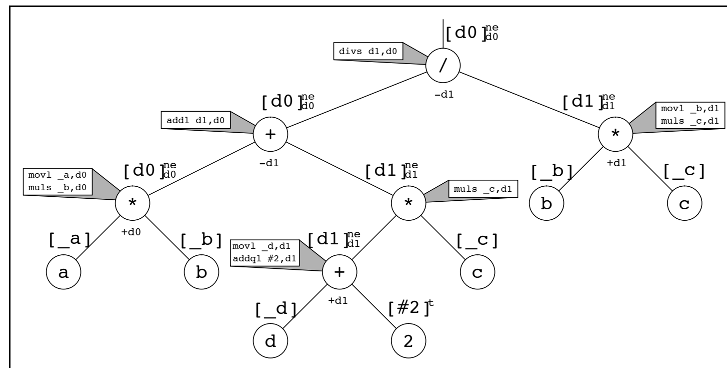


**Figure 6.9:** Parse tree, operand descriptors, and register allocator activity for the expression '`(a*b+c*(d+2))/(b*c)`'. Operand descriptors are shown next to the arcs along which they are propagated. Register allocation and deallocation is shown under the associated parse tree node as '`+dN`' and '`-dN`' respectively.

---

The final MC68020 code produced from the translation of the 3-address code with register names substituted for temporary names, and the code produced directly from walking the parse tree using delayed code generation with operand descriptor-based register allocation, is shown in figure 6.10.

Although a single example is completely inadequate to compare the effectiveness of the two allocation strategies, it does serve to show that the general goals are different.

| translated 3-address code | DCG generated code |
|---|---|
| <pre>movl   _a, d0<br>movl   _b, d1<br>muls   d1, d0<br>movl   _c, d2<br>movl   _d, d3<br>addql  #2, d3<br>muls   d2, d3<br>addl   d3, d0<br>muls   d2, d1<br>divs   d1, d0</pre> | <pre>movl  _a, d0<br>muls  _b, d0<br>movl  _d, d1<br>addql #2, d1<br>muls  _c, d1<br>addl  d1, d0<br>movl  _b, d1<br>muls  _c, d1<br>divs  d1, d0</pre> |
| (a) | (b) |

**Figure 6.10:** Final 68020 code generated for the expression '`(a*b + c*(d+2)) / (b*c)`'. **(a)** The code produced by translating 3-address code and allocating registers by graph coloring uses four registers and four memory accesses in ten instructions. **(b)** The code produced by a delayed code generator uses two registers and six memory references in nine instructions. Neither can be said to be the 'better' version, since the choice of metrics depends to a great extent on the context in which the code is compiled.

---

Code generators using graph coloring allocators typically try to improve performance by caching 'live' values in registers,[24] the idea being that minimizing the number of memory accesses performed increases the efficiency (in both space and time) of the generated code. These goals are perfectly feasible in most circumstances, but can be the cause of register spills when compiling complex expressions. The delayed code generator-based register allocator has slightly different goals – trying to reduce the overall number of registers, ignoring future uses for a value cached in a register.[25] This could be an important consideration in programs where functions at the leaves of the call graph are small, and where register save and restore sequences at the entry and exit points of functions have a significant impact on execution time.

## 6.7 Summary

The naïve compiler of the previous chapter generates rather poor code for some common Smalltalk-80 operations. The main reason for this is the short-sightedness of the code generator, which choses to place every value mentioned in the source program

[24]The 'life' of a value runs from its first use (definition) until its last use.

[25]The building of an explicit parse tree allows the code generator to determine a Sethi-Ullman complexity [ASU86, page 561] for each branch at a particular node, allowing the walk of the parse tree to be ordered so as to reduce the number of registers allocated. If this is done, the delayed code generator-based register allocator will guarantee to allocate the minimum number of registers required to compute any given expression.

into a physical register or memory location before leaving the parse tree node representing that value. The use of delayed code generation to lift this restriction improves the quality of the generated code enormously — to the extent that peephole optimization becomes redundant.

Delayed code generation also seems a promising technique for other, more traditional languages such as C. Common tasks given to the code generator, such as register allocation, can be accommodated easily within the delayed code generation mechanism, and experiments with small examples have been encouraging. Some of the usual compilation headaches for single-pass compilers (such as pushing arguments from right-to-left while parsing them from left-to-right) can also be addressed successfully with delayed code generation.

# Chapter 7

# Results

> There are three kinds of lies: lies, damned lies, and statistics.
>
> Disraeli.

The purpose of this chapter is to assess the validity of the thesis: that a delayed code generator with no peephole optimizations can produce code that is as at least as good as that produced by a more conventional technique incorporating peephole optimization.

Several aspects of the performance of the code produced by the two code generators will be presented, as follows:

- **Execution time**
  The most obvious candidate for measurement is the speed of execution of the generated code. This has been measured for all three systems: PS2.3, the naïve code generator, and the delayed code generator. For the latter two, an uninstrumented virtual machine with no debugging support was used in order to achieve the most realistic results possible.[1]

- **Code size**
  Although it may seem that the number of instructions generated would be an interesting thing to measure, it is not really that useful. For example, open-coded loops contain more machine instructions that close-coded loops, yet run significantly more efficiently. What is much more important is the overall code size. In a virtual memory system with a modest amount of memory (a typical workstation has about 8Mb of physical memory) performance can be adversely affected by seemingly small increases in the size of the generated code due to the very sharp knee in the curve of code size versus paging activity. At least as important is the effect of code size on the processor's instruction-cache, which tends to be small. Small reductions in the spatial locality of a program, due

[1]This is a polite way of saying that the runtime system was going as fast as it could be made to go without compromising the support required by Smalltalk.

to small increases in its size, can drastically increase the traffic between the instruction cache and primary memory.

- **Memory usage**
  The importance of this aspect of the generated code should not be underestimated since it can have a large impact on the performance of a system. Not only is the actual act of allocating storage a relatively costly operation[2] a reduction in the size and/or allocation frequency of objects will reduce both the virtual memory requirements and the overheads due to garbage collection.[3]

The only important result from these figures (insofar as the validation of the thesis is concerned) is the relative performance of the naïve versus delayed code generators for real-world problems (represented here by the macro benchmarks) with all optimizations enabled. This should provide a good indication as to whether the delayed code generator can compete with, or even outperform, the naïve. Nevertheless, for completeness the full set of figures has been included in appendix B so that an interested reader can see the influence on the performance of each particular optimization technique.

## 7.1 A Brief Note Concerning Definitions

Before presenting any of the results it is important to clarify a few points of potential confusion. Most of the comparisons in this chapter and in appendix B concern changes to aspects of the performance of the system in response to variations in the environment in which the code is being compiled or executed. Relative performances are always presented as the ratio of:

$$\frac{\text{dcg measurement}}{\text{naïve measurement}} \ \mathit{or} \ \frac{\text{dcg measurement}}{\text{PS2.3 measurement}}$$

Changes are presented as percentages according to the conventions discussed in [HP90, chapter 1], where the percentage change from $A$ to $B$ is given by:

$$\mathit{change} = \frac{B - A}{A} \times 100$$

[2]For example, allocating a single object is generally much more costly than performing a dynamic bind. However, the impact that memory management has on the overall performance of the system has as much to do with the efficiency of the implementation of the runtime system as it does with the number of objects required by the compiled code.

[3]Smalltalk implementations almost invariably allocate a fixed sized object memory. The garbage collection overhead is thus directly proportional to both the frequency of object allocation and the average size of each object.

## 7.2 Performance of Generated Code

Table 7.1 shows the relative performance of the naïve and delayed code generators. The measurements were made in as realistic an environment as possible: the images used had all optimizations enabled, and the timings were the aggregate of the times for the three macro benchmark classes that exercise large subsystems of Smalltalk.

| | compiler | | | |
|---|---|---|---|---|
| quantity | naïve | dcg | ratio | change |
| image size | 1919384 | 2051676 | 106.9 | 6.9% |
| execution time | 63.34 | 58.78 | 92.8 | –7.2% |
| objects allocated | 84247 | 81927 | 97.2 | –2.8% |
| bytes allocated | 2531804 | 2418764 | 95.5 | –4.5% |
| message sends | 1932592 | 1782457 | 92.2 | –7.8% |
| primitive calls | 994402 | 915421 | 92.1 | –7.9% |

**Table 7.1:** Relative performance of the naïve and delayed code generators. The first two columns show the absolute performance for: the total image size (bytes), the total execution time for the macro benchmarks (seconds), the numbers of objects and bytes allocated for the macro benchmarks, and the numbers of message sends and primitive calls made for the macro benchmarks. The third column shows the relative performance for the delayed code generator with respect to the naïve as a percentage of the latter.

Overall, delayed code generation has had two clear effects. First, the size of the generated code has increased by approximately 6.9%. This is due to additional opportunities that the code generator found for inlining various operations. Second, the execution time of the generated code has decreased by approximately 7.2%, due in part to the additional inlined operations but also to the more efficient code that can be produced in certain situations (see sections 6.1 and 6.5).

## 7.3 Compiler Efficiency

Very early in this thesis it was noted that one of the overriding concerns for a compiler in an EPE is that it be as fast as possible while maintaining a reasonable standard of generated code. Table 7.2 shows the time spent in the various phases of compilation for the naïve and delayed code generators, during the compilation of the 790 methods that were required by the benchmark suite.
The times for the following phases are shown:

- **parse**
  For both compilers this is the time spent performing lexical and syntactic analysis of the method source, including the time taken to generate the initial parse

| phase | compiler | |
|---|---|---|
| | **naïve** | **dcg** |
| parse | 98.1 | 97.0 |
| parse tree optimization | 12.0 | *nil* |
| code generation | 37.1 | 179.4 |
| code optimization | 226.7 | *nil* |
| total | 373.9 | 276.4 |

**Table 7.2:** Compilation times for the naïve and DCG-based compilers. All times are the total in seconds for the compilation of the 790 methods required for the benchmark suite.

tree.

- **parse tree optimization**
  For the naïve compiler only this is the time taken to perform constant folding on the parse tree for certain (otherwise inlined) selectors.

- **code generation**
  This is the time taken to walk the parse tree and generate a stream of 68020 instructions.

- **code optimization**
  For the naïve compiler only this is the time taken to perform peephole optimization on the raw 68020 code to remove redundancies and dead code.

The extra complexity of the delayed code generator has added 142.3 seconds to that phase of compilation — an increase of 384%. However, the savings made by not having to perform the very costly peephole optimization pass more than compensate for this, reducing the overall code generation time by 35%,[4] and the overall compilation time by 26%.

## 7.4 Summary

This chapter presented the performance of a two-pass Smalltalk compiler that builds a parse tree and then walks it to generate code. The back-end for this compiler is pluggable, and the performance was analyzed for both a naïve and a novel code generator. The naïve code generator performs parse-tree optimizations before generating

[4]The overall code generation time for the naïve compiler is taken to be the sum of the times to generate raw code, and perform the optimizations on both the parse tree and the generated code. This is realistic since the delayed code generator performs the equivalent of all three of these operations during its single code generation phase.

low quality code, which is tidied up by a peephole optimizer. The more sophisticated 'delayed' code generator performs many of the same optimizations during code generation, and needs no separate parse tree or peephole optimization passes.

The results show that using a delayed code generator produces a compiler that takes less time to generate better code (in terms of overall performance) than does the use of a three-pass 'optimizing' compiler.[5]

[5] Whether or not parse tree optimization should be considered a distinct pass is open to question.

# Chapter 8

# Conclusion

"Well" said Owl, "the customary procedure in such cases is as follows..."

"What does Crustimoney Proseedcake mean?" said Pooh, "For I am a bear of very little brain and long words bother me."

But Owl went on using longer and longer words, until at last he came back to where he had started.

A. A. Milne.

This thesis began by speculating that it is possible to compile Small- talk-80 directly into machine code for stock hardware, and to do this efficiently in terms of both compiler performance (the compiler must be small and fast) and generated code performance. The results presented in chapter 7 demonstrate that this is true. The techniques developed in chapter 6 for 'delayed code generation' in single-pass recursive descent compilation were demonstrated to produce better code in less time than a compiler using a naïve code generator followed by peephole optimization. Appendix B also shows that the compiled code ran at speeds competitive with a commercially available implementation.[1]

A brief investigation of the techniques applied to the compilation of a popular procedural language showed promising results. Some of the more difficult problems associated with the compilation of procedural languages by one- or two-pass compilation techniques were also addressed using delayed code generation with good results.

## 8.1 Future Work

The full-scale delayed code generator developed for Smalltalk-80 uses a stack-oriented execution model. It would be interesting to develop a register-oriented execution model for Smalltalk-80 and investigate the performance gains made, for example, by

[1]However, it should be remembered that several important aspects of the runtime system were ignored (section 5.1.5), and a proper solution to these would degrade the performance of the generated code.

passing arguments in registers rather than on the stack. The register discipline would have to be constructed carefully to avoid problems with mixed data (OOPS and non-OOPS in registers) during garbage collection.

Further investigation of the applicability of delayed code generation to the compilation of procedural languages would also be worthwhile. The integration of register allocation policies with delayed code generation seems a particularly interesting area.

Possibly most important of all would be an investigation of delayed code generation for RISC architectures.[2] Several problems arise when targeting to RISC architectures that do not arise for CISC architectures, not least of which are those of instruction scheduling and the utilization of delay slots. The problem of filling delay slots in RISC code seems similar in some respects to the problem of filling operand positions in CISC code. In the delayed code generator, the filling of these operand positions is performed *after* visiting the parse tree node in which they are generated. It may be possible to extend the delayed code generation technique to cover entire instructions in addition to operands, in order to provide scope for instruction scheduling and delay slot filling within the code generator.

[2]The RISC versus CISC debate of the late 1980s seems to have been won hands-down by the RISC team.

# Appendix A

# Parse Tree Nodes

## A.1 Leaf Nodes

Literals (including smallintegers, strings and literal arrays) are simply encapsulated by a LiteralNode.

Variables come in four flavors: global, argument, temporary, and instance variables. Each type has its own associated node (figure A.1). For global variables the node contains the name of the variable and the corresponding association in the SystemDictionary. The contents of the global variable appear in the value field of this association. The other three types of node contain the name of the variable, and its index. The code generator uses the index of the variable to generate offsets from the frame pointer (for arguments and temporaries) or from the start of the receiver in memory (for instance variables).

The pseudo variables self and super are represented by their own unique nodes (SelfNode and SuperNode, respectively) which implicitly refer to the method's first argument. It is necessary to have a separate SuperNode since sends to super do not behave like sends to self.

## A.2 Message Nodes

There are four types of message node: one each for unary, binary and keyword messages, and one for cascaded messages (figure A.2). UnaryNodes contain the selector and the node representing the receiver. BinaryNodes contain the selector and nodes representing the receiver and argument. KeywordNodes contain the selector, a node representing the receiver, and an array containing nodes for the arguments. CascadeNodes contain a node for the receiver and an array of message nodes.

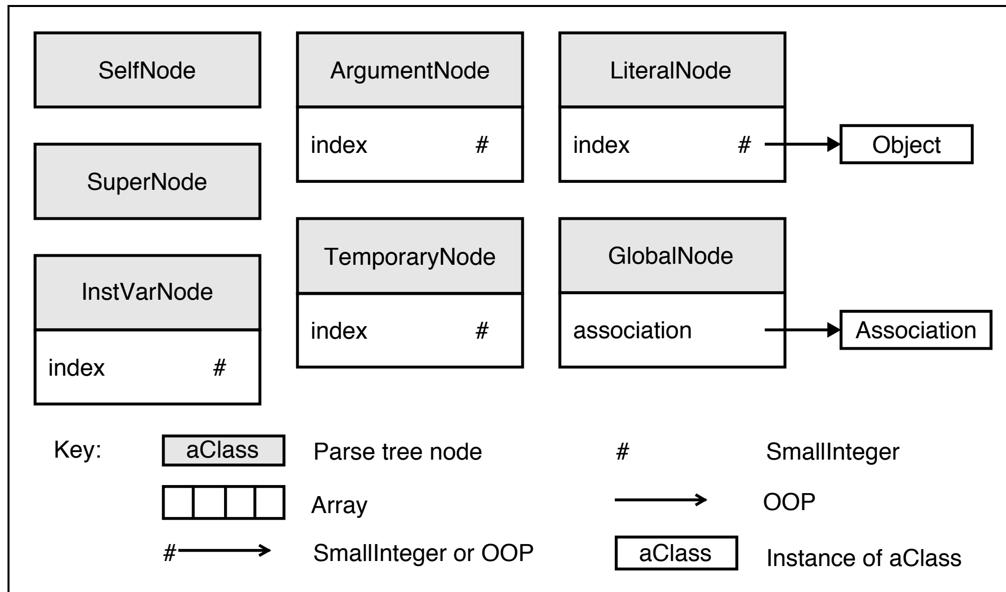


**Figure A.1:** Leaf nodes are generated when either literals or variables are encountered in the source. LiteralNodes contain the value they represent. GlobalNodes contain a reference to the association that contains the value of the variable.

## A.3 Special Action Nodes

The nodes representing assignment, return statements, and primitive methods are shown in figure A.3. AssignmentNodes contain a node each for the left hand side (which must be a leaf node representing a variable) and the right hand side. PrimitiveNodes only ever occur at the top of the parse tree and contain the number of the primitive concerned and a node containing the tree for the fail case code. ReturnNodes contain a single node representing the returned value.

## A.4 Method and Block Nodes

The remaining types of node represent blocks and entire methods (figure A.4).

BlockNodes contain an array of nodes representing the statements of the block's body, and an array of variable nodes representing the arguments (if any). For convenience, they also contain a flag which is true if the block performs a non-local return, and a label by which the block is known during code generation.

MethodNodes contain an array of nodes for the method body, two arrays containing the argument and temporary variables, and the 'message pattern' for the method (the method's selector). These nodes also hold references to the Mapper and 'machine object' created for the method (see sections 5.3.1 and 5.4.1, respectively).

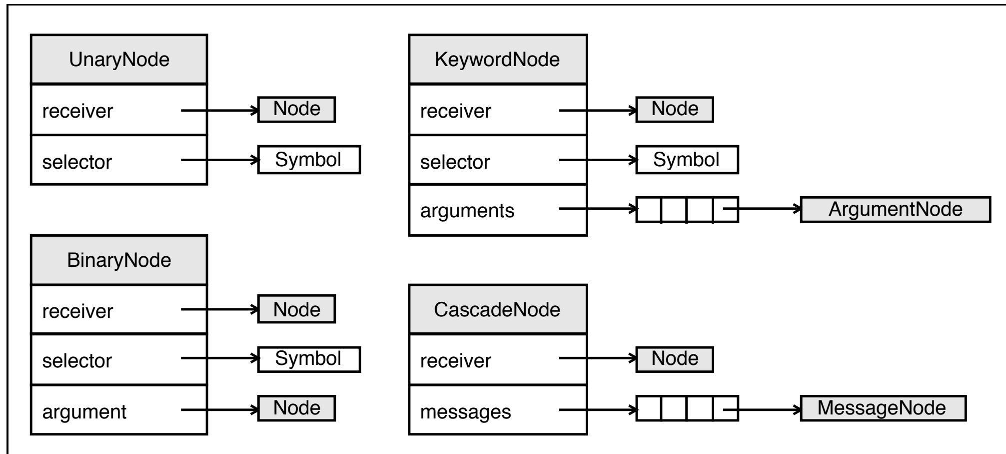


**Figure A.2:** Message nodes must all specify a receiver and selector. In addition, BinaryNodes also specify a single argument, KeywordNodes specify multiple arguments, and CascadeNodes specify multiple messages meant for the single receiver.

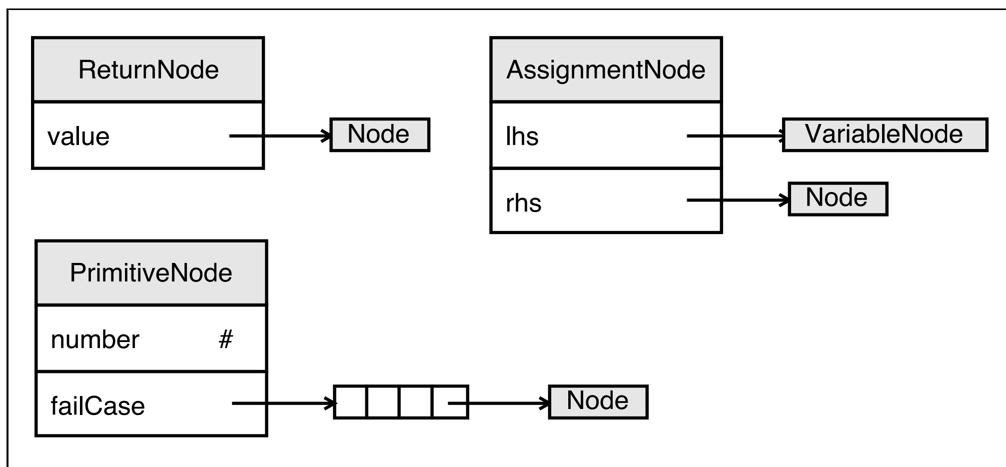


**Figure A.3:** ReturnNodes simply encapsulate the statement whose value is to be returned. PrimitiveNodes contain the primitive number and an array of statements which form the body of the method to be executed if the primitive fails. AssignmentNodes contain a VariableNode of some class for the left hand side, and an expression for the right hand side.

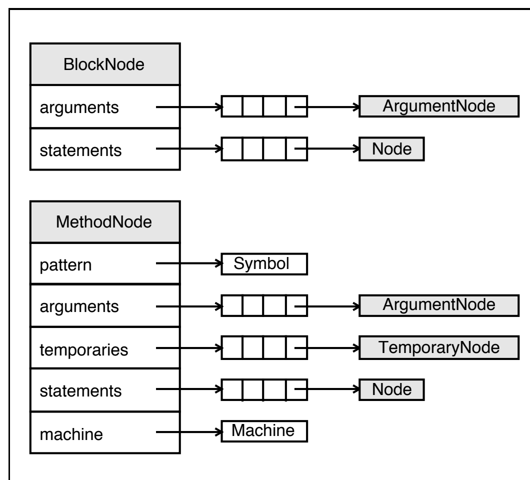


**Figure A.4:** BlockNodes contain an array of arguments and an array of statements which form the body of the block. MethodNodes contain the message pattern (selector) of the method, arrays of arguments, temporaries and statements (for the method body), and an instance of Machine which is used to generate code and manage the use of the target hardware resources.

# Appendix B

# Raw Results

This appendix contains some of the raw, unbridled results from the benchmarking work.

The horizontal axis in each table ranges over the classes of benchmarks. The selectors and activities targeted in each class are described in chapter 4. The vertical divisions delimit the micro benchmarks from the macro benchmarks. The three macro benchmark classes are much more realistic indicators of system performance, so a final column gives the sum of the figures for the three classes of macro benchmark. The names of the benchmark classes in the tables are mnemonic (due to space limitations) as shown in figure B.1.

The vertical axis ranges over optimizations. Each optimization is preceded by either '+' meaning enabled, or '−' meaning disabled. For the '+' cases, all other optimizations are disabled. For the '−' cases, all other optimizations are enabled. Two special cases are '+none' meaning no optimizations enabled and '−none' meaning all optimizations enabled. Each optimization has a mnemonic label in the tables, as shown in figure B.2.

| short name | full description |
|---|---|
| | micro benchmarks |
| arith | arithmetic and comparison operations |
| cond | conditional constructs |
| loop | looping constructs |
| access | literal and variable access |
| activ | block and method activation |
| point | point creation and access |
| mem | memory management intensive |
| nolook | miscellaneous no-lookup operations |
| gfx | graphics ('BitBlt') operations |
| | macro benchmarks |
| struct | data structure access |
| num | number crunching |
| iface | user interface operations |
| macro+= | sum of macro benchmark results |

**Figure B.1:** The mnemonic names and full descriptions used for the benchmark class names in the tables in this appendix.

| short name | full description |
|---|---|
| + | indicates the optimization is *enabled* and all others *disabled* |
| – | indicates the optimization is *disabled* and all others *enabled* |
| arith | arithmetic operations are inlined for SmallInteger arguments |
| at | point creation is inlined |
| class | the 'class' message is inlined |
| if | conditional constructs are inlined |
| none | the "null optimization" (see text) |
| tests | relational operations are inlined for SmallInteger arguments |
| while | looping constructs are inlined |
| xy | access to instances variables is inlined for Point receivers |

**Figure B.2:** The mnemonic names and full descriptions used for the compiler optimizations in the tables in this appendix.

| time | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 86.72 | 4.10 | 89.26 | 226.70 | 54.26 | 19.30 | 1.80 | 26.82 | 18.36 | 25.80 | 50.66 | 77.08 | 153.54 |
| +at | 111.28 | 3.62 | 93.12 | 210.80 | 64.40 | 17.52 | 1.82 | 28.54 | 18.26 | 28.50 | 61.86 | 84.38 | 174.74 |
| +class | 112.24 | 3.68 | 93.88 | 223.46 | 60.38 | 18.62 | 1.74 | 27.66 | 17.64 | 27.16 | 58.30 | 80.70 | 166.16 |
| +if | 99.94 | 0.18 | 62.14 | 219.80 | 35.14 | 17.94 | 1.78 | 27.30 | 17.58 | 19.76 | 29.20 | 65.06 | 114.02 |
| +none | 112.08 | 3.62 | 93.92 | 226.92 | 61.58 | 19.92 | 1.86 | 29.52 | 19.56 | 28.24 | 60.76 | 87.12 | 176.12 |
| +tests | 94.38 | 3.62 | 84.16 | 209.64 | 60.80 | 17.64 | 1.76 | 27.18 | 17.62 | 26.28 | 51.44 | 76.94 | 154.66 |
| +while | 95.80 | 3.64 | 22.14 | 210.30 | 61.00 | 17.88 | 1.76 | 27.74 | 17.64 | 20.52 | 58.64 | 56.76 | 135.92 |
| +xy | 110.94 | 3.68 | 93.28 | 214.88 | 63.50 | 5.22 | 1.82 | 28.22 | 18.22 | 28.36 | 61.20 | 84.12 | 173.68 |
| -arith | 72.56 | 0.18 | 12.98 | 209.92 | 34.58 | 4.28 | 1.88 | 27.78 | 17.18 | 15.02 | 21.68 | 44.98 | 81.68 |
| -at | 45.16 | 0.18 | 4.30 | 210.72 | 25.50 | 5.14 | 1.84 | 26.74 | 17.20 | 12.68 | 12.40 | 38.52 | 63.60 |
| -class | 45.36 | 0.18 | 4.26 | 217.08 | 24.60 | 4.30 | 1.88 | 26.20 | 17.12 | 12.62 | 12.28 | 38.44 | 63.34 |
| -if | 52.68 | 3.66 | 4.28 | 219.26 | 51.06 | 4.12 | 1.78 | 26.36 | 17.08 | 17.28 | 41.38 | 46.22 | 104.88 |
| -none | 45.38 | 0.20 | 4.28 | 217.06 | 24.58 | 4.30 | 1.88 | 26.26 | 17.06 | 12.64 | 12.26 | 38.44 | 63.34 |
| -tests | 61.96 | 0.18 | 14.60 | 210.18 | 25.38 | 4.28 | 1.92 | 26.08 | 17.12 | 13.64 | 19.56 | 42.52 | 75.72 |
| -while | 53.32 | 0.18 | 44.64 | 210.04 | 24.80 | 4.18 | 1.76 | 25.66 | 17.08 | 16.40 | 12.00 | 52.04 | 80.44 |
| -xy | 45.56 | 0.16 | 4.30 | 219.42 | 25.18 | 17.04 | 1.92 | 26.20 | 17.30 | 12.72 | 12.26 | 38.60 | 63.58 |

| time | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 77.37 | 113.26 | 95.04 | 99.90 | 88.11 | 96.89 | 96.77 | 90.85 | 93.87 | 91.36 | 83.38 | 88.48 | 87.18 |
| +at | 99.29 | 100.00 | 99.15 | 92.90 | 104.58 | 87.95 | 97.85 | 96.68 | 93.35 | 100.92 | 101.81 | 96.85 | 99.22 |
| +class | 100.14 | 101.66 | 99.96 | 98.48 | 98.05 | 93.47 | 93.55 | 93.70 | 90.18 | 96.18 | 95.95 | 92.63 | 94.34 |
| +if | 89.17 | 4.97 | 66.16 | 96.86 | 57.06 | 90.06 | 95.70 | 92.48 | 89.88 | 69.97 | 48.06 | 74.68 | 64.74 |
| +none | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| +tests | 84.21 | 100.00 | 89.61 | 92.38 | 98.73 | 88.55 | 94.62 | 92.07 | 90.08 | 93.06 | 84.66 | 88.31 | 87.82 |
| +while | 85.47 | 100.55 | 23.57 | 92.68 | 99.06 | 89.76 | 94.62 | 93.97 | 90.18 | 72.66 | 96.51 | 65.15 | 77.17 |
| +xy | 98.98 | 101.66 | 99.32 | 94.69 | 103.12 | 26.20 | 97.85 | 95.60 | 93.15 | 100.42 | 100.72 | 96.56 | 98.61 |
| -arith | 159.89 | 90.00 | 303.27 | 96.71 | 140.68 | 99.53 | 100.00 | 105.79 | 100.70 | 118.83 | 176.84 | 117.01 | 128.95 |
| -at | 99.52 | 90.00 | 100.47 | 97.08 | 103.74 | 119.53 | 97.87 | 101.83 | 100.82 | 100.32 | 101.14 | 100.21 | 100.41 |
| -class | 99.96 | 90.00 | 99.53 | 100.01 | 100.08 | 100.00 | 100.00 | 99.77 | 100.35 | 99.84 | 100.16 | 100.00 | 100.00 |
| -if | 116.09 | 1830.00 | 100.00 | 101.01 | 207.73 | 95.81 | 94.68 | 100.38 | 100.12 | 136.71 | 337.52 | 120.24 | 165.58 |
| -none | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| -tests | 136.54 | 90.00 | 341.12 | 96.83 | 103.25 | 99.53 | 102.13 | 99.31 | 100.35 | 107.91 | 159.54 | 110.61 | 119.55 |
| -while | 117.50 | 90.00 | 1042.99 | 96.77 | 100.90 | 97.21 | 93.62 | 97.72 | 100.12 | 129.75 | 97.88 | 135.38 | 127.00 |
| -xy | 100.40 | 80.00 | 100.47 | 101.09 | 102.44 | 396.28 | 102.13 | 99.77 | 101.41 | 100.63 | 100.00 | 100.42 | 100.38 |

**Table B.1:** Benchmark execution times for the naïve code generator. The upper half of the table lists absolute time (in seconds) taken to complete the benchmarks in each category, against the various optimizations enabled ("+") and disabled ("-"). The lower half lists the performance relative to the 'base' cases, where '100%' is the performance with all optimizations disabled ('+none') or all enabled ('-none').

| time | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 53.30 | 3.58 | 72.12 | 136.60 | 47.70 | 15.12 | 1.72 | 25.36 | 17.70 | 23.82 | 39.96 | 67.58 | 131.36 |
| +at | 57.60 | 3.62 | 91.92 | 209.68 | 59.76 | 14.36 | 1.74 | 25.90 | 17.68 | 27.76 | 59.54 | 79.14 | 166.44 |
| +class | 56.90 | 3.54 | 91.84 | 200.88 | 60.42 | 15.40 | 1.72 | 19.50 | 17.80 | 27.66 | 58.66 | 79.18 | 165.50 |
| +if | 45.42 | 0.18 | 60.98 | 209.12 | 33.78 | 14.88 | 1.72 | 25.40 | 17.54 | 19.82 | 29.26 | 61.64 | 110.72 |
| +none | 57.56 | 3.50 | 92.12 | 208.06 | 60.38 | 15.18 | 1.76 | 25.44 | 17.82 | 27.72 | 59.34 | 79.26 | 166.32 |
| +tests | 57.56 | 3.50 | 92.12 | 208.00 | 60.40 | 15.18 | 1.80 | 25.44 | 17.82 | 27.72 | 59.38 | 79.22 | 166.32 |
| +while | 42.90 | 3.56 | 20.74 | 201.08 | 58.54 | 14.94 | 1.72 | 25.60 | 17.46 | 20.64 | 58.38 | 55.40 | 134.42 |
| +xy | 57.94 | 3.58 | 92.28 | 208.82 | 59.56 | 4.32 | 1.78 | 24.90 | 17.40 | 27.36 | 58.58 | 77.78 | 163.72 |
| -arith | 33.62 | 0.18 | 11.96 | 201.70 | 34.04 | 3.54 | 1.84 | 19.60 | 17.12 | 14.20 | 20.14 | 40.80 | 75.14 |
| -at | 32.84 | 0.20 | 1.64 | 138.00 | 23.60 | 4.50 | 1.86 | 19.10 | 17.16 | 11.38 | 8.82 | 33.08 | 53.28 |
| -class | 33.34 | 0.20 | 1.64 | 140.62 | 21.76 | 3.50 | 1.86 | 25.72 | 17.00 | 11.42 | 9.22 | 33.06 | 53.70 |
| -if | 38.88 | 3.56 | 1.62 | 136.56 | 47.70 | 3.38 | 1.74 | 19.06 | 16.98 | 16.60 | 38.76 | 42.46 | 97.82 |
| -none | 33.82 | 0.22 | 1.70 | 140.32 | 24.48 | 3.68 | 1.94 | 20.16 | 17.64 | 12.24 | 9.70 | 36.84 | 58.78 |
| -tests | 34.94 | 0.22 | 12.02 | 137.00 | 22.24 | 3.52 | 1.86 | 19.48 | 16.96 | 13.00 | 17.58 | 39.38 | 69.96 |
| -while | 40.94 | 0.20 | 41.32 | 136.72 | 22.66 | 3.40 | 1.74 | 19.28 | 16.98 | 15.48 | 8.74 | 46.92 | 71.14 |
| -xy | 32.84 | 0.20 | 1.62 | 139.10 | 23.74 | 14.10 | 1.86 | 19.20 | 17.14 | 11.42 | 8.72 | 33.18 | 53.32 |

| time | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 92.60 | 102.29 | 78.29 | 65.65 | 79.00 | 99.60 | 97.73 | 99.69 | 99.33 | 85.93 | 67.34 | 85.26 | 78.98 |
| +at | 100.07 | 103.43 | 99.78 | 100.78 | 98.97 | 94.60 | 98.86 | 101.81 | 99.21 | 100.14 | 100.34 | 99.85 | 100.07 |
| +class | 98.85 | 101.14 | 99.70 | 96.55 | 100.07 | 101.45 | 97.73 | 76.65 | 99.89 | 99.78 | 98.85 | 99.90 | 99.51 |
| +if | 78.91 | 5.14 | 66.20 | 100.51 | 55.95 | 98.02 | 97.73 | 99.84 | 98.43 | 71.50 | 49.31 | 77.77 | 66.57 |
| +none | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| +tests | 100.00 | 100.00 | 100.00 | 99.97 | 100.03 | 100.00 | 102.27 | 100.00 | 100.00 | 100.00 | 100.07 | 99.95 | 100.00 |
| +while | 74.53 | 101.71 | 22.51 | 96.65 | 96.95 | 98.42 | 97.73 | 100.63 | 97.98 | 74.46 | 98.38 | 69.90 | 80.82 |
| +xy | 100.66 | 102.29 | 100.17 | 100.37 | 98.64 | 28.46 | 101.14 | 97.88 | 97.64 | 98.70 | 98.72 | 98.13 | 98.44 |
| -arith | 99.41 | 81.82 | 703.53 | 143.74 | 139.05 | 96.20 | 94.85 | 97.22 | 97.05 | 116.01 | 207.63 | 110.75 | 127.83 |
| -at | 97.10 | 90.91 | 96.47 | 98.35 | 96.41 | 122.28 | 95.88 | 94.74 | 97.28 | 92.97 | 90.93 | 89.79 | 90.64 |
| -class | 98.58 | 90.91 | 96.47 | 100.21 | 88.89 | 95.11 | 95.88 | 127.58 | 96.37 | 93.30 | 95.05 | 89.74 | 91.36 |
| -if | 114.96 | 1618.18 | 95.29 | 97.32 | 194.85 | 91.85 | 89.69 | 94.54 | 96.26 | 135.62 | 399.59 | 115.26 | 166.42 |
| -none | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| -tests | 103.31 | 100.00 | 707.06 | 97.63 | 90.85 | 95.65 | 95.88 | 96.63 | 96.15 | 106.21 | 181.24 | 106.89 | 119.02 |
| -while | 121.05 | 90.91 | 2430.59 | 97.43 | 92.57 | 92.39 | 89.69 | 95.63 | 96.26 | 126.47 | 90.10 | 127.36 | 121.03 |
| -xy | 97.10 | 90.91 | 95.29 | 99.13 | 96.98 | 383.15 | 95.88 | 95.24 | 97.17 | 93.30 | 89.90 | 90.07 | 90.71 |

**Table B.2:** Benchmark execution times for the delayed code generator. As in table B, the upper half lists absolute performance and the lower half relative performance.

| time | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 96.48 | 8.28 | 531.94 | 129.68 | 132.42 | 14.28 | 42.58 | 20.58 | 26.80 | 38.78 | 136.60 | 160.36 | 335.74 |
| +at | 102.60 | 8.14 | 611.06 | 127.68 | 140.18 | 50.32 | 41.70 | 21.28 | 26.28 | 39.58 | 140.42 | 161.50 | 341.50 |
| +class | 107.72 | 8.22 | 543.10 | 124.26 | 135.28 | 14.34 | 42.14 | 21.12 | 26.50 | 38.94 | 138.38 | 161.78 | 339.10 |
| +if | 62.54 | 5.38 | 354.00 | 129.88 | 17.22 | 15.30 | 40.30 | 23.50 | 29.66 | 27.90 | 151.26 | 97.48 | 276.64 |
| +none | 103.12 | 8.20 | 593.20 | 123.66 | 136.92 | 14.48 | 41.92 | 21.32 | 26.86 | 39.50 | 140.68 | 157.98 | 338.16 |
| +tests | 88.78 | 8.26 | 528.00 | 59.44 | 132.16 | 14.36 | 41.70 | 21.44 | 26.82 | 38.70 | 136.92 | 159.96 | 335.58 |
| +while | 52.20 | 7.64 | 16.22 | 112.40 | 134.24 | 11.84 | 34.68 | 18.60 | 26.46 | 27.44 | 145.22 | 79.82 | 252.48 |
| +xy | 107.78 | 8.22 | 539.80 | 125.74 | 137.24 | 13.10 | 45.20 | 21.18 | 26.82 | 39.34 | 140.24 | 162.76 | 342.34 |
| -arith | 27.20 | 3.98 | 12.22 | 59.72 | 12.84 | 40.90 | 34.70 | 19.10 | 25.94 | 22.34 | 141.36 | 67.06 | 230.76 |
| -at | 16.22 | 4.02 | 8.26 | 60.34 | 7.12 | 10.92 | 34.90 | 18.20 | 26.38 | 21.72 | 137.88 | 65.48 | 225.08 |
| -class | 16.08 | 4.02 | 8.68 | 58.26 | 7.40 | 40.86 | 34.66 | 18.06 | 25.90 | 21.74 | 137.62 | 65.34 | 224.70 |
| -if | 27.94 | 7.58 | 8.34 | 60.96 | 122.72 | 40.84 | 35.08 | 17.98 | 25.94 | 25.94 | 136.88 | 76.42 | 239.24 |
| -none | 16.26 | 4.04 | 8.18 | 58.94 | 7.46 | 41.48 | 35.32 | 18.42 | 25.86 | 21.76 | 137.60 | 65.38 | 224.74 |
| -tests | 30.32 | 4.08 | 11.92 | 112.16 | 12.54 | 41.26 | 35.02 | 18.30 | 25.88 | 22.54 | 141.22 | 67.24 | 231.00 |
| -while | 34.26 | 5.26 | 342.30 | 60.22 | 6.88 | 44.78 | 37.42 | 20.82 | 25.94 | 25.78 | 134.62 | 86.52 | 246.92 |
| -xy | 16.28 | 4.02 | 8.96 | 58.76 | 7.40 | 46.78 | 39.42 | 18.26 | 25.94 | 21.56 | 137.60 | 65.22 | 224.38 |

| time | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 93.56 | 100.98 | 89.67 | 104.87 | 96.71 | 98.62 | 101.57 | 96.53 | 99.78 | 98.18 | 97.10 | 101.51 | 99.28 |
| +at | 99.50 | 99.27 | 103.01 | 103.25 | 102.38 | 347.51 | 99.48 | 99.81 | 97.84 | 100.20 | 99.82 | 102.23 | 100.99 |
| +class | 104.46 | 100.24 | 91.55 | 100.49 | 98.80 | 99.03 | 100.52 | 99.06 | 98.66 | 98.58 | 98.37 | 102.41 | 100.28 |
| +if | 60.65 | 65.61 | 59.68 | 105.03 | 12.58 | 105.66 | 96.14 | 110.23 | 110.42 | 70.63 | 107.52 | 61.70 | 81.81 |
| +none | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| +tests | 86.09 | 100.73 | 89.01 | 48.07 | 96.52 | 99.17 | 99.48 | 100.56 | 99.85 | 97.97 | 97.33 | 101.25 | 99.24 |
| +while | 50.62 | 93.17 | 2.73 | 90.89 | 98.04 | 81.77 | 82.73 | 87.24 | 98.51 | 69.47 | 103.23 | 50.53 | 74.66 |
| +xy | 104.52 | 100.24 | 91.00 | 101.68 | 100.23 | 90.47 | 107.82 | 99.34 | 99.85 | 99.59 | 99.69 | 103.03 | 101.24 |
| -arith | 167.28 | 98.51 | 149.39 | 101.32 | 172.12 | 98.60 | 98.24 | 103.69 | 100.31 | 102.67 | 102.73 | 102.57 | 102.68 |
| -at | 99.75 | 99.50 | 100.98 | 102.38 | 95.44 | 26.33 | 98.81 | 98.81 | 102.01 | 99.82 | 100.20 | 100.15 | 100.15 |
| -class | 98.89 | 99.50 | 106.11 | 98.85 | 99.20 | 98.51 | 98.13 | 98.05 | 100.15 | 99.91 | 100.01 | 99.94 | 99.98 |
| -if | 171.83 | 187.62 | 101.96 | 103.43 | 1645.04 | 98.46 | 99.32 | 97.61 | 100.31 | 119.21 | 99.48 | 116.89 | 106.45 |
| -none | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| -tests | 186.47 | 100.99 | 145.72 | 190.30 | 168.10 | 99.47 | 99.15 | 99.35 | 100.08 | 103.58 | 102.63 | 102.84 | 102.79 |
| -while | 210.70 | 130.20 | 4184.60 | 102.17 | 92.23 | 107.96 | 105.95 | 113.03 | 100.31 | 118.47 | 97.83 | 132.33 | 109.87 |
| -xy | 100.12 | 99.50 | 109.54 | 99.69 | 99.20 | 112.78 | 111.61 | 99.13 | 100.31 | 99.08 | 100.00 | 99.76 | 99.84 |

**Table B.3:** Benchmark execution times for PS2.3. Again, the upper half lists absolute performance and the lower half relative performance. The timings were all made with the same set of optional primitives and BitBlt copyBits operation as were present in the native code runtime system.

| naïve | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 4395782 | 152036 | 3701066 | 16490186 | 2293998 | 1202036 | 57018 | 2614128 | 65181 | 986816 | 1910516 | 2589711 | 5487043 |
| +at | 6655882 | 156036 | 4301186 | 17260186 | 2949358 | 1168036 | 58018 | 2876128 | 64606 | 1155432 | 2606876 | 3083025 | 6845333 |
| +class | 6655882 | 156036 | 4301186 | 17260186 | 2949358 | 1223036 | 58018 | 2876128 | 73576 | 1155432 | 2606876 | 3086602 | 6848910 |
| +if | 5930662 | 24028 | 3100898 | 15600134 | 1966254 | 1181028 | 56014 | 2512096 | 71543 | 887453 | 1651748 | 2389726 | 4928927 |
| +none | 6655882 | 156036 | 4301186 | 17260186 | 2949358 | 1223036 | 58018 | 2876128 | 73576 | 1155432 | 2606876 | 3086602 | 6848910 |
| +tests | 5365764 | 152028 | 3701008 | 16370146 | 2949314 | 1202028 | 57014 | 2674100 | 72318 | 1092837 | 2129308 | 2821694 | 6043839 |
| +while | 5395358 | 136024 | 1200434 | 13050144 | 2949152 | 1118024 | 53012 | 1946092 | 70899 | 869192 | 2606464 | 2026737 | 5502393 |
| +xy | 6655882 | 156036 | 4301186 | 17260186 | 2949358 | 223036 | 58018 | 2876128 | 56776 | 1155432 | 2606876 | 3077301 | 6839609 |
| -arith | 3880228 | 8012 | 600252 | 12160066 | 1966086 | 42012 | 52006 | 1744044 | 42897 | 652070 | 1173932 | 1468807 | 3294809 |
| -at | 1620128 | 4012 | 132 | 11390066 | 1310726 | 76012 | 51006 | 1482044 | 43472 | 483454 | 477572 | 975143 | 1936169 |
| -class | 1620128 | 4012 | 132 | 11390066 | 1310726 | 21012 | 51006 | 1482044 | 34502 | 483454 | 477572 | 971566 | 1932592 |
| -if | 1845140 | 128016 | 136 | 11390104 | 2293748 | 21016 | 51008 | 1482064 | 35476 | 637981 | 1432536 | 1251210 | 3321727 |
| -none | 1620128 | 4012 | 132 | 11390066 | 1310726 | 21012 | 51006 | 1482044 | 34502 | 483454 | 477572 | 971566 | 1932592 |
| -tests | 2910246 | 8020 | 600310 | 12280106 | 1310770 | 42020 | 52010 | 1684072 | 35760 | 546049 | 955140 | 1237399 | 2738588 |
| -while | 2380444 | 16020 | 1900600 | 13940094 | 1310850 | 84020 | 54010 | 2048068 | 36120 | 656242 | 477820 | 1614424 | 2748486 |
| -xy | 1620128 | 4012 | 132 | 11390066 | 1310726 | 1021012 | 51006 | 1482044 | 51302 | 483454 | 477572 | 980867 | 1941893 |

| dcg | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 3105664 | 148028 | 3100888 | 10100146 | 2293954 | 1181028 | 56014 | 2412100 | 63923 | 944451 | 1432948 | 2295538 | 4672937 |
| +at | 3655882 | 156036 | 4301186 | 17260186 | 2949358 | 1168036 | 58018 | 2776128 | 64606 | 1191802 | 2606876 | 3053213 | 6851891 |
| +class | 3655882 | 156036 | 4301186 | 17260186 | 2949358 | 1223036 | 58018 | 2376128 | 73566 | 1190777 | 2606876 | 3053070 | 6850723 |
| +if | 2930662 | 24028 | 3100898 | 15600134 | 1966254 | 1181028 | 56014 | 2412096 | 71543 | 913853 | 1651748 | 2364694 | 4930295 |
| +none | 3655882 | 156036 | 4301186 | 17260186 | 2949358 | 1223036 | 58018 | 2776128 | 73576 | 1191802 | 2606876 | 3056790 | 6855468 |
| +tests | 3655882 | 156036 | 4301186 | 17260186 | 2949358 | 1223036 | 58018 | 2776128 | 73576 | 1191802 | 2606876 | 3056790 | 6855468 |
| +while | 2395358 | 136024 | 1200434 | 13050144 | 2949152 | 1118024 | 53012 | 1846092 | 70899 | 893202 | 2606464 | 1995925 | 5495591 |
| +xy | 3655882 | 156036 | 4301186 | 17260186 | 2949358 | 223036 | 58018 | 2776128 | 56776 | 1191802 | 2606876 | 3047489 | 6846167 |
| -arith | 1885228 | 8012 | 600252 | 12160066 | 1966086 | 42012 | 52006 | 1244044 | 43532 | 645490 | 1173932 | 1336263 | 3155685 |
| -at | 1620128 | 4012 | 132 | 5890066 | 1310726 | 76012 | 51006 | 1082044 | 43457 | 469254 | 477572 | 839208 | 1786034 |
| -class | 1620128 | 4012 | 132 | 5890066 | 1310726 | 21012 | 51006 | 1482044 | 34497 | 470279 | 477572 | 839351 | 1787202 |
| -if | 1845140 | 128016 | 136 | 5890104 | 2293748 | 21016 | 51008 | 1082064 | 35466 | 643216 | 1432536 | 1216786 | 3292538 |
| -none | 1620128 | 4012 | 132 | 5890066 | 1310726 | 21012 | 51006 | 1082044 | 34487 | 469254 | 477572 | 835631 | 1782457 |
| -tests | 1905246 | 8020 | 600310 | 6780106 | 1310770 | 42020 | 52010 | 1284072 | 35100 | 556334 | 955140 | 1204551 | 2716025 |
| -while | 2380444 | 16020 | 1900600 | 8440094 | 1310850 | 84020 | 54010 | 1648068 | 36105 | 651122 | 477820 | 1479683 | 2608625 |
| -xy | 1620128 | 4012 | 132 | 5890066 | 1310726 | 1021012 | 51006 | 1082044 | 51287 | 469254 | 477572 | 844932 | 1791758 |

**Table B.4:** The number of full message sends performed by the benchmarks compiled using the naïve and delayed code generators.

| naïve | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 3630530 | 69012 | 2500692 | 14590078 | 983226 | 160012 | 55006 | 1970048 | 20012 | 602641 | 955372 | 1619207 | 3177220 |
| +at | 5890630 | 73012 | 3100812 | 15360078 | 1638586 | 126012 | 56006 | 2232048 | 19437 | 771257 | 1651732 | 2094570 | 4517559 |
| +class | 5890630 | 73012 | 3100812 | 15360078 | 1638586 | 181012 | 56006 | 2232048 | 28407 | 771257 | 1651732 | 2097038 | 4520027 |
| +if | 5605528 | 20012 | 2500702 | 14590066 | 1310866 | 160012 | 55006 | 2070044 | 27507 | 647337 | 1174172 | 1832881 | 3654390 |
| +none | 5890630 | 73012 | 3100812 | 15360078 | 1638586 | 181012 | 56006 | 2232048 | 28407 | 771257 | 1651732 | 2097038 | 4520027 |
| +tests | 4600512 | 69004 | 2500634 | 14470038 | 1638542 | 160004 | 55002 | 2030020 | 27149 | 708662 | 1174164 | 1840583 | 3723409 |
| +while | 5150322 | 61008 | 1200418 | 12930064 | 1638464 | 118008 | 53004 | 1706036 | 26848 | 603689 | 1651488 | 1507570 | 3762747 |
| +xy | 5890630 | 73012 | 3100812 | 15360078 | 1638586 | 181012 | 56006 | 2232048 | 28407 | 771257 | 1651732 | 2097038 | 4520027 |
| -arith | 3815202 | 8000 | 600240 | 12040012 | 1310740 | 42000 | 52000 | 1504004 | 16220 | 471290 | 696440 | 1169829 | 2337559 |
| -at | 1555102 | 4000 | 120 | 11270012 | 655380 | 76000 | 51000 | 1242004 | 16795 | 302674 | 80 | 694116 | 996870 |
| -class | 1555102 | 4000 | 120 | 11270012 | 655380 | 21000 | 51000 | 1242004 | 7825 | 302674 | 80 | 691648 | 994402 |
| -if | 1600104 | 53000 | 120 | 11270024 | 983060 | 21000 | 51000 | 1242008 | 8225 | 372478 | 477560 | 770166 | 1620204 |
| -none | 1555102 | 4000 | 120 | 11270012 | 655380 | 21000 | 51000 | 1242004 | 7825 | 302674 | 80 | 691648 | 994402 |
| -tests | 2845220 | 8008 | 600298 | 12160052 | 655424 | 42008 | 52004 | 1444032 | 9083 | 365269 | 477648 | 948778 | 1791695 |
| -while | 2055310 | 12004 | 1300404 | 12930026 | 655462 | 63004 | 53002 | 1606016 | 8884 | 416126 | 244 | 1095652 | 1512022 |
| -xy | 1555102 | 4000 | 120 | 11270012 | 655380 | 21000 | 51000 | 1242004 | 7825 | 302674 | 80 | 691648 | 994402 |

| dcg | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 2340412 | 65004 | 1900514 | 8200038 | 983182 | 139004 | 54002 | 1768020 | 18754 | 548746 | 477804 | 1349847 | 2376397 |
| +at | 2890630 | 73012 | 3100812 | 15360078 | 1638586 | 126012 | 56006 | 2132048 | 19437 | 796097 | 1651732 | 2080028 | 4527857 |
| +class | 2890630 | 73012 | 3100812 | 15360078 | 1638586 | 181012 | 56006 | 1732048 | 28397 | 795072 | 1651732 | 2079216 | 4526020 |
| +if | 2605528 | 20012 | 2500702 | 14590066 | 1310866 | 160012 | 55006 | 1970044 | 27507 | 668207 | 1174172 | 1819664 | 3662043 |
| +none | 2890630 | 73012 | 3100812 | 15360078 | 1638586 | 181012 | 56006 | 2132048 | 28407 | 796097 | 1651732 | 2082496 | 4530325 |
| +tests | 2890630 | 73012 | 3100812 | 15360078 | 1638586 | 181012 | 56006 | 2132048 | 28407 | 796097 | 1651732 | 2082496 | 4530325 |
| +while | 2150322 | 61008 | 1200418 | 12930064 | 1638464 | 118008 | 53004 | 1606036 | 26848 | 621329 | 1651488 | 1493348 | 3766165 |
| +xy | 2890630 | 73012 | 3100812 | 15360078 | 1638586 | 181012 | 56006 | 2132048 | 28407 | 796097 | 1651732 | 2082496 | 4530325 |
| -arith | 1820202 | 8000 | 600240 | 12040012 | 1310740 | 42000 | 52000 | 1004004 | 16855 | 461760 | 696440 | 1110157 | 2268357 |
| -at | 1555102 | 4000 | 120 | 5770012 | 655380 | 76000 | 51000 | 842004 | 16780 | 285524 | 80 | 632285 | 917889 |
| -class | 1555102 | 4000 | 120 | 5770012 | 655380 | 21000 | 51000 | 1242004 | 7820 | 286549 | 80 | 633097 | 919726 |
| -if | 1600104 | 53000 | 120 | 5770024 | 983060 | 21000 | 51000 | 842008 | 8215 | 371343 | 477560 | 754137 | 1603040 |
| -none | 1555102 | 4000 | 120 | 5770012 | 655380 | 21000 | 51000 | 842004 | 7810 | 285524 | 80 | 629817 | 915421 |
| -tests | 1840220 | 8008 | 600298 | 6660052 | 655424 | 42008 | 52004 | 1044032 | 8423 | 372604 | 477648 | 930267 | 1780519 |
| -while | 2055310 | 12004 | 1300404 | 7430026 | 655462 | 63004 | 53002 | 1206016 | 8869 | 405476 | 244 | 1033890 | 1439610 |
| -xy | 1555102 | 4000 | 120 | 5770012 | 655380 | 21000 | 51000 | 842004 | 7810 | 285524 | 80 | 629817 | 915421 |

**Table B.5:** The number of primitive calls performed by the benchmarks compiled using the naïve and delayed code generators.

| oops | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 272045 | 955058 | 582793 | 1809896 |
| +at | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 272045 | 955058 | 582793 | 1809896 |
| +class | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 272045 | 955058 | 582793 | 1809896 |
| +if | 50022 | 10 | 140 | 240047 | 5 | 55010 | 50005 | 80034 | 13520 | 68677 | 10 | 102601 | 171288 |
| +none | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 272045 | 955058 | 582793 | 1809896 |
| +tests | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 272045 | 955058 | 583068 | 1810171 |
| +while | 195016 | 100006 | 6 | 45 | 655343 | 55006 | 50003 | 26 | 14351 | 201509 | 954966 | 281634 | 1438109 |
| +xy | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 272045 | 955058 | 582793 | 1809896 |
| -arith | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 57477 | 2 | 26768 | 84247 |
| -at | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 57477 | 2 | 26768 | 84247 |
| -class | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 57477 | 2 | 26768 | 84247 |
| -if | 195016 | 100006 | 6 | 45 | 655343 | 55006 | 50003 | 26 | 14351 | 201509 | 954966 | 281809 | 1438284 |
| -none | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 57477 | 2 | 26768 | 84247 |
| -tests | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 57477 | 2 | 26693 | 84172 |
| -while | 50022 | 10 | 140 | 240047 | 5 | 55010 | 50005 | 80034 | 13520 | 68677 | 10 | 102676 | 171363 |
| -xy | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 57477 | 2 | 26768 | 84247 |

| bytes | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6239152 | 22921392 | 14786436 | 43946980 |
| +at | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6239152 | 22921392 | 14786436 | 43946980 |
| +class | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6239152 | 22921392 | 14786436 | 43946980 |
| +if | 1200528 | 240 | 3360 | 5761128 | 120 | 880240 | 800120 | 1920816 | 302220 | 1358320 | 240 | 3261828 | 4620388 |
| +none | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6239152 | 22921392 | 14786436 | 43946980 |
| +tests | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6239152 | 22921392 | 14793436 | 43953980 |
| +while | 4680384 | 2400144 | 144 | 1080 | 15728232 | 880144 | 800072 | 624 | 322164 | 4546288 | 22919184 | 7558620 | 35024092 |
| +xy | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6239152 | 22921392 | 14786436 | 43946980 |
| -arith | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1089520 | 48 | 1442236 | 2531804 |
| -at | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1089520 | 48 | 1442236 | 2531804 |
| -class | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1089520 | 48 | 1442236 | 2531804 |
| -if | 4680384 | 2400144 | 144 | 1080 | 15728232 | 880144 | 800072 | 624 | 322164 | 4546288 | 22919184 | 7563220 | 35028692 |
| -none | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1089520 | 48 | 1442236 | 2531804 |
| -tests | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1089520 | 48 | 1440036 | 2529604 |
| -while | 1200528 | 240 | 3360 | 5761128 | 120 | 880240 | 800120 | 1920816 | 302220 | 1358320 | 240 | 3264028 | 4622588 |
| -xy | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1089520 | 48 | 1442236 | 2531804 |

**Table B.6:** Memory utilization for the naïve code generator. The upper half of the table shows the number of object pointers allocated for the benchmarks in each class for each particular optimization. The lower half shows the total number of bytes allocated.

| oops | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 280525 | 955058 | 572083 | 1807666 |
| +at | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 280525 | 955058 | 571808 | 1807391 |
| +class | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 280525 | 955058 | 571808 | 1807391 |
| +if | 50022 | 10 | 140 | 240047 | 5 | 55010 | 50005 | 80034 | 13520 | 70517 | 10 | 97051 | 167578 |
| +none | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 280525 | 955058 | 571808 | 1807391 |
| +tests | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 280525 | 955058 | 571808 | 1807391 |
| +while | 195016 | 100006 | 6 | 45 | 655343 | 55006 | 50003 | 26 | 14351 | 206849 | 954966 | 272789 | 1434604 |
| +xy | 505140 | 104018 | 600318 | 1130087 | 655389 | 76018 | 51009 | 282062 | 15028 | 280525 | 955058 | 571808 | 1807391 |
| -arith | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 58757 | 2 | 23093 | 81852 |
| -at | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 58757 | 2 | 23168 | 81927 |
| -class | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 58757 | 2 | 23168 | 81927 |
| -if | 195016 | 100006 | 6 | 45 | 655343 | 55006 | 50003 | 26 | 14351 | 206849 | 954966 | 272964 | 1434779 |
| -none | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 58757 | 2 | 23168 | 81927 |
| -tests | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 58757 | 2 | 23168 | 81927 |
| -while | 50022 | 10 | 140 | 240047 | 5 | 55010 | 50005 | 80034 | 13520 | 70517 | 10 | 97126 | 167653 |
| -xy | 6 | 2 | 2 | 19 | 1 | 55002 | 50001 | 10 | 13402 | 58757 | 2 | 23168 | 81927 |

| bytes | arith | cond | loop | access | activ | point | mem | nolook | gfx | struct | num | iface | macro+= |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| +arith | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6443592 | 22921392 | 14471516 | 43836500 |
| +at | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6443592 | 22921392 | 14464516 | 43829500 |
| +class | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6443592 | 22921392 | 14464516 | 43829500 |
| +if | 1200528 | 240 | 3360 | 5761128 | 120 | 880240 | 800120 | 1920816 | 302220 | 1403400 | 240 | 3070348 | 4473988 |
| +none | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6443592 | 22921392 | 14464516 | 43829500 |
| +tests | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6443592 | 22921392 | 14464516 | 43829500 |
| +while | 4680384 | 2400144 | 144 | 1080 | 15728232 | 880144 | 800072 | 624 | 322164 | 4675368 | 22919184 | 7288060 | 34882612 |
| +xy | 12123360 | 2496432 | 14407632 | 27122088 | 15729336 | 1384432 | 824216 | 6769488 | 338412 | 6443592 | 22921392 | 14464516 | 43829500 |
| -arith | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1121160 | 48 | 1295356 | 2416564 |
| -at | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1121160 | 48 | 1297556 | 2418764 |
| -class | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1121160 | 48 | 1297556 | 2418764 |
| -if | 4680384 | 2400144 | 144 | 1080 | 15728232 | 880144 | 800072 | 624 | 322164 | 4675368 | 22919184 | 7292660 | 34887212 |
| -none | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1121160 | 48 | 1297556 | 2418764 |
| -tests | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1121160 | 48 | 1297556 | 2418764 |
| -while | 1200528 | 240 | 3360 | 5761128 | 120 | 880240 | 800120 | 1920816 | 302220 | 1403400 | 240 | 3072548 | 4476188 |
| -xy | 144 | 48 | 48 | 456 | 24 | 880048 | 800024 | 240 | 299388 | 1121160 | 48 | 1297556 | 2418764 |

**Table B.7:** Memory utilization for the delayed code generator. The upper half of the table shows the number of object pointers allocated for the benchmarks in each class for each particular optimization. The lower half shows the total number of bytes allocated.

## B.1 Unexpected Results

Perhaps the most surprising result is the effect of enabling a special send of the '@' message in PS2.3, where the performance is actually degraded.

A *small* degradation of performance would make sense for a bytecode interpreter, which would be required to perform an extra indirection during the conversion process between selector index (in the bytecode) into an object pointer (from the 'SpecialSelectors' array) for use in the dynamic bind. However, it is much harder to imagine why the effect is so pronounced, since secondary compilation of bytecoded methods into n-code should produce identical results for both a full send and a special send of '@' — especially considering that the justification for its special send status is simply to save valuable literal frame slots.

## B.2 Absolute Performance of the DCG Compiler

Although a comparison of the absolute performance of PS2.3 and Native Code Smalltalk-80 is not that relevant, table B.8 has been included — for entertainment purposes only!

| | PS2.3 | dcg | PS2.3 ÷ dcg |
|---|---|---|---|
| unoptimized execution time | 338.16 | 166.32 | 2.03 |
| optimized execution time | 224.74 | 58.78 | 3.82 |
| unoptimized ÷ optimized | 1.50 | 2.83 | |

**Table B.8:** The performance of PS2.3 and the delayed code generator version of Native Code Smalltalk-80.

# Appendix C

# Assembly Language Conventions

The following tables give the notation used by both Sun and Mo- torola for the addressing modes used in this thesis. Full details of the addressing capabilities of the MC68020 and their notations can be found in [Sun88] and [Mot85].

| | Notation | |
|---|---|---|
| Meaning | Sun | Motorola |
| Address Register | a*n* | An |
| Data Register | d*n* | Dn |
| Index (Data or Address) Register | r*i* | Xn |
| Constant Displacement | *d* | d or od |
| Immediate Data | xxx | xxx |

| | Notation | |
|---|---|---|
| Mode | Sun | Motorola |
| Data Register Direct | d*n* | Dn |
| Address Register Direct | a*n* | An |
| Address Register Indirect | a*n*@ | (An) |
| Address Register Indirect with Displacement | a*n*@(*d*) | (d,An) |
| Address Register Indirect with Index | a*n*@(*d*,r*i*) | (An,Xn) |
| Address Register Indirect with Postincrement | an@+ | (An)+ |
| Address Register Indirect with Predecrement | an@- | -(An) |
| Memory-Indirect Pre-Indexed | a*n*@(r*i*)@(*d*) | ([An,Xn],od) |
| Immediate | #xxx | #xxx |

# Appendix D

# Examples

This appendix contains several examples to illustrate delayed code generation for an entire Smalltalk method, and for some small expressions in C. Section 6.6 describes in detail the techniques used.

## D.1 Conditional Statement

Figure D.1 shows the parse tree, operand descriptors, and generated code for the statement 'if(a < b) a= b;'. Code for the comparison is generated first, taking into account the addressability constraints on the operands of the 'cmpl' instruction, resulting in $[]^{lt}$ (no physical value or size). The 'if' node generates a jump instruction on the inverse of this condition, 'jge failLabel', to skip the body of the 'then' clause if the condition is not satisfied. Finally the code for the assignment and a label for the jump destination are generated, before returning a null operand descriptor [] as the result. ('if' statements have no overall value in C.)

## D.2 String Copy Loop

Figure D.2 shows the parse tree, operand descriptors, and generated code for the statement 'while(*to++= *from++);'. This example assumes that both 'from' and 'to' are variables of type 'register char *'.

Firstly the 'while' node generates a label to mark the beginning of the test, which consists of an assignment statement. Neither of the two pointer dereferencing operations produce any code, since they translate naturally into an addressing mode supported directly by the hardware. It is only when the assignment finally takes place that a 'movb' instruction (taking account of the size of the source and destination) is generated. The result of the assignment is the 'fixed' version of one of the operands. Since fixing an operand does not affect the condition codes register, the result is still tagged with the 'ne' condition implied by the move instruction.

The 'while' node now performs an optimization: since the body of the loop is null it can generate a jump back to the start of the test directly, using the 'ne' condition

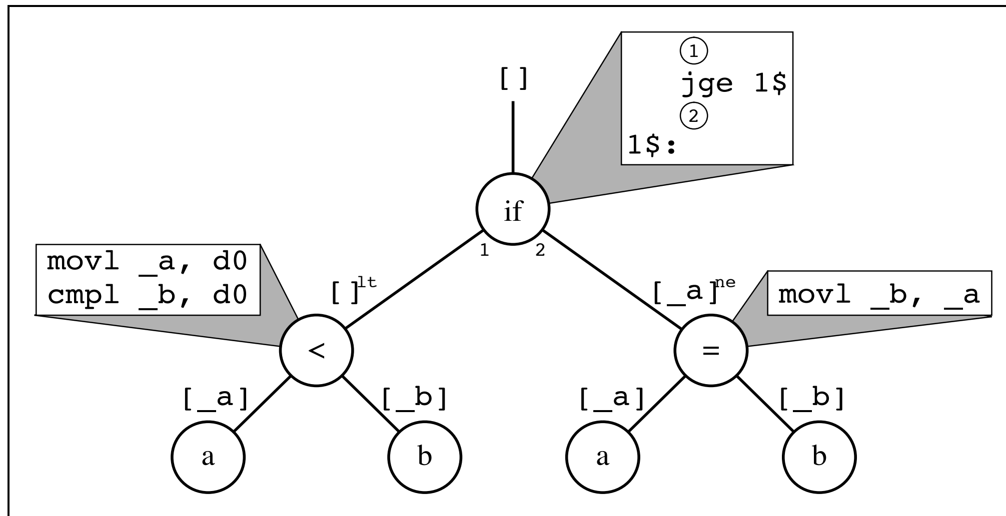


**Figure D.1:** Parse tree, operand descriptors and generated code for ‘`if(a < b) a= b;`’

---

synthesized in the assignment. As in the previous example, the overall result is the null descriptor, [].

## D.3 Function Call

Figure D.3 shows the parse tree, operand descriptors, and generated code for the function call ‘`fread(buf, sizeof int, a*a + b*b, fp);`’.

The function call node (‘`()`’) first evaluates the name of the function, which yields $[\texttt{\#_fread}]^{t}_{4}$. Next the four arguments are processed left-to-right, pushing the resulting operand descriptors onto a compile-time stack. Once the last has been dealt with, they are popped off this stack and an instruction generated to move them onto the run-time stack thereby pushing the required arguments in reverse order. In the case of the two constants, $[\texttt{\#4}]^{t}_{4}$ and $[\texttt{\#_buf}]^{t}_{4}$, the push is performed by a ‘push effective address’ instruction which is shorter and faster than the equivalent move instruction. This transformation is carried out by the ‘M68000>>emitPush:’ method which checks for a literal argument, and uses a ‘`pea`’ if possible. The function call is completed by generating a ‘`jbsr`’ instruction, and then the stack pointer incremented over the arguments (which were counted as they were evaluated).

The return value from a function call, in the absence of any interprocedural optimizations, always comes back in a standard place – in the case of C it is usually `d0`. The size of the resulting operand descriptor will depend on the type of the function’s value, and the state of the condition codes will be implementation dependent.

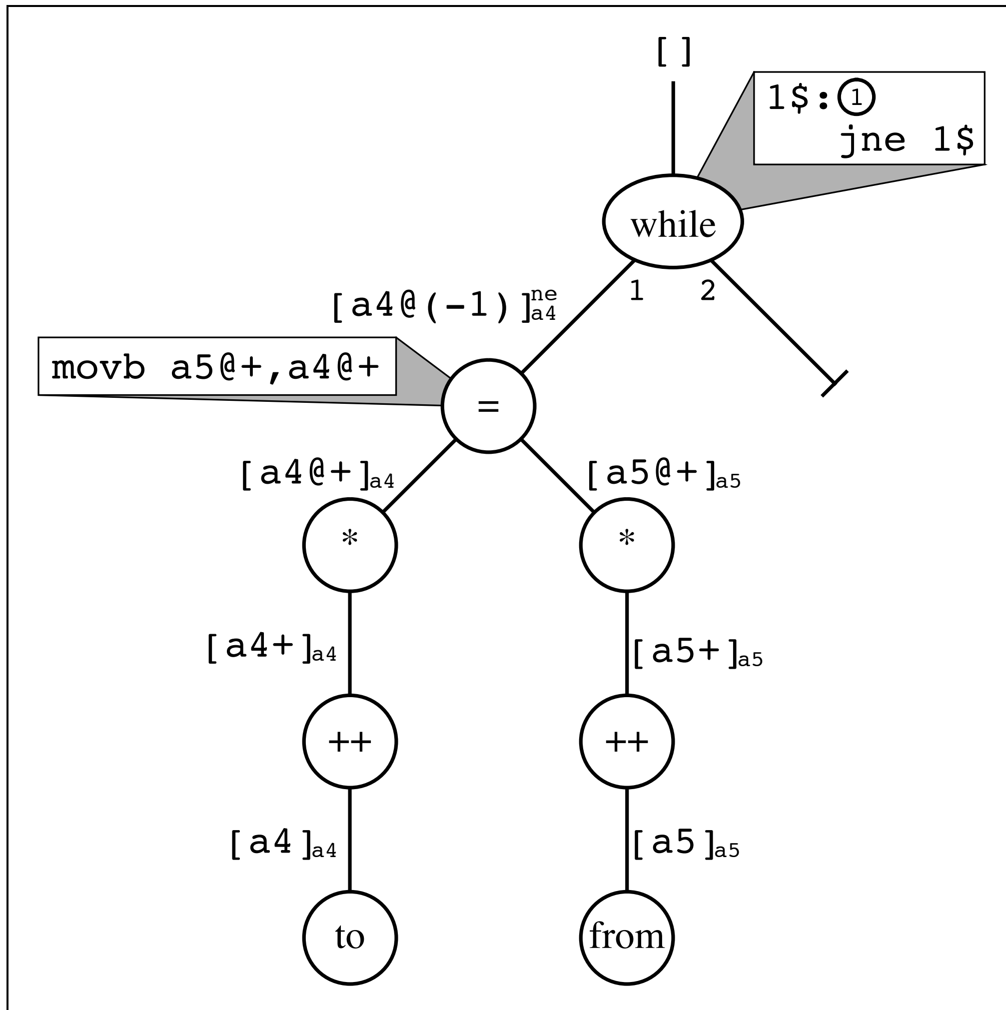


**Figure D.2:** Parse tree, operand descriptors and generated code for ‘while(*to++= *from++);’

## D.4 Generated Code For `nfib`

To illustrate the code generated by the back-end described in chapter 6, the definition and final compiled code for the ‘nfib’ function is quoted below. (The skeptical reader can write down the parse tree and compile the method “by hand” to verify that the code shown is indeed produced by the code generator described above.) The comments in `typewriter` font are those inserted by the compiler; those in *italic* are a commentary. So, the function

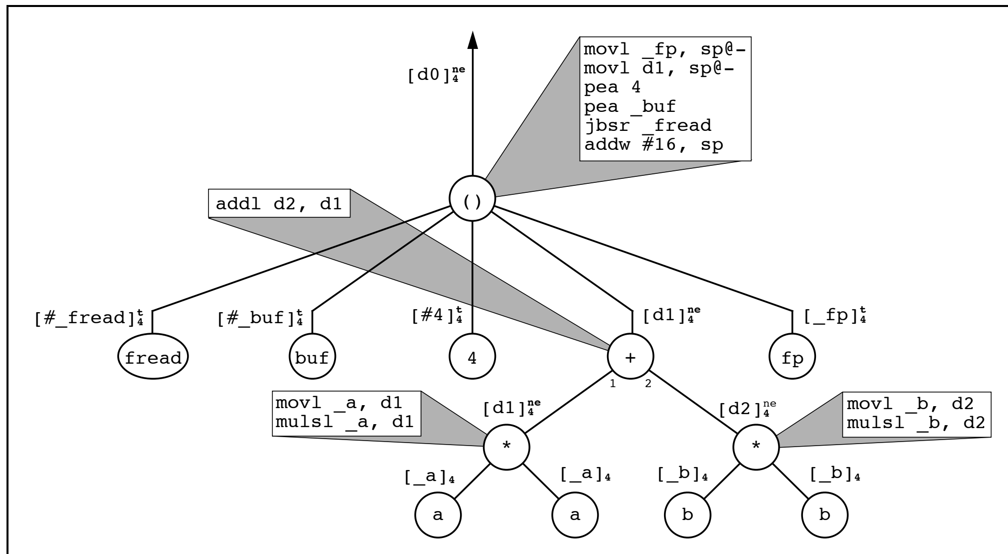


**Figure D.3:** Parse tree, operand descriptors and generated code for ‘`fread(buf, sizeof int, a*a + b*b, fp);`’

---

```
SmallInteger>>nfib
    ↑self <= 1
        ifTrue: [1]
        ifFalse: [(self - 1) nfib + (self - 2) nfib + 1]
```

results in the compiled code shown on the next page (all optimizations are enabled).

```
BEGIN(SmallInteger__nfib)
      .long O(24074)       | literal frame
      pea   a3@            old home
      pea   a6@            old frame
      movl  sp, a6         new frame
      pea   O(48082)       | method
      movl  a6@(12), d0    self
      btst  #0, d0         integer?
      jeq   2$             no
      cmpl  #I(1), d0      <= 1?
      jhi   1$             no
3$:   movl  #I(1), d0      return 1
      jra   4$             exit
1$:   movl  a6@(12), d0    self
      btst  #0, d0         integer?
      jeq   5$             no
      subql #2, d0         self - 1
6$:   movl  d0, sp@-       1st arg
      movl  sp@, d0        receiver
      movl  #O(1415), d1 | nfib
      jbsr  Send
      addql #4, sp         pop args
      movl  d0, sp@-       result
      movl  a6@(12), d0    self
      btst  #0, d0         integer?
      jeq   7$             no
      subql #4, d0         self - 2
8$:   movl  d0, sp@-       1st arg
      movl  sp@, d0
      movl  #O(1415), d1 | nfib
      jbsr  Send
      addql #4, sp         pop args
      movl  d0, d1         rhs
      movl  sp@+, d0       lhs
      btst  #0, d0         inline?
      jeq   9$             no
      btst  #0, d1         inline?
      jeq   9$             no
      addl  d1, d0         yes
      subql #1, d0         adjust tag
10$:  btst  #0, d0         integer?
      jeq   11$            no
      addql #2, d0         + 1
4$:   unlk  a6             old frame
      movl  sp@+, a3       old home
      rts                  exit
            deferred sends...
2$:   movl  d0, sp@-
      movl  #I(1), sp@-
      movl  sp@(4), d0
      movl  #O(1818), d1 | <=
      jbsr  Send
      addql #8, sp
      cmpl  #TRUE, d0
      jeq   3$
      jra   1$
5$:   movl  d0, sp@-
      movl  #I(1), sp@-
      movl  sp@(4), d0
      movl  #O(3663), d1 | -
      jbsr  Send
      addql #8, sp
      jra   6$
7$:   movl  d0, sp@-
      movl  #I(2), sp@-
      movl  sp@(4), d0
      movl  #O(3663), d1 | -
      jbsr  Send
      addql #8, sp
      jra   8$
9$:   movl  d0, sp@-
      movl  d1, sp@-
      movl  sp@(4), d0
      movl  #O(3650), d1 | +
      jbsr  Send
      addql #8, sp
      jra   10$
11$:  movl  d0, sp@-
      movl  #I(1), sp@-
      movl  sp@(4), d0
      movl  #O(3650), d1 | +
      jbsr  Send
      addql #8, sp
      jra   4$
END(SmallInteger__nfib)
```

# Appendix E

# Further Implementation Notes

This appendix presents some brief notes on the more important aspects of implementation that would be addressed if Native Code Smalltalk-80 were to be developed into a 'production' Smalltalk-80 system. These comments are provided 'without warranty', and are based as much on supposition and gut feeling as they are on hard facts.

## E.1 Further Improvements in the Generated Code

There is still room for some improvement in the compiled code, to reduce both execution time and the overall size of compiled methods.

### E.1.1 Shared Deferred Sends

The amount of extra method space taken by deferred sends is directly proportional to the number of deferred sends, regardless of the mix of selectors between them. Where a particular message is inlined several times, each occurrence will cause a deferred send. A single deferred send could be shared by each of these inlined sends by using a relative, indirect 'join' address. (The same comment applies to forking where a pair of relative, indirect 'join' addresses would be used for the 'fork' and 'no fork' cases.)

For example, the generated code for 'nfib' shown on page 138 has three separate deferred sends each dealing with an inlined addition. Much space could be saved in the compiled code by sharing a single deferred send between several inlined sends of the same selector, placing an offset in an address register indicating the location in the method body at which the value is required.[1]

The code for an inlined send would now look something like this:

[1]This address register would have to be saved on entry to, and restored on exit from, every method with inlined sends. Garbage collection would be complicated slightly due to this. Alternatively every method could save and restore this register (or a dummy value, or even just decrement the stack pointer since this stack location is ignored during garbage collection), making for a trivial change to garbage collection but introducing a slight redundancy in methods that have no inlined sends.

```
        arguments in d0 and d1
        lea     FULL-CONT+N, aN   | (relative) join address
        btst    #0, d0            | SmallInteger?
        jeq     FULL              | no
        btst    #0, d1            | SmallInteger?
        jeq     FULL              | no
        perform inlined operation, result in d0
CONT:   rest of method...

        deferred sends...
FULL:   movl    d0, sp@-
        movl    d1, sp@-
        movl    #selector, d1
        jbsr    Send              | perform full send
        addql   #8, sp
        jmp     pc@(aN)           | rejoin method body
```

(Compare this with the code shown on page 92.)

### E.1.2 Better Use of Class Information

In some situations the code generator can emit a SmallInteger tag check even though it knows that the quantity it is dealing with is a SmallInteger. The most common case is when several inlined arithmetic operations appear in an expression. The 'nfib' example described in section D.4 contains the following expression:

(self - 1) nfib + (self - 2) nfib + 1

which reduces to

$\langle thing1 \rangle + \langle thing2 \rangle + 1$

incurring an unnecessary tag check in the second addition if the inlined version of the first addition succeeded. (See the code at and around the label `10$` on page 138.) The code generator could make a simple rearrangement of the code to avoid the class check in the second addition in cases where the first inlined addition is executed. This would require a little thought about communicating the necessary class information between successive message sends, but should not be impossible to implement.

## E.2 Support for Debugging

### E.2.1 Failed Message Sends

Smalltalk recovers from an unknown method by sending the receiver of the errant message a 'doesNotUnderstand:' with the failed Message as the argument. The Message

contains the failed selector and an Array of the actual arguments. For this Array to be constructed, the number of actual arguments must be available at runtime. In the code produced by the compilers described in this report, this information is not available.

Various solutions present themselves, but most have at least one drawback that prevents their use in a realistic implementation.

No mechanism for counting the arguments at runtime is possible since sends associated with the computation of arguments for an enclosing message send are accumulated on the stack. There is consequently no convenient non-OOP 'marker' on the stack from which to count the number of actual arguments.

The selector is available to the runtime system, and it is not inconceivable that this could be used to determine the number of arguments expected by the intended destination method. Determining the argument count by inspecting the type of the selector (unary, binary, or keyword with embedded colons) would work if messages failed only from within compiled code. Unfortunately messages can also fail due to the 'perform' primitives, which could defeat a selector inspection mechanism if the supplied Array of arguments is not of the size implied by the selector. However, if sends occurring indirectly via this primitive (and it's close relatives) were handled specially, then this mechanism could be used in practice.[2]

The only safe mechanism is to place a count of the actual arguments in a register prior to the message send. For compiled sends, this figure is available at compile time. For indirect sends via the 'perform' primitives, this can be determined easily at runtime from the actual arguments supplied prior to performing the message send. Given such a count the solution to the problem follows trivially, although this is the least desirable solution because of the added overhead on each message send.

One possible alternative will be mentioned later during the discussion on inlined caches.

## E.3 Inline Caches

A preposterous[3] amount of time is spent by the runtime support in performing dynamic binds. Even with the method cache hit rate at effectively 100% (the 'working set' of <class,selector> pairs for even the macro benchmarks is relatively small), profiling the benchmark suite revealed the following situation:

[2]Crafty programmers could still defeat this by changing the compiler's notion of what constitutes a binary selector. The runtime system would be oblivious to such changes.

[3]At least for an implementation that is already faster than PS2.3 by a factor of 3.8 `;-)` (section B.2)

| %time | name | *English translation* |
|---:|---|---|
| 43.3 | `Send` | *dynamic binding* |
| 18.1 | `VALERR` | *primitives* |
| 13.2 | `_GC` | *GC: mark/sweep* |
| 10.1 | `_BitBlt` | |
| 8.5 | `_compact` | *GC: compaction* |
| 3.8 | `AllocS` | *object allocation* |
| 2.3 | `primtab` | *compiled code* |
| 0.3 | `_main` | *initialization* |
| + traces <= 0.1 | | |

(‘`VALERR`’ appears near the top because it is the last external symbol defined in the runtime system before the primitives; ‘`primtab`’ is the last external symbol in the runtime system and will occur immediately before the start of the loaded image — the associated figure represents the time spent executing the compiled code in the image.) The fundamental need for an inline cache should be obvious from this table.

### E.3.1 Inline Cache Design

An inline cache needs to handle three situations:

**Initialization**
The first time a particular message send is encountered, the inline cache must be initialized by performing a full dynamic bind (via the method cache, if one is present). The call site must be patched with a reference to the destination method, before that method is entered.

**Cache hit**
Due to the locality of type usage found in Smalltalk, the second and subsequent encounters of some particular message send will most probably be to the same class of receiver. In such cases, the previously determined method can be entered without delay.

**Cache miss**
If the class of the receiver differs from that of the previous receiver then a ‘re-initialization’ must occur, patching the call site with the result of a new full dynamic bind before entering the newly determined method.

Inline caches can be implemented in many different ways, but all have to manage essentially the same few pieces of information in order to perform the tasks listed above:

- the class of the receiver,
- the selector,

- the cached method address from the previous send, and
- the class of the previous receiver.

The variation between designs comes in the dynamic relationships between various items in the list above, and the location(s) in which they are kept:

- embedded within the code at the 'point of send',
- with the destination method, or
- in an auxiliary data structure appended to the method proper.

Choice of representation for method references, such as object memory address versus OOP, also increases the range of designs possible. Assuming we can choose either a real address or OOP for the cached method address, ignoring the additional degrees of freedom introduced by any dynamic relationships between different quantities, and assuming no flexibility in where each of the previously mentioned tasks are to be performed, we already have at least 162 possible organizations for our inline cache.[4]

Rather than embark upon an exhaustive investigation into inlined cache construction (which might well make an admirable thesis topic in itself), we will consider very briefly just one likely looking candidate.

### E.3.2 Example Inline Cache

This is a fairly straightforward inline cache. The only caveats are that the OOP of the destination method is cached rather than the address (this simplifies garbage collection enormously), and the 'inline cache binding materials' (ICBMs) are kept together in a silo at the end of the method after the compiled code (which removes a little overhead associated with tweezing the return address to skip these materials were they to be kept inline at the point of send). Thus:[5]

[4]Admittedly, a significant number of these will be meaningless, or at best highly inefficient.

[5]Note the ICBM field containing the number of arguments — this is to help with the handling of failed messages, and the consequent construction of the Array of arguments for the resulting 'doesNotUnderstand:'.

```
            usual send prologue: arguments on stack,
            receiver in d0, NO SELECTOR NEEDED
            lea    pc@(icbm42), aN    | address of ICBM
            movl   aN@(0), aM         | method OOP
            jbsr   obmem@(aM)@(12)    | call method
            addql  #nArgs, sp         | tidy stack
            rest of method...

            end of method, start of silo
            some ICBMs...
icbm42:     .long  method OOP
            .long  selector
            .long  previous class
            .long  argument count
            some more ICBMs...
```

Each method entry now has a prelude that checks the inline cache for a match between the actual receiver's class and the previous receiver's class:

```
        12 bytes of method header
entry:  arrive here with ICBM address in aN
        movl  obtab@(d0)@(0), aC  | class of receiver
        cmpl  aC, aN@(8)          | cache hit?
        jeq   hit                 | yes - enter method
        jmp   icMiss              | no - patch up calling ICBM
hit:    normal method entry...
```

The cache miss code (only one copy, somewhere inside the runtime system) performs the dynamic binding:

```
        recover from inline cache miss
icMiss: arrive with ICBM address in aN, receiver class in aC
        movl  aN@(4), aS                | selector
        bind on aC×aS, result in aM
        movl  aM, aN@(0)                | patch method OOP
        movl  aC, aN@(8)                | and previous class
        dispatch to method, skipping cache class check...
        jmp   obmem@(aM)@(12+hit-entry)
```

Cache initialization can be ensured by constructing ICBMs so as to point to an arbitrary method (either something unlikely to change such as ‘Object>isNil’ or some method defined especially for this purpose that has been removed from its class organization so as to make it invisible to the programmer), and then put some ludicrous value in the ‘previous class’ field (such as ‘nil’ or zero). The first encounter of the send would consequently miss the cache regardless of the class of the receiver, causing the correct method OOP to be found and placed in the cache.

With a little ingenuity it is possible to reduce the number of fields in each ICBM, and hence the total size of the compiled methods in the image, by effectively moving information from the call site to the destination site (there will always be many more call sites than destinations). For example the selector need not be cached at the point of send but instead kept with the destination method itself, although this complicates the initialization of the 'method OOP' field for sends in which the selector is a new message.

More efficient schemes are also possible if self-modifying code is used. However, certain processors explicitly forbid self-modifying code because of problems of consistency between their on-chip code and data caches.[6] Some initial experiments,[7] with inline caches in code compiled 'by hand' in the spirit of a delayed code generator, puts the performance of the compiled code for the 'nfib' example at just over 70% that of the equivalent C. The same experiments suggest that the application of a little flow analysis, and further specialization in the compiled code for arithmetic and relational operations on receivers discovered to be SmallIntegers at runtime, would almost remove the performance gap between Smalltalk and C.[8]

### E.3.3 Interference with Deferred Sends

Inline caches become less effective if shared deferred message sends are being used. The more inlined selectors that share a single deferred send, the less the consistency of type usage at that send. In the worst case, we might have a method that adds a pair of Points and then a pair of Fractions; the inline cache at the associated deferred send would thrash mercilessly.

### E.3.4 Cache Consistency

When compiling or removing a method, Smalltalk is careful to maintain the consistency of any caches that may become inconsistent. The method cache is fairly easy to purge, but an inline cache presents a much more serious problem. The solution depends to a large extent on the particular implementation strategy adopted, but the choice would seem to be between the following:

- keeping out of line structures at the end of methods holding inline cache information. Purging these becomes a matter of trawling the object memory for CompiledMethods and then sifting through the inline cache structures appended to the compiled code;

- keeping the cache information embedded within the code itself at each point of send, and appending a 'map' of the locations of each send point to the method;

[6] The MC68040, for example, tends to catch fire when presented with self-modifying code.

[7] Undertaken by Eliot Miranda.

[8] The technique is similar to the 'method splitting' used by the SELF compiler, but applied in a much more selective fashion in order to avoid SELF's notorious space explosion problems.

- keeping the cache information embedded within the code itself at ech point of send, and marching through the code looking for a particular instruction sequences which act as a ‘signature’ for the inline cache.

# Bibliography


[ASU86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Addison-Wesley, 1986. ISBN 0-201-10194-7.

[Ash87] Peter James Ashwood-Smith, "The Source Level Optimization of Turing Plus", University of Toronto Dept. of Computer Science, March 7, 1987.

[Atk86] Robert G. Atkinson, "Hurricane: An Optimizing Compiler for Smalltalk", SIGPLAN Notices 21(11) pp. 151-158, November 1986.

[Bor79] Richard Bornat, "Understanding and Writing Compilers", Macmillan Publishers Ltd., 1979. ISBN 0-333-21732-2.

[CU89] Craig Chambers and David Ungar, "Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language", in Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation, published as SIGPLAN Notices 24(7), July 1989.

[CUL89] Craig Chambers, David Ungar and Elgin Lee, "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes", in Proc. of the Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 49-70, October 1989.

[CP83] W. Citrin and C. Ponder, "Implementing a Smalltalk compiler", in Smalltalk on a RISC: Architectural investigations, proceedings of CS292R, pp. 167-185, Department of Computer Science, University of California at Berkeley, April 1983.

[CW73] R. W. Conway and T. R. Wilcox, "Design and implementation of a diagnostic compiler for PL/I", CACM 16(3) pp. 169-179, March 1973.

[Cor86] James R. Cordy, "An Orthogonal Model for Code Generation", Technical Report CSRI-177, Computer Systems Research Group, University of Toronto, January 1986.

[CH90] J. R. Cordy and R. C. Holt, “Code Generation Using an Orthogonal Model”, Software Practice and Experience 20(3) pp. 301-320, March 1990.

[DB76] L. Peter Deutsch and Daniel G. Bobrow, “An Efficient, Incremental, Automatic Garbage Collector”, CACM 19(9) pp. 522-526, Sept 1976.

[DS83] L. Peter Deutsch and Alan M. Schiffman, “Efficient Implementation of the Smalltalk-80 System”, in PROC 11th ACM POPL, Salt Lake City, UT, 15–18 January 1984.

[Dyb87] R. K. Dybvig, “The Scheme Programming Language”, Prentice-Hall Inc., 1987. ISBN 0-13-791864-X.

[Eli88] Nicholas L. Eliot, “Automatic Derivation of Orthogonal Code Generators from Applicative Specifications”, M.Sc. Thesis, Queen’s University, Kingston, Ontario, Canada, July 1988.

[GR83] Adele Goldberg and David Robson, “Smalltalk-80: The Language and its Implementation”, Addison-Wesley, 1983. ISBN 0-201-11371-6.

[Hal86] Charles Brian Hall, “A Production-Quality Machine-Independent Code Generator for Turing”, M.Sc. Thesis, Department of Computer Science University of Toronto, September 1986.

[HP90] John L. Hennessy and David A. Patterson, “Computer Architecture — A Quantitative Approach”, Morgan Kaufmann Publishers, Inc., San Mateo CA, 1990. ISBN 1-55880-069-8.

[Hol87] R. C. Holt, “Data Descriptors: A Compile-Time Model of Data and Addressing”, ACM TOPLAS, vol. 9, no. 3 (July 1987), pages 367-389.

[HCW82] R. C. Holt, J. R. Cordy and D. B. Wortman, “An Introduction to S/SL: Syntax/Semantic Language”, ACM TOPLAS 4(2) pp. 149-178, April 1982.

[HWW87] Trevor Hopkins, Ifor Williams and Mario Wolczko, “MUSHROOM—A Distributed Multi-User Object-Oriented Programming Environment”, in Proc. Joint Workshop of the BCS Parallel Processing and Object Oriented Programming and Systems Specialist Groups, London, October 1987.

[Jan90] Goran T. Janevski, “Automatic Generation of Modular Semantic Analyzers from Functional Specifications”, M.Sc. Thesis, Queen’s University, April 1990.

[JGZ88] Ralph E. Johnson, Justin O. Graver and Lawrence W. Zurawski, “TS: An Optimizing Compiler for Smalltalk”, in OOPSLA ’88 Conference Proceedings, Published as SIGPLAN Notices 23(11), November 1988.

[KR78] Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, Prentice Hall, 1978. ISBN 0-13-110163-3.

[Kra83] Glenn Krasner, “Bits of History, Words of Advice”, Addison-Wesley, 1983. ISBN 0-201-11669-3.

[LB83] J. Larus and W. Bush, “Classy: A method for efficiently compiling Smalltalk”, in Smalltalk on a RISC: Architectural investigations, proceedings of CS292R, pp. 186-202, Department of Computer Science, University of California at Berkeley, April 1983.

[McC60] John McCarthy, “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”, CACM vol. 3 no. 4, pp. 184-195, 1960.

[McC62] John McCarthy et al., “LISP 1.5 Programmer’s Manual”, MIT Press, Cambridge MA, 1962.

[Mir87] Eliot Miranda, “BrouHaHa — A Portable Smaltalk Interpreter”, in Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, published as ACM SIGPLAN Notices 22(12) pp. 354-365, December 1987.

[Mos87] J. Eliot B. Moss, “Managing Stack Frames in Smalltalk”, Proceedings of the SIGPLAN ’87 Symposium on Interpreters and Interpretive Techniques, published as SIGPLAN Notices 22(7) pp. 229-240, July 1987.

[Mot85] Motorola Inc., “MC68020 32-Bit Microprocessor User’s Manual”, Second Edition, Prentice-Hall Inc., Englewood Cliffs NJ, 1985. ISBN 0-13-566860-3 (Prentice-Hall Edn.), 0-13-566878-6 (Motorola Edn.).

[PH90] David A. Patterson and John L. Hennessy, “Computer Architecture — A Quantitative Approach”, Morgan Kaufmann Publishers, Inc., San Mateo CA, 1990. ISBN 1-55880-069-8.

[Ros80] Alan Rosselet, “PT: A Pascal Subset”, Technical Report CSRG-119, Computer Systems Research Group, University of Toronto, September 1980.

[Sun88] “Assembly Language Reference for the Sun-2 and Sun-3”, Part Number 800-1773-10, Sun Microsystems Inc., Mountain View CA, May 1988.

[ST84] N. Suzuki and M. Terada, “Creating efficient systems for object-oriented languages”, in Proc. of the Eleventh ACM Symposium on the Principles of Programming Languages, pp. 290-296, January 1984.

[Wil71] T. R. Wilcox, "Generating machine code for high-level programming languages", Ph.D. dissertation, Computer Science Dept., Cornell University, Ithaca, NY, 1971.

[Wil89] Ifor W. Williams, "The MUSHROOM Machine – An Architecture for Symbolic Processing", in Proc. IEE Colloquium on VLSI and Architectures, London, March 1989.

[Wol88] Mario I. Wolczko, "Introducing MUST — The MUSHROOM Programming Language", MUSHROOM project technical report, University of Manchester, UK, October 1988.

[Wol84] Mario I. Wolczko, "Implementing Smalltalk-80 on the ICL PERQ", M.Sc. Thesis, Department of Computer Science, University of Manchester, UK, October 1984.