University of Oslo
Department of Informatics

Development of a Simula front-end to GCC

Knut Aksel Røysland

Cand. Scient. Thesis

January 2002

Preface

This document constitutes the research report of my Cand. Scient. thesis at the Department of Informatics, University of Oslo. The thesis was carried out over a two-year period from 1999 to 2001 and consisted of performing and documenting the development of a Simula implementation based on the GNU Compiler Collection (GCC).

I would like to thank my supervisor, professor Stein Krogdahl, who provided valuable assistance in the preparation of this thesis. I also want to acknowledge the efforts of the developers of Cim, the inventors of Simula, and the contributors to the GCC project.

I chose to write this document in English to obtain experience in presenting textual material in this language.

Knut Aksel Røysland

Contents

Chapter 1

Introduction

1.1 Objective

The objective of this study has been to design, build and evaluate a Simula compiler that is based directly on the multi-platform optimization and code generation framework of the GNU Compiler Collection (GCC). Utilization of this framework, known as the GCC back-end, should enable this Simula implementation to generate good-quality assembler code for the variety of machine platforms that are supported by GCC.

In addition to the GCC back-end, such a Simula implementation would consist of two main parts—a Simula front-end, which cooperates with the GCC back-end at compile-time, and a Simula run-time system, which is used by Simula programs at run-time. The new Simula implementation envisioned in this report, is referred to as *GSC* (GCC-based Simula Compiler). So far, only parts of the front-end has been implemented.

The main focus of this thesis can be naturally divided into two fairly independent aspects:

- The first concerns the suitability of the GCC back-end with respect to the Simula language. Evaluation of the back-end's suitability is based on a comparison with an existing Simula implementation which uses the C programming language as intermediate, platform-independent representation. This comparison provides for a discussion on the GCC back-end's interface compared to C, as means for expressing the semantics of Simula programs in a platform-independent fashion that still facilitates optimization and platform-specific code generation.

- The second aspect concerns a few "high-level" optimizations that have been employed and tested in the the current GSC implementation. These optimizations involve techniques that seek to favor execution speed at the expense of memory parsimony—a principle that has been discussed throughout the history of Simula, although, due to memory scarcity being a main concern traditionally, such techniques have not been extensively employed in actual implementations. Today, common optimization practices in general, typically involve sacrificing memory if time-efficiency can be improved. The optimization techniques presented in this report, may be regarded as Simula's counterparts to this trend.

The C-based Simula implementation, Cim [7], has been used as a reference in this study, and, as indicated above, it will be used to compare the GCC back-end and C. In addition, Cim has provided technical inspiration to the structure of GSC.

It is worth noting that the two aspects outlined above interfere with each other. For instance, when observing diverging properties in GSC and Cim and the programs they generate, this may be the results of either one or both of the two aspects. However, it is essential to the accuracy of this study, to identify which one of them any particular advantageous or disadvantageous behavior can be ascribed to.

1.2 Background

The compilers that comprise the GNU Compiler Collection, share a common framework for optimization and code generation. The manifestation of this framework, is a component called the *back-end*. The term "back-end" relates to the conceptual position of this component in the compilation process. It performs optimization and generates assembler code based on an intermediate representation of the source program being compiled. This representation is provided to the back-end by the complementing part—the *front-end*.

The intermediate representation that is used between the front-end and the back-end, has to facilitate effective communication and express the source program's semantics in an accurate manner, with a perspective spanning as many programming languages as feasible on one side, and a wide variety of target machine platforms on the other. The result is a quite comprehensive interface which has to be mastered adequately by a front-end developer.

Hence, much of the background study has consisted of reading and analyzing the GCC back-end and its interface. Additionally, several existing GCC front-ends have been studied, most notably the C front-end, which is the original front-end of GCC, and the Java front-end, which is one of the recent additions to the set of GCC front-ends.

The Simula-to-C compiler, Cim, has been a central source of information with respect to the Simula programming language. Simula literature has also provided valuable insight, like Ole-Johan Dahl's "*Runtime organisasjon for*

Algol/Simula" [6] and the Simula Standards Group's formal specification of Simula [5] from 1986.

1.3 Conception

The work with the thesis was initiated September 1999. The original idea to develop a Simula front-end to GCC, was based on my desire to pursue knowledge in compiler development. The exact choice of technology came as a result of my admiration of the GNU Project [1] from which GCC originates. My motivation for choosing Simula was twofold:

- Simula is closely related to my department—the Department of Informatics at University of Oslo. Simula was conceived at the Norwegian Computing Centre in Oslo in the 1960s, and the Department of Informatics has hosted both of its originators, Ole-Johan Dahl and Kristen Nygaard.

- Simula contains certain elements that dictate the structure of the run-time organization in a way which deviates from traditional organizations typically employed in programs compiled from other languages, like C. In particular, arranging for Simula's coroutine mechanism, reveals a challenge to the back-end interface of GCC, and will be given much attention in this report.

I contacted professor Stein Krogdahl who has participated in the development efforts of several former Simula implementations. In particular, he has supervised the development of Cim. He agreed to be my supervisor and has provided valuable inspiration and feedback throughout the work with this report.

1.4 Organization of this report

Chapters 2 and 3 provide an overview of the background information that I had to acquire prior to and during the design and implementation of GSC. The design and implementation efforts themselves, are presented in chapters 4 and 5, while chapters 6 and 7 comprise the results and concluding discussions.

Chapter 2 outlines the structure of the GCC back-end and gives a rough introduction to its interface. The interface is comprehensive, and an in-depth presentation would far exceed the scope of the chapter.

Chapter 3 presents the Simula programming language with emphasis on its support for *coroutines* and what consequence they have on the run-time organization.

Chapter 4 outlines the main design challenges and describes the most important decisions that I made in light of chapters 2 and 3.

Chapter 5 gives an overall description of the implementation work, including some technical details.

Chapter 6 presents a few selected Simula programs that have been compiled using GSC. Code efficiency and quality is compared to the corresponding code produced by Cim.

Chapter 7 provides a summary and also suggests further considerations and studies.

Chapter 2

The GCC back-end

This chapter provides a brief presentation of GCC and the communication between a front-end and the back-end. The back-end's interface is quite comprehensive, so examining it and providing a complete reference, constitute a considerable effort. This presentation is merely intended to be introductory and is biased toward the aspects that are most relevant to the GSC implementation.

2.1 Background

The *GNU Compiler Collection* (GCC) comprises compilers for a number of programming languages, most notably C, C++, Java, FORTRAN, Pascal and Ada.

GCC is an important part of the *GNU Project* [1], initiated by Richard Stallman in 1984. The people contributing to the GNU Project and its backing organization, the *Free Software Foundation*, are devoted to developing *free*[1] *software*. The ideology surrounding free software, is a study of its own. In this context, the most important property of GCC as being free software, is its providing for unrestricted access to all the source code comprising the project. For my study, it has also been a decisive factor that GCC is regarded as being of good technical quality and that it is continually being improved and refined by its wide base of developers as well as quality-conscious users.

2.2 Front-end and back-end

A GCC-based compiler is constructed by joining the GCC *back-end* and a *front-end*. The front-end is specific to the programming language for which the compiler is to work. It parses and translates the source program into a

[1]"free" as in "freedom", as opposed to having zero price

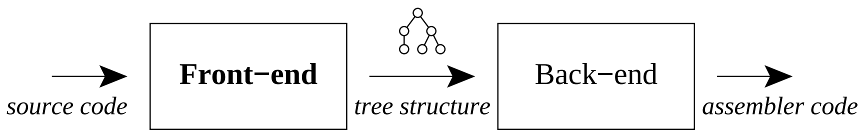

Figure 2.1: Overall structure of a GCC compiler

representation which is handed over to the back-end. The back-end, which is common to all GCC-based compilers, optimizes the representation and translates it into assembler language for a destined target machine platform.

Each machine platform supported by GCC, is represented by a *machine description* which specifies the properties of the platform and enables the back-end to produce assembler code, accordingly. When the back-end is configured and built, the appropriate machine description is selected and incorporated into the generic back-end source code through macro expansions, i.e. the macros are expanded using definitions provided by the machine-specific description. By default, the machine description corresponding to the platform of the present machine and operating system, is chosen—resulting in a *native compiler*. However, another description may be selected explicitly, in order to build a *cross-compiler*.

By limiting GCC to containing only one generic back-end, the intricate functionality involving code optimization, is kept in one place. In order to extend GCC to supporting additional languages or machine platforms, new front-ends or new machine descriptions have to be created, but the most complex part, the back-end, remains the same. This way, the total effort involved in creating a collection of compilers for a certain set of language and architecture combinations, is reduced from $O(M \cdot N)$ to $O(M+N)$ where M is the number of languages and N the number of machine platforms.

2.3 Overview

The versatile nature of the back-end's interface gives rise to an elusive conceptual demarcation between the front-end and the back-end. Some of the functionality included in the GCC distribution, provides routines intended to aid the front-end in its internal work. Conversely, a front-end may provide certain "hook" functions which perform back-end-related work and which are activated from within the back-end. This text will not refer to the terms "front-end" and "back-end" in a way that necessitates an exact definition of the two. However, for the sake of simplicity I will denote all source code written by me as part of the front-end, as well as source code stemming from other front-ends of GCC. The remaining source code (including machine descriptions) is classified as part of the back-end.

The starting point of execution in a GCC compiler, lies in the back-end. Before activating the front-end, the back-end analyzes all recognized command-line options, like the `-O`*n* option which chooses the level of back-end optimization effort. It also initializes various global structures, like those that represent certain basic data types and certain constant values. After that, the front-end is activated.

Typically, the two main parts of a front-end are the syntactic and the semantic analyzers. The syntactic analyzer (or parser) retrieves lexical tokens from a lexical analyzer. The semantic actions in the grammatical productions of the parser, then constructs a syntax tree which may ultimately comprise the entire source program, provided that the program is syntactically correct. Depending on the properties of the source programming language, semantic analysis is done either during syntactic analysis or completely after it. Languages like C permits semantic analysis to be performed along with syntactic analysis, while Simula requires a separate semantic analysis pass because identifiers may be referenced prior to their declarations in the source code.

The front-end may be considered the "driving" part during the actual compilation, while the back-end operates like a function library which acts upon requests from the front-end. When the semantic analyzer finishes, the front-end's job is done and control returns to the back-end which performs any pending work and outputs the final assembler code.

2.4 Register Transfer Language (RTL)

The GCC back-end uses an internal representation referred to as *Register Transfer Language* (RTL). This representation was inspired by an implementation of a peephole optimizer, called PO, developed by Davidson and Fraser [4]. It is a representation that replaces assembler language as the basis for optimization analysis. The RTL representation undergoes several optimization passes which include jump optimization, common subexpression elimination (CSE), loop optimization, flow analysis, instruction combination, instruction scheduling and register allocation [3].

A front-end is allowed to produce RTL code directly, but the preferred interface is based on tree structures much like syntax trees. There may, however, be certain situations where interfacing RTL directly may be convenient due to some expressive limitations in the syntax tree interface.

2.5 The front-end/back-end interface

The bulk of the interaction between a front-end and the back-end, takes place while the semantic analyzer of the front-end activates back-end functions in order to perform optimizations and generate assembler code. As indicated already, most of the information transmitted to the back-end through these calls, is in the form of tree structures. These tree structures need to conform to the specifications stated by the back-end interface, or else the back-end will fail in confusion.

To take GSC as a concrete example, the parser of the Simula front-end produces a syntax tree. When the parser completes, the syntax tree contains a combination of tree nodes, some of which will end up in the back-end, while others are only significant to the semantic analyzer (which has not yet started) and then thrown away before the back-end gets involved. An example of nodes that are thrown away by the semantic analyzer, are nodes that represent textual identifiers. When the semantic analyzer performs symbol look-up, it replaces such nodes with the nodes that represent the actual associated entities (e.g. variable, procedure, or class declaration), according to the scope and visibility rules of Simula.

This is one of several examples that the back-end permits front-end specific extensions to be part of the tree structures. This flexibility enables a front-end to accommodate its own internal needs within the same data structure framework. This is no problem as long as these extensions do not conflict with the back-end expectations of the tree structures.

The flexibility illustrated here is typical for the back-end interface and how it tries provide effective and seamless interaction between the front-end and itself with a minimal amount of duplicated work. The interface may be likened to the fingers of two folded hands rather than a definite flat intersection.

2.5.1 The tree structure

The rest of this chapter will give a more detailed presentation of the back-end interface. These details are not strictly needed in order to comprehend the rest of this report, but they will hopefully provide a convenient introduction for the curious reader.

The general internal structure of a tree node, as defined by the back-end, is a `union` containing several `struct`s. Since a C `union` defines alternative internal structural layouts, it effectively causes tree nodes to have varying structure depending on the purpose of the node. All nodes, however, share a common set of attributes (a *header*), making them partly isomorphic. In a sense, this is analogous to having an abstract super-class which defines the header, and then having the alternative tree structures represented by sub-classes of this super-class. This is C, however, so resorting notions like *inheritance*, borrowed from object-oriented language terminology, is inappropriate in this context.

Listing 2.1 shows the complete definition of `union tree_node` which constitutes the general definition of a node. The first attribute is `common`, which is of the compound type struct tree_common and represents the common header mentioned above. Listing 2.2 shows the most important attributes of this header.

Listing 2.1: Top-level union-definition of the back-end's tree node data type

```
union tree_node
{
  struct tree_common common;
  struct tree_int_cst int_cst;
  struct tree_real_cst real_cst;
  struct tree_string string;
  struct tree_complex complex;
  struct tree_identifier identifier;
  struct tree_decl decl;
  struct tree_type type;
  struct tree_list list;
  struct tree_vec vec;
  struct tree_exp exp;
  struct tree_block block;
};
```

Listing 2.2: All tree nodes contain the attributes from struct tree_common

```
struct tree_common
{
  union tree_node *chain;
  union tree_node *type;
  enum tree_code code : 8;
   ⋮
};
```

The other attributes of union tree_node, which are also structs, include struct tree_common as their first attribute, as well, consequently resulting in coinciding positions of struct tree_common at the very front of any tree node no matter how union tree_node is qualified.

Common attributes

One of the attributes shown in listing 2.2 is code. Being common to all tree nodes, the code attribute is used to store a numeric value that determines what type of tree node it is. This value is referred to as the *tree code* and gives an indication of the purpose of the node. The back-end defines the basic set of tree codes by means of an enumeration type that associates textual symbols to the underlying numeric values. In the following text, tree codes will be referred to by their textual equivalents.

Table 2.1 lists all the 145 tree codes defined by the back-end of GCC version 3.0.2, and gives a loose impression of the extent of the back-end interface. Tree codes form natural categories which are indicated by the suffixes of the codes.

- The **TYPE** category represents codes used in root nodes of data type trees, i.e. tree structures that represent data types in the source language.

- The **CST** category contains nodes that represent (compile-time) constant values, typically stemming from literals found in the source program being compiled.

- The **DECL** category contains nodes used to represent declarations, i.e. entities, like variables, procedure and classes which are declared in the source program.

- The **REF** category of codes are used in root nodes of reference trees, i.e. sub-trees within syntax trees, which represent accesses to actual entities that have a position in memory, like a variable, an array element or an object attribute. In languages like C, REF-trees are used when the access is remote (through a pointer), or when the element being referenced is not represented directly by a declaration node of its own, like array elements or `struct` attributes. Accesses to primary variables may be represented simply by the appropriate declaration node, which in that case is directly linked into the syntax tree. The Simula front-end uses REF-nodes to a larger extent, due to the circumstances surrounding the use of manually handled, heap-based frames, as will be presented in chapters 3 and 4.

- The **EXPR** category is the largest one and contains nodes used within syntax trees to represent all kinds of semantic operations ranging from elementary arithmetic operations to program control manipulators like loop constructs and GOTO statements.

To provide some concrete examples to the preceding list, the following description treats five typical types of tree nodes, one from each of the categories:

- The INTEGER_TYPE code is used in nodes representing integer data types. Information like the precision (i.e. bit-width) of the type and whether it is signed or not, is stored within the node or accompanying nodes when the INTEGER_TYPE node is created.

ERROR_MARK	IDENTIFIER_NODE	OP_IDENTIFIER
TREE_LIST	TREE_VEC	BLOCK
VOID_TYPE	INTEGER_TYPE	REAL_TYPE
COMPLEX_TYPE	VECTOR_TYPE	ENUMERAL_TYPE
BOOLEAN_TYPE	CHAR_TYPE	POINTER_TYPE
OFFSET_TYPE	REFERENCE_TYPE	METHOD_TYPE
FILE_TYPE	ARRAY_TYPE	SET_TYPE
RECORD_TYPE	UNION_TYPE	QUAL_UNION_TYPE
FUNCTION_TYPE	LANG_TYPE	INTEGER_CST
REAL_CST	COMPLEX_CST	STRING_CST
FUNCTION_DECL	LABEL_DECL	CONST_DECL
TYPE_DECL	VAR_DECL	PARM_DECL
RESULT_DECL	FIELD_DECL	NAMESPACE_DECL
COMPONENT_REF	BIT_FIELD_REF	INDIRECT_REF
BUFFER_REF	ARRAY_REF	CONSTRUCTOR
COMPOUND_EXPR	MODIFY_EXPR	INIT_EXPR
TARGET_EXPR	COND_EXPR	BIND_EXPR
CALL_EXPR	METHOD_CALL_EXPR	WITH_CLEANUP_EXPR
CLEANUP_POINT_EXPR	PLACEHOLDER_EXPR	WITH_RECORD_EXPR
PLUS_EXPR	MINUS_EXPR	MULT_EXPR
TRUNC_DIV_EXPR	CEIL_DIV_EXPR	FLOOR_DIV_EXPR
ROUND_DIV_EXPR	TRUNC_MOD_EXPR	CEIL_MOD_EXPR
FLOOR_MOD_EXPR	ROUND_MOD_EXPR	RDIV_EXPR
EXACT_DIV_EXPR	FIX_TRUNC_EXPR	FIX_CEIL_EXPR
FIX_FLOOR_EXPR	FIX_ROUND_EXPR	FLOAT_EXPR
EXPON_EXPR	NEGATE_EXPR	MIN_EXPR
MAX_EXPR	ABS_EXPR	FFS_EXPR
LSHIFT_EXPR	RSHIFT_EXPR	LROTATE_EXPR
RROTATE_EXPR	BIT_IOR_EXPR	BIT_XOR_EXPR
BIT_AND_EXPR	BIT_ANDTC_EXPR	BIT_NOT_EXPR
TRUTH_ANDIF_EXPR	TRUTH_ORIF_EXPR	TRUTH_AND_EXPR
TRUTH_OR_EXPR	TRUTH_XOR_EXPR	TRUTH_NOT_EXPR
LT_EXPR	LE_EXPR	GT_EXPR
GE_EXPR	EQ_EXPR	NE_EXPR
UNORDERED_EXPR	ORDERED_EXPR	UNLT_EXPR
UNLE_EXPR	UNGT_EXPR	UNGE_EXPR
UNEQ_EXPR	IN_EXPR	SET_LE_EXPR
CARD_EXPR	RANGE_EXPR	CONVERT_EXPR
NOP_EXPR	NON_LVALUE_EXPR	SAVE_EXPR
UNSAVE_EXPR	RTL_EXPR	ADDR_EXPR
REFERENCE_EXPR	ENTRY_VALUE_EXPR	FDESC_EXPR
COMPLEX_EXPR	CONJ_EXPR	REALPART_EXPR
IMAGPART_EXPR	PREDECREMENT_EXPR	PREINCREMENT_EXPR
POSTDECREMENT_EXPR	POSTINCREMENT_EXPR	VA_ARG_EXPR
TRY_CATCH_EXPR	TRY_FINALLY_EXPR	GOTO_SUBROUTINE_EXPR
LABEL_EXPR	GOTO_EXPR	RETURN_EXPR
EXIT_EXPR	LOOP_EXPR	LABELED_BLOCK_EXPR
EXIT_BLOCK_EXPR	EXPR_WITH_FILE_LOCATION	SWITCH_EXPR
EXC_PTR_EXPR		

Table 2.1: All 145 tree codes defined by the back-end of GCC version 3.0.2

- An INTEGER_CST node, stores integer constants which often stem from integer literals in the source program. An INTEGER_CST node is always associated with an integer data type by a reference to an INTEGER-_TYPE node. This association is particularly significant when performing type-checking during semantic analysis.

- A VAR_DECL node is used to represent a global or local variable declaration. Like the CST node, the declaration node contains a reference to a data type node (tree) which specifies the type of the declaration. Additional information about the declaration, like its name and containing scope, is represented by references to the nodes that represent these entities.

- An INDIRECT_REF node is used to represent dereferencing of a pointer. The node references another node (possibly a root in a tree) that represents the memory address to be accessed. This may turn out to be an arithmetic expression (cf. pointer arithmetics in C) or another indirect reference to somewhere where the address value is stored. Also, the INDIRECT_REF node links to a data type node to specify the desired interpretation of the memory contents found where the address refers. This is analogous to explicit type-casting and is necessary since the back-end does not try to deduce the type associated to a node in the syntax tree, but rather expects that the front-end's semantic analyzer does this in conjunction with type-checking.

- A PLUS_EXPR node is one of many arithmetic expression nodes. It signifies a binary addition operation and, hence, references two other nodes or sub-trees, a left-hand one and a right-hand one. As with INDIRECT_REF, a PLUS_EXPR node has to be associated with a data type which specifies the desired storage representation of the calculated sum. The Simula specification states, for instance, that summation of an integer and a floating-point expression should be carried out and represented using a floating-point data type. Other languages may have other rules, so the back-end simply expects that all the syntax trees provided by the front-end, have complete type specifications.

A front-end may extend the set of tree codes for its own convenience, but must ensure either that all tree nodes using these extended tree codes, are replaced by nodes using only the basic set, or alternatively, that a *callback function* (also referred to as a *hook*) is defined so that the back-end can activate that function when encountering a node with an extended tree code. If none of these measures are taken, the back-end will issue an error message and fail upon encountering anomalies in the tree structures given to it.

As emphasized in the above description, tree nodes often reference data type specifications. In fact, it is so common that the type pointer attribute is defined as a part of all tree nodes, as shown in listing 2.2.

The listing also includes a third attribute—the tree node pointer chain. This attribute is used in cases where it proves convenient, for whatever reason, to keep nodes organized in a chain (linked list), which is a situation that tends to arise quite often. Nodes representing attributes in a compound data type, arguments in a function profile, or FOR-list elements of a FOR statement, are all organized in chains using the chain attribute.

In addition to the three attributes described here, there are quite a few Boolean (1-bit) flag attributes, like the side_effects_flag, which, in case of an EXPR node, indicates whether evaluation of the expression (or any nodes beneath it in the syntax tree) may result in side effects. Typically, this is true for CALL_EXPR nodes, which represent function activations, and MODIFY_EXPR nodes, which represent assignments. Another flag, called constant_flag, indicates whether the expression represented by the node, will return the same value each time it is evaluated, a fact that may be utilized when optimizing the code.

Identifier nodes

The purpose of an IDENTIFIER_NODE node is simple. Such a node represents an identifier, meaning that it contains a single textual string. The creation of identifier nodes is performed by a back-end function which maintains a complete survey of all identifier nodes already created. This is done to ensures that a text string is represented by only one (unique) identifier node. If a certain identifier node already exists, that node will be returned anew.

This property of the identifier nodes, is convenient because it enables textual equality to be determined solely by comparing the addresses of the two operands, which represent the same identifier if and only if they are the same node (i.e. if and only if the operands' addresses are equal).

Data type structures

The term *data type tree* has been mentioned already. The purpose of a TYPE node is to define the properties of a data type used by the source language. The data type can be either simple or compound.

The simple data types are the ones that are not composed of other data types, but instead defines the amount of memory occupied by a variable of the type. Simple data type nodes include INTEGER_TYPE, REAL_TYPE, ENUMERAL_TYPE, BOOLEAN_TYPE, CHAR_TYPE, POINTER_TYPE, and REFERENCE-_TYPE.

Compound data types are constructed by combining other data types.

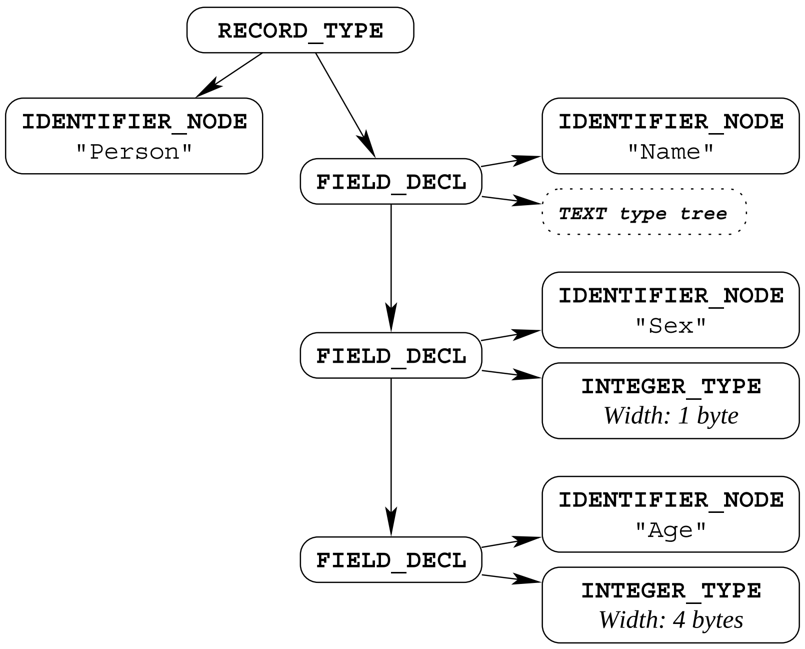

Figure 2.2: A tree representing a compound data type

Typical compound data type nodes are ARRAY_TYPE, RECORD_TYPE and UNI-ON_TYPE.

Figure 2.2 demonstrates the tree structure used to represent a compound data structure (cf. a C struct) containing three fields; Name, Sex, and Age, which are of text type, character type and integer type, respectively. The figure does not include the tree structure used to represent the text type, which may be a pointer type to a character array, or something similar.

Although the current version of GSC does not support classes, the structure shown in figure 2.2 indicates a possible way in which the front-end may construct the representation of a Simula class, like the one shown in listing 2.3.

Listing 2.3: A simple Simula class declaration

```
CLASS Person;
BEGIN
  TEXT Name;
  CHARACTER Sex;
  INTEGER Age;
END;
```

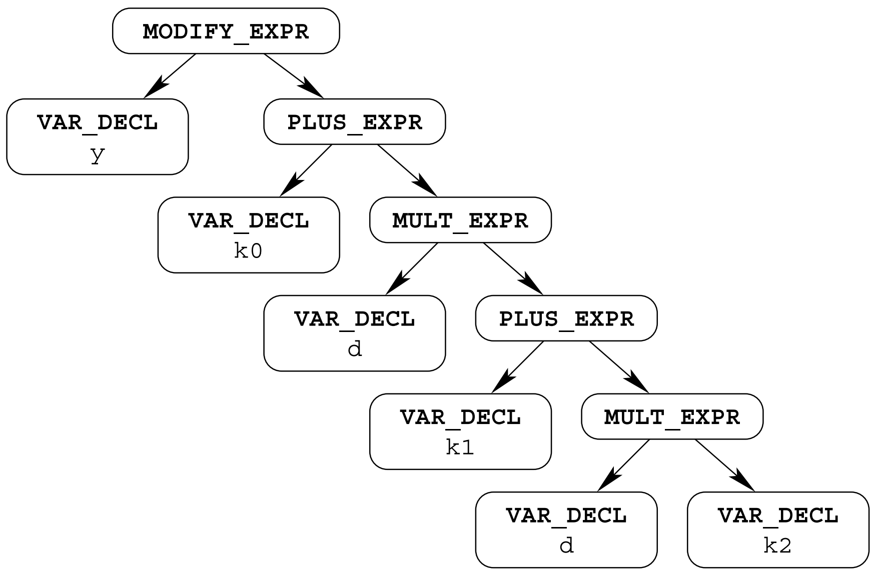

Figure 2.3: A syntax tree representing an arithmetic expression

Declaration structures

A declaration is represented by a tree node which specifies the declaration's name (by means of a pointer to an identifier node) and its type (by means of a pointer to a data type tree). Additionally, a declaration node may reference other nodes, like a node representing the containing function or block, if any.

Typical declaration nodes are `TYPE_DECL`, `VAR_DECL`, `PARM_DECL`, `RESULT-_DECL` and `FIELD_DECL`.

Syntax trees

Among the most essential tree nodes are the ones that are used to represent the actual statements and expressions of the source program. These nodes are used to construct *syntax trees* which, in principle, comply to the long-standing, traditional structure described in compiler science. This is where the `EXPR` category is relevant, but nodes from other categories, like `DECL` and `REF`, may also be part of syntax trees.

In the GSC front-end, the skeletons of the syntax trees are built up by the parser. The semantic analyzer then revises the syntax trees, which includes exchanging identifier nodes with nodes that represent actual entities, as well as adding data type specifications to the nodes of the syntax trees.

Figure 2.3 shows how the statement `y := k0 + d * (k1 + d * k2);` might be represented.

Miscellaneous nodes

There are a few tree codes that do not fit into a larger category. One such code is the `ERROR_MARK`, which is used by nodes that indicate error situations. The semantic analyzer inserts such a node into the syntax tree to indicate that an error has been recognized and a message has been issued. When the recursive analysis recedes back, up the syntax tree, the error-mark node will induce new error-mark nodes, effectively turning the entire tree into error-mark nodes. The existence of an error-mark node in the tree, will indicate that an erroneous situation has already been recognized and a corresponding error message has been issued. This will enable the semantic analyzer to refrain from issuing multiple messages stemming from the same or a related error situation.

Two other miscellaneous tree codes are the `TREE_LIST` and `TREE_VEC` nodes. These are typically used to arrange other nodes in certain patterns. A `TREE_LIST` node is used to create linked lists of nodes. As mentioned above, all tree nodes contain a `chain` attribute, which enables them to be part of a chain of nodes. However, if a node is to be part of several chains, these chains have to be built up of `TREE_LIST` nodes, each of which references a node that should be part of the chain. A `TREE_VEC` node contains an array (vector) of node pointers. The `TREE_LIST` and `TREE_VEC` nodes are used frequently, and in some contexts they are expected by the back-end.

2.5.2 Generating code

Code generation is usually invoked by calling one of several *expand* functions in the back-end. The term “expand” signifies that the tree structure provided to the function, is converted into the RTL representation.

The front-end is free to choose the size of the tree structures sent to the back-end. The back-end will accept each piece and generate RTL-code according to order in which the syntax tree pieces are delivered.

A front-end typically transmits one tree structure per statement. If more than one statement is contained in a structure, the front-end must make sure that the back-end does not regard the tree as only one statement, and so, rearranges the evaluation order in an attempt to optimize. Simula also has requirements on evaluation order within individual statements, so extra measures will have to be taken to ensure that things are done in the appropriate order.

On the other hand, if the front-end chooses to transmit very small tree fragments at a time, or, alternatively, generates the RTL-code itself, this may degrade optimization because the strict ordering which is being indicated indirectly in that case, by the sequencing of the small fragments, may pose unnecessarily rigid constraints.

So, this is another testimony to the versatility of the back-end interface.

Either, a few large tree structures may be transmitted in a few back-end invocations, or, alternatively, many small tree structures may be transmitted in many invocations. The most appropriate approach may depend on the source language or other circumstances.

When the tree structures of an entire function has been transmitted, the front-end instructs the back-end to perform optimization and code generation. After that, control returns to the front-end which may then transmit the next function.

This way, each function is treated as a separate unit and optimized individually in the back-end. More appropriately, each such function should be referred to as a *compilation unit*, because they do not necessarily have to correspond to logical units of the (high-level) source language. In C, however, they usually do, and each unit correspond to a C function. The way GSC divides Simula programs into compilation units, will be discussed in chapter 4.

Chapter 3

Simula

This chapter presents the Simula programming language, with particular attention to the aspects whose requirements differ from those of the C programming language. The facilitation of quasi-parallel sequencing which is required by Simula coroutines, turns out to be the most non-trivial. Hence, this matter is devoted much attention in this chapter as well as the rest of this report.

3.1 Background

Simula is the programming language that introduced what has later become known as object-oriented programming, including classes, inheritance, and virtual procedures. Since its conception in the 1960s, the language has had substantial influence on programming language research from which well-known object-oriented programming languages like C++ and Java have evolved.

The original motivation behind Simula stemmed from the desire to develop a more powerful language for use in implementations of discrete event simulations. This fact is reflected in its name, which is an abbreviation for ***Simu****lation* ***La****nguage.* Simula also includes a feature known as *coroutines,* which is particularly aimed at facilitating discrete event simulations. Compared to the object-oriented ideas introduced by Simula, coroutines have not received the same amount of attention. An important reason for this, probably stems from the extra complexity that coroutines impose on activation record organization, as will be discussed in the following section.

3.2 Coroutines

The coroutine mechanism enables several class objects to execute in *quasi-parallel*, i.e. they do not execute simultaneously, but execution may alternate between two or more coroutines and give the impression of parallelism. At first glance, this may seem comparable to features like threads in certain other languages (e.g. Java), which also provides parallel or quasi-parallel execution. There are, however, fundamental differences between the two concepts, which lead to preference of quite dissimilar implementation methods.

Simula coroutines are *cooperative*, meaning that coroutine alternations are controlled explicitly by executing certain program statements. Furthermore, a coroutine does not come into existence until after the procedure and object activations that will constitute the coroutine, have been performed. Consequently, the corresponding activation records (frames) have already been created when they become part of a coroutine.

A coroutine is created when, inside an object activation (i.e. a `NEW` call), the program execution encounters an activation of the object's `DETACH` method. This method makes the object's activation (including all dynamically enclosed procedure and object activations) suspend temporarily, and moves program control to the statement succeeding the object activation statement (i.e. the `NEW` call). Usually, such a detached (or suspended) object will be reactivated later by an appropriate invocation of one of the system routines `CALL` or `RESUME` which take a reference to the object as parameter and direct program control back to where the `DETACH` was activated as if it has returned like an ordinary routine. If, later, `DETACH` is activated again, program control will alternate back to where the `CALL` or `RESUME` procedure was activated.

The main challenge posed by coroutines is to determine how activation records (from now on referred to as *frames*) are to be organized. The traditional technique, where frames are organized by using a stack, proves to be inadequate because a frame may last longer than some of the frames that initially dynamically enclose it. Stated another way, a coroutine may be reactivated within (i.e. be reattached to) another dynamic context than the one from which it was suspended (or detached).

This makes Simula coroutines differ fundamentally from Java's threads. Also, coroutines should be expected to far outnumber threads in a Java program. These facts strongly disfavors the method usually employed by Java implementations, where each thread is given a reasonably large address-space designated to its execution stack.

So basically, the stack methodology (i.e. using a continuous stack-organized memory area for allocating frames) is inadequate for Simula programs that utilize coroutines. As pointed out by Dahl [6], a plausible solution is to allocate frames on the heap. This way, frames become independent

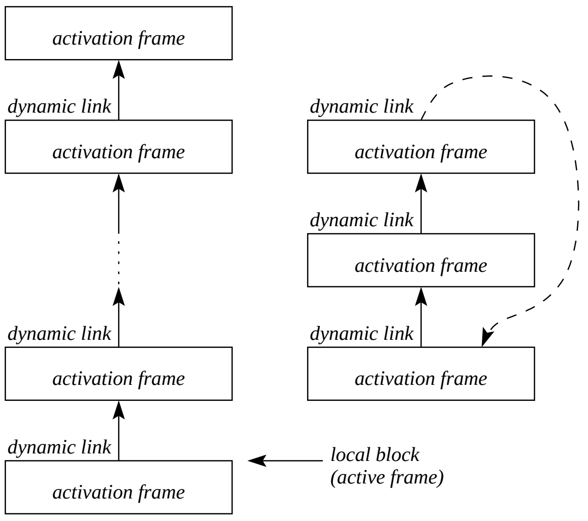

Figure 3.1: The operating chain and a detached coroutine chain

of each other can be organized and reorganized easily during the course of a program's execution. The chain containing the active frame (i.e. the frame corresponding to the currently executing block instance), is always the same chain as the one that contains the frame of the outermost block instance. This chain is commonly referred to as the *operation chain*. Detached coroutines make up loose chain fragments. Figure 3.1 illustrates such a situation, including the operating chain and a detached coroutine chain. To indicate where to resume execution when a coroutine is reactivated, the dynamic link and exit address of the outermost frame (dynamically) of the coroutine, are used to indicate the reactivation point. Hence, a detached coroutine forms a circular chain of dynamic links. Also, since only class objects may be detached, the outermost frame always corresponds to an object instance.

The traditional stack organization of frames is regarded so fundamental that modern CPUs have designated registers and instructions for operating such a stack. Hence, the abandoning of the stack methodology turns out to constitute a significant impact on the design of the Simula front-end and its interaction with the back-end, especially since the back-end has an inclination toward stack-based execution. This is a main reason why implementing a Simula front-end to GCC presents an interesting endeavor.

3.3 Cim

Simula has been implemented on a number of machine platforms, by different vendors and research communities. One of the latest implementations is called Cim and uses the C programming language as intermediate language in the compilation process. Hence, Cim effectively provides Simula support to the same wide range of target machines that is supported by C. At the same time Cim profits from the effort put into code optimization in the underlying C implementation.

Being relatively low-level, the C language proved to provide an acceptable basis for intermediate representations of Simula programs. C, however, lacks some expressive power relating to a few aspects of the translation of Simula programs. This forced the Cim developers into resorting to some ad-hoc techniques, having negative impact on code quality and efficiency as well as structural elegance.

More specifically, C lacks a mechanism for obtaining the address of a program label. This compels Cim into using a `switch`-statement where each `case` contains a `goto` statement to a particular label. Effectively, this enables a program to treat program labels as variables and parameters, by using the corresponding integer which is uniquely mapped into a particular program label by the `switch`-statement. Furthermore, C does not offer a mechanism for obtaining the relative address of a particular field of `struct`s and `union`s.

Simula also has some properties that could be exploited by code optimization, but which get hidden in the transition to C. For example the absence of general pointers and pointer arithmetics in Simula reduces pointer aliasing. In C, however, the extensive generality of pointers forces the code optimizer to be more careful because a memory location may be accessed through different pointers. In Simula, two pointers or references either reference the same object, which means that they are identical, or they reference different objects which means that the will be used to access completely disjoint memory areas.

For an ordinary Simula program which consists of one single source file, the C code that Cim produces, consists mainly of `struct` definitions and a `main` routine that contains the semantics of the entire program. Cim does not map each Simula procedure and class into individual C functions. This is because the implied utilization of the machine stack during function activations in C, must be avoided. The code produced by Cim, takes care of procedure and object activations manually, by explicitly allocating storage for parameters on the heap and then activating the procedure or object by an appropriate `goto` statement. This way, Cim does not impose activity on the machine stack.

The structure of Cim and its run-time system, have provided important guidelines in the construction of GSC.

3.4 The run-time system

The Simula run-time system is quite extensive. It includes support for implicit garbage collection as well as standard input/output capabilities for reading from and writing to terminal and files. There is also a set of mathematical procedures included in the run-time system.

The run-time system can be divided into two parts. The first contains the functionality that may be implemented using the standard Simula syntax and semantics. The second part consists of the mechanisms which communicate with operating system or which handles the structures behind the scenes. The second part includes functionality to check that arrays are accessed within their bounds. It also includes mechanisms to determine the class membership of class objects as well as class inheritance properties.

These are the mechanisms that lie behind operators like `IS`, `IN` and `QUA`. When programming in a high-level language like Simula, one often forgets that these, seemingly bashful operators not only rely on moderately complex routines being activated at run-time, but also that they require that information about class inheritance is somehow stored by the compiler in a static data area of the program in order to be available at run-time.

3.5 Memory management and garbage collection

The mechanisms concerning memory management and garbage collection, are also examples of parts of the run-time system, which to some extent have to cooperate with the compiler.

Memory management in Simula can be implemented in several ways as long as it behaves like a heap in the sense that objects of varying size can be allocated and freed in arbitrary order. If the structure and organization provided by the compiler and run-time system in general, permit it, a memory manager may be able to move objects around in memory, so that their addresses change. For this to be possible, the memory manager needs to be able to locate and update all affected references.

Moving of objects may be relevant when performing garbage collection. As in other languages, like Java, garbage collection is the process of locating objects that are no longer reachable from any variable in the executing program, and deallocating these objects so that the memory they occupy may be used by subsequent allocations.

Garbage collection includes analyzing all potential references to objects, so that they can be marked as reachable. The amount of information that the compiler provides to the run-time system's garbage collector, can be divided into three categories:

1 **Exact information** The compiler provides the garbage collector with complete information about all references in all objects.

2 **Approximate information** This means that the positions of all potential references are accounted for in the information given by the compiler. However, all the positions need not represent references at all times.

3 **No information** The compiler does not generate information about reference positions at all. This effectively implies that the entire interior of all objects, must be considered to contain potential references.

The garbage collector which is part of the run-time system used by Cim, requires that Cim conforms to case 1. The reason is that the garbage collector moves allocated objects around in memory (in order to diminish fragmentation of free space). Hence, the garbage collector has to know exactly whether a memory position is a reference or not, so that it can update exactly all affected references.

Case 2 will occur if an allocated object changes its profile, i.e. if memory positions inside it are used to store different kinds of entities throughout the object's lifetime. As long as the information provided by the compiler, is conservative and includes all potential references, no references will be forgot when the garbage collector performs its inspection. This is no problem as long as the garbage collector accepts that a reported reference position may not point to an object since it may not represent a reference at that time. Case 2 also implies that objects may not be moved around, so the memory manager may need some other means to control the amount of fragmentation.

Case 3 also causes no trouble for a garbage collector that does not move the objects around. The main difference from case 2 is that case 3 is generally less efficient both when it comes to collecting garbage as well as the execution speed, because more memory positions have to be inspected and these may inadvertently point to other objects even though they do not represent references at the time. This scheme is called *conservative garbage collection* [10]. The percentage of uncollected objects is usually bounded by a small value.

Chapter 4

Design

This chapter brings the discussions of the preceding two chapters, together and outlines the most important aspects of the structural design of the Simula front-end.

4.1 GSC's relation to Cim

In principle, the combination of Cim and the C front-end of GCC, constitutes a Simula front-end. However, when regarded as such a unit, it is fairly evident that Cim and the C front-end do not interact optimally with each other. The C language, which connects the two, has limited flexibility when it comes to expressing semantic properties of Simula programs, and it also reduces the compile-time efficiency due to C needing its own syntactic analysis.

The penalty associated with generating and parsing intermediate C source code, could be avoided simply by introducing appropriate connections between the C code generator of Cim and the semantic analyzer of the C front-end. Although, such a "layer of glue" between Cim and the C front-end, would increase the compile-time efficiency, it would still be confined by the same restrictions as C, and therefore would not be able to utilize the inherently more powerful interface that the GCC back-end provides.

Still, the combination of Cim and the C front-end of GCC is evidence that a fairly efficient front-end producing fairly efficient code, can be constructed. This realization led my attention toward trying to aim for better efficiency from the very beginning. Hence, my objective for implementing GSC, was to explore and, if possible, utilize the benefits of not having to relate to C as Cim does. A consequence of this is that the GSC implementation is not based directly on the source code of Cim. However, many of the techniques employed in GSC, have been adopted from Cim more or less unchanged.

4.2 Activation frames

As introduced in chapter 3, a *frame* is an area of storage, which stores information associated with a *block instance*, i.e. one particular activation of a block. Note that the term “block instance” is an abstract term, while “frame” is its physical manifestation in memory at run-time.

Due to the existence of coroutines, a block instance may live longer than block instances that, initially, enclose it dynamically. Consequently, frames cannot form a stack, but instead have to be allocated on the heap.

The GCC back-end does not offer built-in support for heap-based frames, so the front-end has to determine the layout and compute the space requirement of the frames, itself. It also has to generate the run-time semantics for allocating the frames and accessing the attributes stored therein. The back-end merely regards such heap-based frames as ordinary heap-allocated objects like class objects or other data types (e.g. C `structs`) that have been allocated using `malloc` or a similar memory manager.

In a run-time environment that can manage solely with a stack of frames, the active frame is always the one at the top. This enables it to grow and shrink easily through the course of its lifetime because it is located next to the free memory area above the stack. If a new frame is created on the top, it may always be positioned according to the temporal size of the underlying frame. This effectively eliminates any gap of unused space from arising between the frames.

Heap-based frames require more planning and care. Due to the inherent restrictions of heaps, a heap-based frame is not generally free to grow during its lifetime, unless it is moved to a new location at the same time. It may shrink, but the space that is vacated is likely to be too small to be of any use to another frame, again, unless the frame is also moved somewhere else.

So, if a frame is allowed to move, it may also be resized by allocating a new area of appropriate size, moving its contents, and finally freeing the old area occupied by the frame. Such operations are, however, costly and will induce significant run-time overhead if they are to be performed every time space have to be added to or removed from a frame. Also, if a frame is allowed to move, all referencing pointers will have to be updated.

The conclusion drawn in conjunction with the GSC implementation, is that frames should remain at fixed locations, which in turn means that resizing is impossible or futile. Either, additional space has to be allocated in new independent objects, which is a typical technique employed by many traditional Simula implementations, or the compiler has to deduce the maximum space required by a frame for its entire lifetime, so that this space can be *preallocated*, as will be discussed in section 4.2.2.

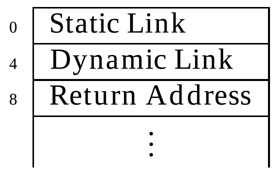

Figure 4.1: Activation frame header

4.2.1 The frame header

The frame headers laid out by GSC, are, in principle, traditional. However, GSC's experimentation with preallocation of *parameter frames* (cf. section 4.2.2), makes the frame headers end up in odd places.

Unlike standard C, Simula supports nested blocks, which means that all block instances need a *static link* (*access link*). However, analogous to C, Simula block instances need a *dynamic link* (*control link*) and an exit address. These entities are typically stored at the top of the frame—the *frame header*, as shown in figure 4.1. The static link is positioned at the very top to facilitate efficient traversing of the static chain (cf. section 4.11).

Dahl [6] describes, and several classic Simula implementations include, a fourth header element—a *prototype pointer*. The prototype pointer identifies a compiler-generated (static) data block which determines certain properties of the corresponding frame. The existing GSC implementation does not produce prototype blocks and, hence, does not reserve space for a prototype pointer.

4.2.2 Preallocation

As introduced above, a frame being allocated at run-time, may be given a size that exactly accommodates the space requirement at the time of allocation. If the block instance, to which the frame belongs, subsequently needs additional space (typically in order to store temporaries), this additional space has to be provided, either by means of additional frames, or by somehow resizing the original frame. This will keep memory utilization tight, by not tying up more memory than needed at any instant.

An alternative approach is to initially allocate the amount of space that equals the maximum amount required by the block instance for its lifetime. This will possibly waste some memory, but instead minimize the number of allocations required.

As indicated in the introduction chapter, a main focus has been to favor time-efficiency because time is typically regarded as a more precious quantity than memory, which tends to be relatively spacious on computers nowadays. The superior objective is therefore to reduce the number of allocations in order to save time spent on heap allocations and maintenance.

This is achieved by utilizing the technique whose name has already been introduced—*preallocation*. This technique implies allocating space that is not immediately required, but may become needed during the course of a block instance's lifetime.

The following sections describe the three types of preallocations that have been implemented in GSC.

Preallocation of temporaries

Evaluation of expressions may result in temporary values which needs to be stored in memory and later retrieved in order to complete the computation. In GSC, only certain temporaries have significance to the front-end and need to be set aside in heap-based frames. These are the temporaries that are live during procedure or object activations, because such activations may lead to coroutine switches which may possibly invalidate the temporaries if they are not stored in the heap-based frame. Preallocation involves accounting for the space needed by such temporaries when allocating the frame belonging to the block that requires the temporary.

Other implementations, like Cim, use a global stack to store temporaries. When a procedure or object is activated, the current content of the stack is moved into a designated object which is allocated on the heap. Such an object is referred to as an *accumulator stack object* [6, 8, 9].

As indicated above, the main motivation for deviating from such a scheme, is to keep the number of allocations at a minimum. As pointed out by Dahl [6], the number of such temporaries is often zero or otherwise small, thereby constituting a small impact on the total size of a frame.

It is also significant to note that temporaries whose lifetimes are disjoint (e.g. temporaries stemming from different expressions), can share the same space within the frame.

Temporaries that are not live during procedure or object activations, are of no concern to the front-end. Such temporaries either resides in machine registers during their entire lifetime, or they are *spilled* out to the global machine stack. These temporaries are allocated and organized solely by the back-end.

Listing 4.1 shows a program which uses a temporary to store the value of the variable `I` while performing a call to a procedure `P`. This is necessary because Simula requires operands to be evaluated in left-to-right order as explained in section 4.5. Figure 4.2 demonstrates how space for this temporary, *t0*, is preallocated in the frame. The figure also refers to the term "parameter frame", which is described shortly.

Listing 4.1: A temporary copy of `I` is needed during activation of `P`

```
BEGIN
  INTEGER I, J;

  INTEGER PROCEDURE P (A, B); INTEGER A, B;
  BEGIN
    ⋮
  END;

  ⋮
  J := I + P (9, 16);
  ⋮
END;
```

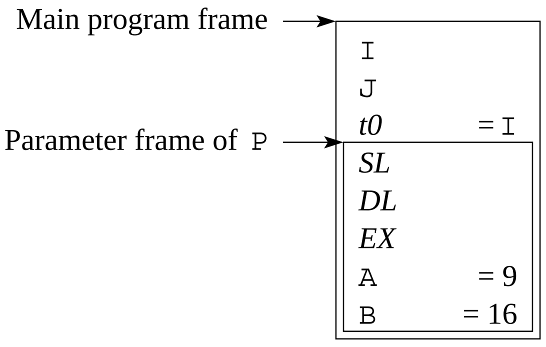

Figure 4.2: Frame containing a temporary value *t0* which is a copy of `I`

Preallocation of sub-blocks

A sub-block may have its own local variable declarations and may therefore need a frame to store both these variables and also any temporaries that are needed by the constituent statements. If memory consumption is the primary concern, a sub-block's frame should be allocated if and when the sub-block is entered and then be deallocated when it is exited.

A sub-block's activation is always dynamically enclosed by its containing block's corresponding activation. Consequently, the lifetime of a sub-block instance is always contained within the lifetime of its containing block's instance. There is also never more than one active sub-block instance. This makes it possible to preallocate a sub-block's frame within the containing block's frame, which is a technique described by Dahl [6].

A preallocated sub-block frame behaves like a compound temporary in the containing frame. The space occupied by the sub-block frame, may be shared with other statements (including sub-blocks) in the surrounding block (i.e. on the same level as the sub-block).

In GSC, preallocation of sub-blocks are performed recursively, i.e. sub-blocks within sub-blocks are preallocated within preallocated frames, and so on. In particular, if the controlled statement of a `FOR` statement is a sub-block containing local variables, the space occupied by these local variables, is preallocated once and for all within the containing block's frame, before the `FOR` statement is activated. This is particularly beneficial when the controlled statement is repeated many times, because that will save many heap-allocations. It is, however, still necessary to reset the local variables to zero prior to each iteration of the controlled sub-block, because Simula requires that local variables are initialized to zero.

Preallocation of parameter frames

The most untraditional type of preallocation done in GSC, concerns what is called *parameter frames*. This technique should be regarded more as an experimental curiosity and its actual implication on execution speed, has not been investigated thoroughly.

Preallocation of a parameter frame implies that the space used to keep the procedure parameters and return value being transmitted between the calling routine (*caller*) and the procedure being called (*callee*), is preallocated in the frame of the caller. In addition to the parameters and the return value, this area needs to store the static link, dynamic link and exit address of the callee, all of which are initialized by the caller before activating the callee.

Figure 4.3 shows the structure of the frames when the execution is within the `Q` procedure of the program showed in listing 4.2. It is worth noting that `Q` needs no local frame, and consequently refrains from allocating one.

The parameter frame conceptually introduces an extra (fictitious) block-level to which the parameters are bound. Since the actual procedure body is a proper sub-block of this "parameter block", the dynamic and static links of the procedure body's frame, will always coincide and may therefore be represented by the same pointer. Furthermore, no exit address is needed because the procedure body, which becomes a proper sub-block, implicitly exits to the containing, fictitious, parameter block.

As noted above, this technique is experimental. Its advantages and disadvantages have not been fully elucidated. However, some possible positive effects are listed here:

- Preallocation reduces the number of heap-allocations if the alternative is to allocate an object specifically for the parameter block. If, however, the parameter frame is to be allocated within the frame of

Listing 4.2: Program performing a call to `P` which calls `Q`

```
  BEGIN
    INTEGER I, J;

    INTEGER PROCEDURE Q (C, D); INTEGER C, D;
    BEGIN
      Q := C * D;
⇒   END;

    PROCEDURE P (A, B); INTEGER A, B;
    BEGIN
      INTEGER K, L;

↪     Q (A + B, A - B);
      ⋮
    END;

↪   P (9, 16);
    ⋮
  END;
```

the callee, the caller either needs a way to acquire the information on the amount of space to allocate for the callee, or it has to activate the callee which then allocates the required space and then inquires the caller about the actual parameter values.

- Simple procedures that have no local variable declarations and need no space for temporaries, will not induce a frame allocation at all.
- Procedures that perform many calls, can have a single preallocated area for the parameter frames within its own frame, rather than having to allocate the space in conjunction with each activated procedure's frame.

There are also some possible negative aspects of the parameter frame preallocation scheme:

- The static/dynamic chain of the callee's body will point to somewhere in the interior of the caller's frame, which may complicate the search for garbage by the garbage collector.
- The introduction of an extra (fictitious) block-level will introduce an extra static link to be dereferenced when accessing outer entities.

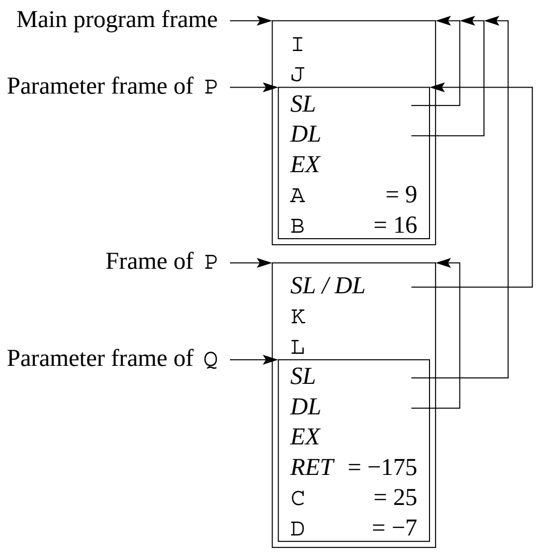

Figure 4.3: Internal structure of frames in conjunction with procedure calls

- The space preallocated for the parameter frame, may constitute unnecessary wasting of space in the caller's frame.
- The call/return sequence of procedure activations deviate from that of class object activations, since the parameters of a class object cannot reside within the dynamically enclosing frame, because an object may be detached from it. Consequently, procedure and object activations will use different schemes.

4.3 Procedure activations

The preallocation techniques described above, leads to the creation of only one frame per procedure activation. A frame is allocated by the *activation sequence* which is executed when the procedure is activated.

The GCC back-end knows the concepts of routines and routine activations. However, the activation sequences generated by the back-end, are stack-oriented. This is acceptable for many languages, but Simula procedure activations have to be heap-oriented. Effectively, this leads to the GSC front-end having to "deceive" the back-end in a way that either makes the back-end refrain from generating stack-oriented activation sequences, or somehow generates code that provide circumvention of these automatically generated, stack-oriented activation sequences.

The caller's part of the activation sequence consists of allocating the parameter frame (if not already preallocated), and then initializing its fields. The following list gives a more detailed description of how these fields are initialized:

- First, the actual parameters are evaluated and stored in their appropriate positions with respect to the parameter profile (or signature) of the procedure being activated.
- The caller determines the static context of the callee and sets the static link of the parameter frame, appropriately.
- The dynamic link is set equal to the currently local block pointer.
- The exit address is set to the program point following the actual jump operation which activates the callee.
- The local block pointer is set to point at the parameter frame.
- The caller then activates the callee by a jump instruction to the starting address of the callee.
- When the callee returns to this point, the local block pointer is set back to the old value which was stored in the dynamic link of the parameter frame.
- The return value, if any, is extracted from the parameter frame.

4.4 Compilation units

As described in chapter 2, the front-end instructs the back-end when to optimize and generate code from a series of tree structures. GSC may either do as Cim does and generate one large compilation unit (cf. one large C function), or it may split the source program into smaller units which are optimized individually by the back-end.

The most significant motivation for splitting a Simula program into several compilation units, is to enable resource-effective optimization. A Simula procedure represents a natural optimization unit. If instead an entire Simula program is combined into one large unit, the back-end will use a disproportionate amount of memory and computation effort to perform optimizations, since the time-complexity of the back-end's optimization mechanisms, is super-linear.

Another reason to generate one compilation unit per Simula procedure, is that an externally visible assembler-level symbol is associated with each unit, and will provide for easy linking between program pieces that come from different source files.

On the other hand, since the GCC back-end produces standard prologues and epilogues in conjunction with each compilation unit, these have to be disabled or circumvented, because they tamper with the global stack. It is easier to control this if the back-end is simply not aware of the procedure boundaries of Simula.

Still, I chose to pursue the alternative that I believe may ultimately lead to the best and most effective solution once the issue of disabling the back-end-generated prologues and epilogues, has been solved. That is to generate one compilation unit per logical code unit, i.e. per procedure and class body.

Listing 4.3 shows a sample Simula program which includes a class declaration and two procedure declarations. Listing 4.4 demonstrates by using the C language, how GSC divides the elements of the Simula program into several units (cf. C functions) which are treated individually in the GCC back-end. The listings are not intended to illustrate any form of correspondence between Simula procedure parameters and C function arguments, but merely illustrates the fact that the hierarchical structure of a Simula program is flattened and that names are given in a manner that will uniquely identify the components in the flat name-space of the assembler and linker.

Listing 4.3: Sample Simula program

```
BEGIN
    CLASS C;
    BEGIN
        PROCEDURE P1;
          ⋮

        P1;
    END;

    PROCEDURE P2;
    BEGIN
        NEW C;
    END;

    P2;
END;
```

Listing 4.4: C code illustrating how GSC splits a program into units

```
void C_P1 (void);
{
    ⋮
}

void C (void);
{
    C_P1 ();
}

void main_P2 (void);
{
    C ();
}

void main (void);
{
    main_P2 ();
}
```

4.5 Evaluation order

The Simula specification states that the list of actual parameters transmitted to a procedure or class object, are to be evaluated from left to right. In binary expressions, the left-hand sub-expression is to be evaluated prior to the right-hand one. The C specification does not state any of this, and leaves this issue to be determined by each implementation. In good C tradition, this provides for more flexibility for the compilers in choosing the optimal evaluation order on any particular machine platform.

Hence, in order not to undermine the laxity of languages like C and potentially inhibit optimization opportunities that may arise thereof, the GCC back-end does not necessarily favor one particular evaluation order when presented with a syntax tree containing an arithmetic expression like, say, a `PLUS_EXPR` top node and two sub-trees representing each operand. Consequently, GSC has to remedy this situation in some way, typically by restructuring the syntax tree or by sequencing through multiple RTL expansions.

Evaluation order is relevant only when evaluation of the operands may interfere with one another, i.e. when the evaluations have *side-effects*. Ideally, the compiler should try to determine the possibility of interference and only bother with the constrained sequencing when such a possibility exists. In practice, compilers are more coarse and consider all assignments and procedure calls as having potentially harmful side-effects.

4.6 Elementary data types

The front-end has to specify the properties of Simula data types. The front-end specifies the width (in bits and bytes) of the elementary *value types* (`BOOLEAN`, `INTEGER`, `SHORT INTEGER`, `REAL`, `LONG REAL` and `CHARACTER`). The back-end takes care of reserving enough space for each data type. When laying out the structure of frames and other compound structures, the back-end will decide the position of the constituent elements based on size and preferred alignment.

The `BOOLEAN` data type has a bit-width of 1. The back-end chooses how to align a variable of this type and whether to compact several adjacent Boolean variables into each byte, e.g. in a Boolean array. The front-end specifies that only one bit is significant, so the back-end is free to do this.

The front-end specifies that the bit-widths of the `INTEGER` and `SHORT INTEGER` data types are 32 and 16, respectively. These types are signed. Again, variable alignment is decided by the back-end to best suit the machine platform in question.

The `REAL` and `LONG REAL` floating point data types have a precision of 32 and 64 bits, respectively.

The `CHARACTER` data type is specified as an 8-bit unsigned entity.

4.7 References

Simula references are stored the same way as C pointers. The front-end uses the `POINTER_TYPE` tree code to represent references. This leaves the back-end to decide the bit-width of these references based on what is required by the address space of the underlying machine platform.

Reference assignments may have to be accompanied by a check at run-time to verify that the object being assigned to the references, belongs to the class (or a sub-class thereof) that qualified the reference declaration. The qualification information itself is however determinable solely at compile-time, and so a reference variable need not comprise more than the actual address value. An object's class membership is determined from its prototype pointer which is part of the object's frame.

4.8 Arrays

An array is a compound data type which is based on the elementary data type that make up the array elements. As discussed by Dahl [6], array variables are best handled in separately allocated frames. The only thing that is left in the frame of the block that owns the array, is the array frame's address, i.e. a pointer to the array frame. When producing code for an array access, the front-end will generate an accompanying check that verifies that the access is legal with respect to the the number of dimension and the bounds of the array. Since the number of dimensions and the lower and upper bounds can be set dynamically, this information has to be included, typically within the array frame (*dope vector* [6]).

The final structure of arrays has not been decided and array handling has not been implemented. It appears more or less straightforward to employ a scheme based on Cim's array handling.

4.9 `TEXT`

The `TEXT` data type has not been implemented in the current version of GSC. A `TEXT` variable may principally be represented internally in two ways.

- It may be created and treated as class object with its own fixed-size, dynamically allocated area containing its attributes, `START`, `LENGTH` and `POS`, in addition to `OBJ` which is a reference to a varying-size, dynamically allocated area containing the actual textual content. This will make a `TEXT` variable consist of two individually allocated pieces.
- In order to avoid that, the attributes `OBJ`, `START`, `LENGTH` and `POS` may instead be incorporated into the local block's frame. Consequently, only the characterăarray holding the textual content, will be allocated separately. This will however require that `TEXT` parameters have to be transmitted by its four constituents, which will complicate the compiler, but probably still be advantageous to efficiency, since there will be fewer memory allocations involved.

The choice on which scheme to employ has not been determined, but neither option will inflict on the general compiler and run-time organization.

4.10 Classes

Classes have not been implemented in GSC. It appears however natural that the front-end uses compound data types based on `RECORD_TYPE` to lay out the structure of class objects. This is indicated in section 2.5.1 on page 13.

4.10.1 Prototype frames and the class hierarchy

As described by Dahl [6], prototype frames are determined at compile-time and loaded into a static part of the memory at run-time. The information contained in the prototype frames, is used by the run-time system. Each prototype object describes properties relating to a class. Most importantly, it contains the pointer that identifies the super-class prototype object. This information is needed for operators like `IS`, `IN` and `QUA` as well as run-time checks that verifies that reference assignments are legal in the sense that a reference only is allowed to reference an object of the class, or a sub-class thereof, which qualified the reference declaration.

4.10.2 Inheritance

Inheritance is solved by ensuring that the `RECORD_TYPE` data type corresponding to a sub-class, stores its inherited attributes at the same relative position as its super-class. This way, the position of an attribute can be determined at compile-time, although run-time checks to ensure that an object belongs to the correct class, may have to be generated (at least in relation with `QUA`).

4.10.3 Virtual procedures

As with other object-oriented languages, it is an indisputable fact that calls to virtual procedures requires extra run-time overhead. This is because the actual virtual procedure to call depends on the class membership of the object whose procedure is called. In certain situations, this can be determined at compile-time, but the general case requires the caller to consult a *virtual method table* that is stored along (or within) the class' prototype frame. The virtual method table associates the exact procedure entry address to a compile-time-determined integer value which uniquely identifies the set of commensurable virtual procedures in an inheritance hierarchy.

4.10.4 Object initialization

Conceptually, object initialization (through a `NEW`-call) has many similarities to activating a procedure. Like a procedure, an object may take parameters and its execution is often similar to the execution of a procedure. It may, however, detach itself from its dynamic context and may therefore continue execution after the block which originally called for its creation (`NEW`-call), is gone. Hence, the technique used in conjunction with procedure calls, where the parameter frame is preallocated in the caller's frame, cannot be employed for class object activations. This is, however, not very unfortunate because the creation of an object will lead to an allocation anyway. The class parameters are part of the attributes of the class object. Hence, if the allocation of the object is performed prior to evaluation of the parameters, the parameter values can be placed directly in the objects. During the evaluation of the parameters and the object execution, the object will be pointed to by a temporary reference situated in the caller's frame. If the `NEW`-expression is assigned to a reference variable, this temporary reference will be copied into this variable when the `NEW`-call returns. Otherwise, the temporary reference will die and the object become inaccessible and effectively die, as well.

4.11 Accessing non-local variables

Simula enables access to entities (variables, procedures and classes) located in outer blocks, i.e. statically enclosing blocks. At any point in a program's execution there exists at least one block instance for each of the outer block levels, relative to the active block. Hence, whenever a statement refers to an identifier of an entity located in an outer block, there will always be an appropriate frame in which to find the entity. The compiler can determine at compile-time which textual block contains the entity, so it knows the relative block-level and the structure of the block instance. The exact block instance, however, has to be determined at run-time by traversing the static chain.

At compile-time, the compiler will determine the difference in block-level between the active block, and the block containing the entity being accessed. This difference equals the number of links to traverse along the static chain in order to get to the appropriate block instance.

Dahl [6] discusses an optimization technique which involves an array structure maintained at run-time, where the *n*th element is a pointer to the block instance on block level *n* (counting from the outer-most block). Such a structure is referred to as a *context vector* or a *display* and provides quicker access to the correct block instance when the block level difference is larger than one. However, it requires more run-time maintenance, especially when program control moves inwards to an interior (i.e. higher) block-level.

The current implementation of GSC does not use a context vector, but merely produces code to dereference the static link of every block instance's activation frame on the path to the destined activation frame. The fact that the static link is the very first element of all frames, frees the compiler and run-time system from having to contemplate the entire structure of the frames that lie along the static chain.

4.12 FOR statements

A Simula FOR statement specifies a variable (the *controlled variable*) which is assigned values specified by a set of one or more patterns (FOR-list elements). The FOR statement causes its body (the *controlled statement*) to repeat zero or more times.

If there are more than one FOR-list element, the controlled statement is repeated until all elements the elements has been exhausted. Figure 4.5 shows a program which demonstrates a FOR statement including a list containing each of the three types of FOR-list elements supported by Simula. Execution of this program will make it iterate through (and output) the following values of I: 1, 2, 3, 3, 2, 1, 10.

Listing 4.5: A program utilizing three FOR-list elements

```
BEGIN
  INTEGER I;

  FOR I := 1 STEP 1 UNTIL 3, I - 1 WHILE I > 0, 10 DO
     BEGIN
       OutInt (I, 0);
       OutImage;
     END;
END;
```

FOR statements that contain more than one FOR-list element require some extra attention because each element has its own initialization operation, loop test and incrementation step. This is solved by allocating a control variable which is updated each time control proceeds to a new FOR-list element. Alternatively, the code of the controlled statement could be generated once for each FOR-list element. This would save the space and time involved in treating a FOR-list control variable, but the executable code would become more spacious. Yet, another approach could be to enclose the controlled statement in a fictitious (anonymous) procedure. This would reduce code size, and the control variable depicted above, would be represented implicitly by the changing dynamic link of the fictitious procedure.

As already indicated, my final choice was to generate a dedicated variable indicating the active FOR-list element, and include the necessary jump operations that makes execution return to the appropriate FOR-list element's code after having executed the controlled statement. Section 5.9 will give a more detailed presentation of this scheme.

4.13 Garbage collection

This report will not go into great details on garbage collection and what schemes and algorithms to prefer. The garbage collector is a part of the run-time system and, so, is fairly independent of the compiler. Some basic compliance is necessary, however. First of all, the run-time system's memory manager (including the garbage collector) and the compiler need to agree on the basic properties of references (pointers), like the bit-width.

Furthermore, if the chosen garbage collection scheme is like case 1 or 2, as described in section 3.5, the compiler also has to provide reference information which somehow will let the garbage collector know of the references in order to decide which objects are reachable or not. Obviously, the compiler and the garbage collector have to agree on the nature of this scheme.

It can also be noted that if the GC being used is non-moving (as required by cases 2 and 3 in section 3.5), the compiler may be able to perform better optimizations because the process of dereferencing references may be merged. An example is when adjacent statements repeatedly references variables found on a certain outer block level. If frames are allowed to move during garbage collection, the static link may have to be traversed every time.

To relieve the garbage collector of some of its pressure, GSC will try to include explicit deallocation of objects in certain cases where the compiler can infer the end of an object's lifespan. An evident such case is that the (procedure) activation frames may be deallocated when a procedure terminates. However, activation frames should still be subject to garbage collection because some procedure activations may never end properly since they may get detached (suspended) and their references get thrown away.

4.14 Conformity between compiler and run-time system

The issue of compliance between the compiler and the garbage collector raises the more general issue of the compiler's compliance with the run-time system in general. This issue is partly rooted in the differences between the language used to implement run-time system and the GSC compiler. The run-time system is typically implemented in C for convenience (like access to existing implementations like the one used by Cim) and efficiency. In that case, the run-time system's elementary data types will be based on those of C. Depending on the platform and the C implementation in question, these types may not be in compliance with their counterparts used by GSC.

This issue has not been dedicated very much attention. It will probably be reasonably easy to make a C-implemented run-time system play together with GSC-compiled Simula programs, especially if it is complied with GCC, which effectively will make both the run-time system and Simula programs be generated by the GCC back-end.

Cim does not face this same issue because the Simula programs are translated into C and therefore can be guaranteed to match the run-time system which is also implemented in C. To accomplish something similar in the case of GSC, the run-time system would have to be implemented in a language which can be compiled by GSC. This could be a language which is a superset of Simula and which provides the additional semantics needed to implement the Simula run-time system. Such semantics include general memory access through a general (C-like) pointers as well as mechanisms to access the hidden attributes and structures of the run-time organization, like static and dynamic chains, return addresses, prototype frames, virtual method tables and so on.

The meta-language used by Ole-Johan Dahl in [6], to express the semantics of certain run-time system routines, could be used as inspiration for such an extension to Simula.

Chapter 5

Implementation

The current implementation of the Simula front-end is not complete. However, I have tried to structure the code carefully and believe that future inclusion of the remaining functionality is a feasible task. This chapter provides an overview over the functionality that has been currently implemented. It is intended to give a brief impression of the implementation effort that I have put into GSC.

5.1 Overview

In order to build GSC, GCC has to be configured with the Simula front-end activated. The earliest version of GSC was based on version 2.95 of GCC. When GCC version 3.0 was released on June 18, 2001, I incorporated the necessary adaptations in order to get GSC to work with this new major version. The current version of GSC also works with the two subsequent revisions that have been released at the time of this writing—versions 3.0.1 and 3.0.2.

The GSC implementation consists of a directory of files. This directory is located as a sub-directory of the GCC back-end. Table 5.1 on page 55 lists the files comprising GSC. The most significant files, `decl.c` and `expr.c`, will be treated shortly. The remaining files are presented briefly in section 5.10 on page 53. The GSC source code can be acquired from:
`http://www.stud.ifi.uio.no/~knutroy/gsc/`.

The following list provides brief descriptions of the two most important files, which contain the initialization mechanism, the declaration handling, and the semantic analyzer of GSC.

`decl.c` contains the code that builds declaration nodes. This file also builds the data type trees for the basic Simula data types. It is described further in the following section.

expr.c contains the semantic analyzer and is described further in section 5.5.

5.2 The lexical analyzer

The lexical analyzer has been hand-written, contrary to the alternative option of using a lexical analyzer generator. Its main function, `yylex`, returns a *syntactic token* each time it is invoked by the parser.

The lexical analyzer incorporates a perfect hash function which is able to uniquely isolate at most one keyword candidate by looking at three characters in the token. This enables the lexical analyzer to quickly determine whether a token is a potential keyword and also which of them it might be. This method was inspired by a similar technique used in the C front-end of GCC version 2.95.

5.3 The syntax analyzer

The syntactic analysis is performed by a *parser* which is based on a specification of Simula's grammar. I early decided that I wanted to design my own grammar, instead of using the grammar included in Cim.

The motivation that led to this decision, was twofold: Firstly, I wanted to acquire experience in constructing a YACC-grammar myself, including getting intimate insight into the particular grammar with which I had to work. Secondly, Cim's grammar contained some conflicts and accepted some anomalous syntactic patterns, which I wanted to avoid. For instance, Cim's grammar accepts an arbitrary number of unary negation signs to precede an arithmetic expressions. In Cim, this anomaly is recognized and rejected during semantic analysis, but I wanted this situation to be a violation of the grammar itself, so that it could be caught during syntactic analysis.

The construction of a conflict-free grammar constituted a considerable effort. The final grammar is supposed to include the complete syntax of Simula, although large parts of it has not been tested and audited. The grammar contains no conflicts and consists of 216 *productions*. The file `parse.y` contains the grammar specification. The derived LALR(1)-parser generated by GNU Bison [11], comprises 457 states.

5.4 The intermediate representation

In Simula, the scope of a declaration includes the entire block in which it is located, including the part that textually precedes it. This means that all declarations of a block have to be spotted and recorded before the semantic analysis of the block's statements (including the statements of procedures and classes declared inside the block) can begin.

Traditionally, this has been accomplished by conceptually performing two passes over the source program, where the first pass performs syntax analysis, locates declarations, and produces an intermediate representation of the source program, while the second pass reads this intermediate representation, performs semantic analysis with respect to the declaration information gathered in pass one, and finally generates code.

The intermediate representation of the source program, between the first (syntactic) and the second (semantic) pass, has been implemented differently throughout the history of Simula implementations. Early implementations typically used a textual intermediate representation that was written to secondary storage in pass one, and then read back into primary storage (memory) in pass two. This had a positive influence on memory utilization which was a leading concern earlier.

Cim uses an intermediate representation which is referred to as the *M language* [8, 9]. This representation consists of a stream of tokens, suitable for output to a file although Cim keeps it in memory. The tokens of this representation are collected more or less directly from the parser, although their succession may be rearranged. This is especially true with tokens that constitute arithmetic expressions in which case they are ordered in a post-fix fashion so as to facilitate seamless and unambiguous construction of the syntax tree in the second pass. The parser is responsible for determining the correct precedence of the infix operators of Simula and producing the correct post-fix arrangements.

Contrary to Cim, GSC does not utilize a "flat" intermediate representation like the one just described. Instead, the building of the syntax tree proper is initiated by syntactic analyzer. The syntax tree is then revised by the semantic analyzer. The decision to abolish an intermediate stream-like representation of the source program, was based on several observations:

- The M representation would increase the complexity of the front-end, with respect both to overall structure and execution speed.
- The historical objective to pursue memory parsimony, has diminished.
- The Java front-end to GCC, which has to perform two passes over the source as well, also builds the skeleton of the syntax tree during syntactic analysis.

5.5 Semantic analysis

The semantic analyzer is located in the file `expr.c`. The following list includes brief descriptions of the most important functions that comprise the semantic analyzer.

expand_program is the root function of the semantic analyzer. It is activated by the syntax analyzer at the end of parsing. It receives the entire syntax tree through a pointer to the ultimate root node which represents the outer-most block of the Simula program.

expand_procedure is called for each procedure declaration and controls the compilation the procedure body after first compiling all internal declarations through a call to `expand_decls`.

expand_class is similarly called for each class declaration. First internal declarations are compiled and then the class body itself is expanded.

expand_decls identifies all local declarations and calls `expand_procedure` and `expand_class` for all local procedure and class declarations, respectively.

expand_stmt compiles a statement, including calling itself recursively if necessary when the statement is compound, i.e. if it contains substatements (which is true for blocks and `IF`, `WHILE`, `FOR`, and `INSPECT` statements).

complete_stmt recurses through all the statements of a block (including sub-blocks), similarly to `expand_stmt`. `complete_stmt` is called prior to `expand_stmt` and is used to perform the semantic analysis and also calculates the amount of frame space which is required by the statement.

complete_expr is called from `complete_stmt` for all simple statements. It provides `complete_stmt` with information about the space requirement of the simple statement, which may be nonzero if the simple statement needs temporaries or a parameter frame due to procedure or object activations.

Figure 5.1 shows an activation tree which indicates the pattern by which the functions of the semantic analyzer activate each other.

5.6 Frame layout and preallocation

Among the things which are done in the initial stage of the semantic analysis, is the structuring of activation frame layout. Due to preallocation, this calls for a recursive analysis of the body of each procedure and class, including the body of the outer-most program block. The analysis is done

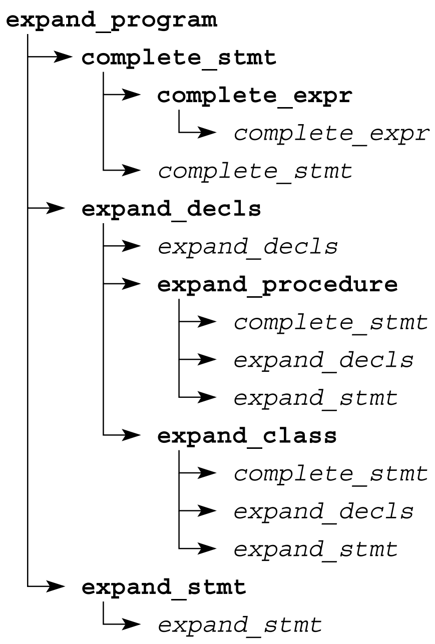

Figure 5.1: Activation tree of the semantic analyzer

in `complete_stmt` which calls itself recursively in order to analyze all statements and sub-statements that are supposed to have their local variables and temporaries preallocated in the surrounding frame.

The culminating frame layout is described by a compound data structure definition analogous to a C data structure which is a `struct` containing `structs` and `unions` inside `structs` and `unions` and so on.

Simple statements may need temporaries if required by the strict evaluation order of Simula. This is true if a statement contains activations of procedures. In that case space is needed to store local temporaries while an activated procedure is executed. Also, a procedure activation will need space for the preallocated parameter frame, as explained in section 4.2.2. The space requirement of simple statements is computed by the recursive function `complete_expr` which performs type-checking at the same time.

Block statements need a frame comprising all local variables. These are organized like fields of a C `struct`. `complete_stmt` is then called for each internal statement to compute their local frame layout. The returned frame layouts are combined into a `union` (in the C sense) which means that each statement will utilize the same area of space. Hence, the total size of the union equals the space required by the most spacious statement—remembering that a statement can be anything from a simple assignment statement, to a comprehensive sub-block statement, possibly containing its own declarations and sub-blocks and so on.

IF statements consist of two or three parts—the Boolean condition expression, the THEN clause which is a statement, and alternatively an ELSE clause statement. These will all have disjoint lifetimes during the execution of an IF statement, and consequently their local frames may be organized in a union-like pattern.

WHILE statements are similar to IF statements, but consist of only the Boolean condition expression and the body statement. These also have disjoint lifetimes and, hence, share the same area.

FOR statements have two alternative frame layouts depending on the number of FOR-list elements. If there are more than one, the frame must include space for a control variable. This variable is live for the entire execution of the FOR statements and, therefore, cannot share space with other parts of the FOR statements. Evaluation of the FOR-list element expressions and the execution of the controlled statement, are all disjoint and may use the same memory area.

5.7 Evaluation order

Evaluation order is important if evaluation of one operand may influence the evaluation of the others, so the back-end has to be made aware of the strict evaluation order of Simula.

The RTL expansion mechanism of the back-end usually generates the RTL code to perform the evaluation operation of an expression tree. However, for tree nodes that represent simple variables, the expansion routine will return an *lvalue* representation instead of an *rvalue*, which means that the value of the variable may not be accessed at the desired moment during run-time. This is a problem if the simple variable is the left-hand node of an operation and the evaluation of the corresponding right-hand contains side-effects.

The Java front-end faces a similar challenge due to similar evaluation order requirements imposed by the Java specification. The Java front-end uses a SAVE_EXPR to force immediate evaluation of lvalues. A SAVE_EXPR makes the sub-tree beneath it be evaluated once and makes the result be put in a temporary variable. If another part of the syntax tree references the SAVE_EXPR node, the back-end will realize that this expression has already been evaluated, and so, the temporary value will be returned.

So, according to the Java front-end it is sufficient to encapsulate the left-hand sub-tree within a SAVE_EXPR node, i.e. insert a SAVE_EXPR node as the

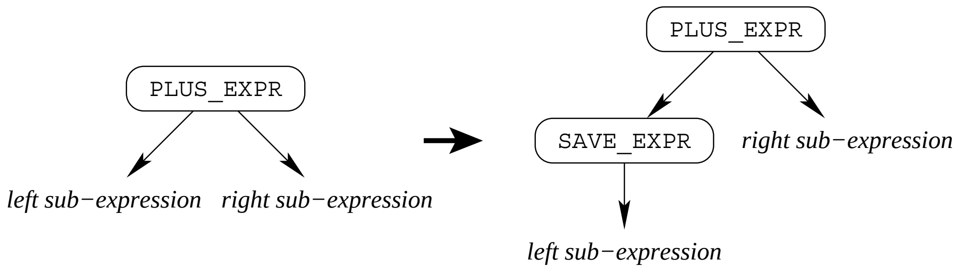

Figure 5.2: Technique used by the Java front-end to ensure correct evaluation order

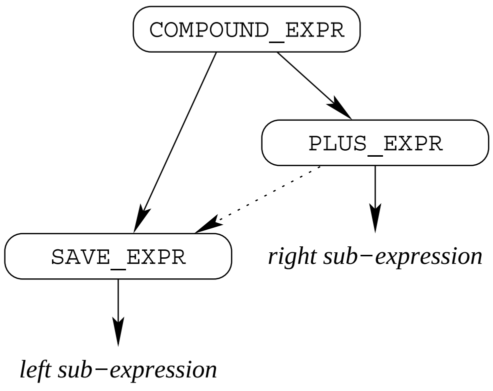

Figure 5.3: Technique used by GSC to ensure correct evaluation order

left-hand node, and let the original left-hand node (including any sub-tree beneath it) become the only child of the `SAVE_EXPR` node. This transformation is shown in figure 5.2.

GSC elaborates this scheme in a way that makes the desired evaluation order more clearly. I employed this scheme because it seemed presumptive to expect that the back-end's RTL expansion mechanism would expand the sub-trees in left-to-right order strictly and unconditionally. This is shown in figure 5.3. This technique is conceptually similar to actually splitting an expression into several sequenced statements. The use of `COMPOUND_EXPR` should guarantee correct order of evaluation.

5.8 Explicit storing of temporaries

As discussed in the previous section, the constrained evaluation order in Simula, demands that program execution makes copies of variables that may be changed as side-effects of procedure or object activations. These

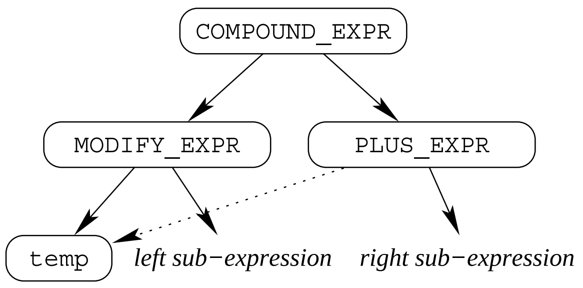

Figure 5.4: Explicit storing of a temporary

copies are stored in temporary variables and are the kind of temporaries that are live across procedure and object activations. Hence, they need to be stored in heap-allocated frames. The technique chosen by GSC, is to preallocate such areas within the activation frame of the local block as described in section 4.2.2.

The actual copying operation has to be incorporated into the syntax tree by the front-end because the back-end's notion of temporaries is insufficient since this type of temporary has to be stored in a designated heap-based location.

The front-end elicits this copying by inserting appropriate MODIFY_EXPR nodes into the syntax tree to represent assignment operations. The left-hand child specifies the destination, while the right-hand sub-tree represents the value to be assigned.

Extending the example of the previous section, the modified syntax tree looks like the one in figure 5.4.

In this situation the SAVE_EXPR has been made superfluous, because the temporary value is saved explicitly in the the MODIFY_EXPR. In fact, if the SAVE_EXPR had not been removed, it could have provoked an erroneous stack-based temporary being generated by the back-end.

When the right sub-expression does not have potential side-effects, there is no need to make a heap-based temporary copy of the left side. Furthermore, when the left sub-expression is (or reduces to) a constant value, there is also no need for a temporary to store this value because a constant is immune to side-effects. The current implementation is aware of these optimization criteria, and attempts to take appropriate action, although this mechanism has not been extensively audited.

It can also be noted that evaluation order is insignificant when a binary operation (left and right side) has no side-effects. This can potentially be recognized and utilized by the front-end which in that case may refrain from generating the extra COMPOUND_EXPR node described in this and the previous section.

When only the left sub-expression has side-effects, no heap-based temporary is needed, but evaluation order is again significant because the evaluation of the left side can have side effects that influence the right-hand side.

5.9 FOR statements

The current GSC implementation contains support for code generation of FOR statements. A FOR statement containing more than one FOR-list element, needs a control variable to signify the start address of the next iteration. When there is only one element, the start address can be determined and incorporated into the generated code at compile-time.

The following is an algorithmic presentation of how the current front-end implementation arranges the semantics of FOR statement.

1 If the FOR statement has more than one FOR-list element, declare (and allocate space for) a control variable which will contain the start address of the FOR-list element to be activated on the next iteration.

2 For each FOR-list element, perform one of the following three steps based on the corresponding element type. It is important that the elements are handled in order of appearance so that the chronological ordering gets correct at run-time.

2 a `...`*`A1`*` STEP `*`A2`*` UNTIL `*`A3`*`, ...`

- If the FOR statement has more than one FOR-list element, expand an assignment operation where the FOR-list control variable is assigned to the address of the label that will be generated tree steps down.
- Generate the initialization operation, which means assigning *A1* to the controlled variable.
- Generate a jump operation that will make execution skip the incrementation step the first time.
- Generate the label, marking the program point where control should return on subsequent iterations. This is the address which is assigned to the FOR-list control variable three steps above. If there is only one FOR-list element, this is the target address for the unconditional jump generated after the controlled statement in step 6 below.
- Generate the incrementation step, meaning increasing the controlled variable by *A2*.
- Generate a label, marking the point where the control continues on the first iteration, when the incrementation step is skipped.

- Generate the loop test which checks whether a new iteration of this FOR-list element should be done, i.e. whether `A2 * (C - A3) <= 0` where *C* is the controlled variable.
- Generate a conditional jump to the controlled statement if the loop test is true.

2b `...V WHILE B, ...`

- If the FOR statement has more than one FOR-list element, expand an assignment operation where the FOR-list element control variable is assigned to the address of the label that will be generated in the next step.
- Generate the label, marking the program point where control should return on subsequent iterations.
- Generate the initialization operation, which means assigning *V* to the controlled variable.
- Generate the loop test, checking whether a new iteration of this FOR-list element should be done, i.e. whether *B* is true.
- Generate a conditional jump to the controlled statement if the loop test is true.

2c `...V, ...`

- If the FOR statement has more than one FOR-list element, expand an assignment operation where the FOR-list element control variable is assigned to the address of the label that will be generated three steps down.
- Generate the initialization operation, which means assigning *V* to the controlled variable.
- Generate an unconditional jump to the controlled statement.
- Generate the label, marking the program point where control should return on subsequent iterations.

3 Generate an unconditional jump to the exit label which effectively ends the FOR statement.

4 Generate the label which marks the start of the controlled statement.

5 Generate the code for the controlled statement (the body) of the FOR statement.

6 If FOR statement has more than one FOR-list element, expand an unconditional jump to the address contained in the control variable. Otherwise generate an unconditional jump to the label generated in step 2.

7. Generate the exit label which marks the end of the FOR statement.

The Simula specification [5] states that a FOR-list element of type ...*A1* STEP *A2* UNTIL *A3*, ... should repeat as long as *A2* * (*C* - *A3*) <= 0 is true (where *C* is the controlled variable). However, it is unfortunate to generate this expression exactly as stated in the specification. This is because the back-end is not able to realize that its truth-value can be determined without performing the time-consuming subtraction and multiplication. GSC, therefore, generates a revised version which enables the back-end to optimize better. This version eliminates the subtraction and multiplication, but introduces one extra comparison operation instead. An additional benefit is that when *A2* is represented by a compile-time determinable constant (which is usually the case), the expression will be simplified further by the back-end. If written out in Simula syntax, the revised Boolean expression looks like in listing 5.1.

Listing 5.1: The revised loop test

```
IF A2 > 0
THEN
  C <= A3
ELSE
  IF A2 < 0
  THEN
    C >= A3
  ELSE
    TRUE
```

It is however worth noting that if *A2* equals zero at compile-time so that the expression becomes unconditionally true, the evaluation of *A3* will not be performed under this scheme. This will lead to erroneous behavior if the evaluation of *A3* has side-effects. Consequently, a final implementation should ensure that *A3* is evaluated if appropriate even though the truth-value can be derived from *A2* alone.

Figure 5.5 shows the loop condition represented as a syntax tree, approximately like the one transmitted to the back-end.

5.10 The remaining files

The files decl.c and expr.c contains the most interesting pieces of the GSC implementation, and they have been presented earlier in this chapter. Here follows a brief description of the remaining files belonging to GSC. Table 5.1 lists all the primary files (i.e. those which are not automatically generated), including their size counted by number of lines and bytes at the time this report was completed.

config-lang.in contains a few definitions which are required by the configuration process of GCC, including the creation of GCC's Makefile.

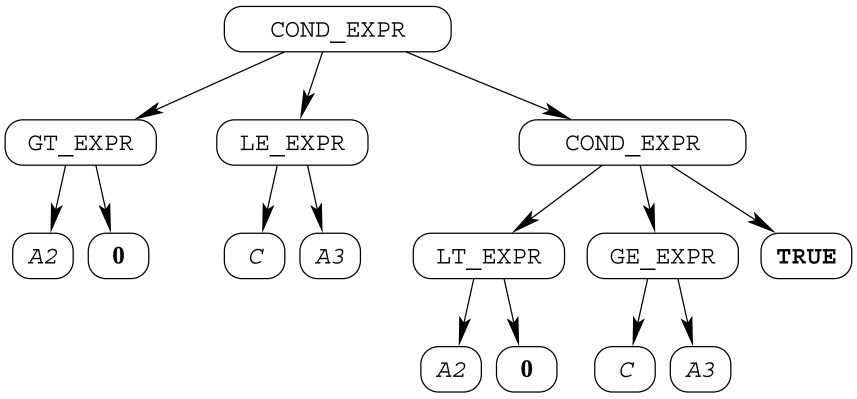

Figure 5.5: Syntax tree of the revised loop test

`lang.c` contains a few Simula-specific extensions to mechanisms that are performed by the back-end.

`lex.c` contains the lexical analyzer.

`Make-lang.in` contains the template for the part of the GCC `Makefile` that is specific to the Simula front-end, which includes identifying file dependencies and the commands used to automatically generate files like `parse.c`, `parse.h` and `resword.h`.

`parse.y` contains the YACC-compatible specification of the Simula grammar, including semantic actions to build the skeleton of the syntax tree. This file is read by *GNU Bison* which produces the syntax analyzer represented by `parse.c` and `parse.h`.

`resword.gperf` contains the 64 reserved keywords of Simula, including the symbolic grammatical terminals used in the grammar specification to represent them. This file is read by a perfect hash function generator, *GNU gperf* [12], to generate a hash function which gets incorporated into the lexical analyzer (`lex.c`) and facilitates efficient recognition of keywords.

`simula-tree.def` contains the definition of the extended set of tree codes which are used in the syntax tree between syntactic and semantic analysis.

`simula-tree.h` contains function profile declarations and extended tree node access macros.

Filename	*Lines*	*Bytes*
gcc/simula/**config-lang.in**	35	1 295
gcc/simula/**decl.c**	535	14 802
gcc/simula/**expr.c**	1 840	52 313
gcc/simula/**lang.c**	319	8 193
gcc/simula/**lex.c**	1 135	25 477
gcc/simula/**Make-lang.in**	138	4 217
gcc/simula/**parse.y**	1 171	22 789
gcc/simula/**resword.gperf**	66	1 016
gcc/simula/**simula-tree.def**	78	2 954
gcc/simula/**simula-tree.h**	170	6 178
Total	5 487	139 234

Table 5.1: The files comprising GSC

Chapter 6

Results

This chapter presents a few simple programs that have been compiled using GSC.

6.1 Run-time library

As noted earlier in this report, GSC does not have a real run-time system. However, to facilitate simple output, rudimentary versions of the two procedures `OutInt` and `OutImage` which are connected to the *standard output stream* (“stdout”), have been implemented. Listing 6.1 shows the C source code of these two functions.

Listing 6.1: The preliminary run-time system

```
#include <stdio.h>

void
OUTINT (int i, int w)
{
  printf ("%d", i);
}

void
OUTIMAGE (void)
{
  putchar ('\n');
}
```

To provide convenient linking between Simula and this C-based run-time library, GSC simply presume that all undeclared identifiers refer to external functions and, hence, generates traditional stack-oriented function calls. To indicate that this is not standard Simula behavior, it issues a warning when doing so. There is no mechanisms ensuring that external functions are activated correctly with respect to the expected number and types of parameters. Hence, the programmer is responsible for providing two `INTEGER` parameters to `OutInt`. Then, the rest of the pieces fall into place automatically since GSC's INTEGER type is equivalent to C's int, and the stack-oriented call sequence generated by GSC conforms to that of the C-based run-time library generated by the GCC C compiler.

As a provisional memory manager, GSC uses the standard C functions malloc and free to allocate and deallocate heap memory. The current version of GSC is only able to handle allocation of frames, while allocation of class objects has not been implemented. The lack of a garbage collector does not affect the following sample programs since frames are deallocated explicitly when control leaves their containing block. Hence, these programs do not generate garbage at all.

6.2 Sample programs

The programs that are used to test GSC, are simple. The actual Simula compiler, sc1, outputs an assembler file based on the Simula source file which is specified as a command-line argument. The assembler file and the run-time system are then linked together to form an executable file. The GCC compiler driver, gcc, is used for this purpose. The compiler driver automatically invokes the C compiler (which compiles the run-time system), the assembler (which assembles the Simula program), and the linker (which generates the executable file). The optimization level of GSC is specified using the -O command-line argument. The assembler listings shown in this chapter have been generated on optimization level 2 (-O2) unless otherwise stated.

6.2.1 Program 1

The first program simply uses OutInt to output the number 10. The program has been implemented in three ways. The first version provides the number directly. The other two use a constant, I, in the evaluation of the parameter value 10. The tree versions are shown in listings 6.2, 6.3, and 6.4, respectively.

The second and third version demonstrate that GSC does not allocate space for constants. Since the values of constants are determinable at compile-time, GSC rather applies their values at compile-time. Compound arithmetic expressions that only contain constant constituents, are *constant folded* by the back-end on the front-end's request.

Listing 6.2: Program 1 a

```
BEGIN
  OutInt (10, 0);
  OutImage;
END;
```

Listing 6.3: Program 1 b

```
BEGIN
  INTEGER I = 10;

  OutInt (I, 0);
  OutImage;
END;
```

Listing 6.4: Program 1 c

```
BEGIN
  INTEGER I = (1 + 2 + 2) * 2 // 3;

  OutInt (I * I + I - I // 2 * 2, 0);
  OutImage;
END;
```

Listing 6.5 shows the assembler code produced by GSC. The first three and the last four lines constitute the standard prologue and epilogue, respectively. As have been discussed earlier, the standard prologue and epilogue should ideally not be generated because they conflict with the heap-oriented call / return sequence employed by the front-end.

Listing 6.5: Assembler code generated by GSC for the three versions of program 1

```
main:
       pushl  %ebp
       movl   %esp, %ebp
       subl   $16, %esp
       pushl  $0
       pushl  $10
       call   OUTINT
       call   OUTIMAGE
       movl   %ebp, %esp
       xorl   %eax, %eax
       popl   %ebp
       ret
```

6.2.2 Program 2

Program 2, shown in listing 6.6, introduces a local variable, `I`, and a `FOR` statement where `I` is used as the controlled variable.

Listing 6.7 shows the non-optimized assembler code produced by GSC (i.e. when the optimization level is zero), while listing 6.8 includes the corresponding optimized code (optimization level 2). The effect of the optimization is clearly evident. In the non-optimized version, some instructions are repeated and some operations are split into several instructions in cases where a single instruction would have sufficed. The first obvious example is that `%esp` (the stack pointer register) is subtracted in two steps, on lines 3 and 4. Another example is that the `%eax` register is initialized to the

Listing 6.6: Program 2

```
BEGIN
  INTEGER I;

  FOR I := 0 STEP 2 UNTIL 10 DO
     BEGIN
       OutInt (I, 0);
       OutImage;
     END;
END;
```

variable __lb and then the memory position to which it refers, is set to zero, twice. This is a behavior caused by the semantics given to the back-end by the front-end, as explained below.

There are even more examples of things that have been optimized in listing 6.8, and they all provide reassuring evidence that the back-end does in fact perform code optimization.

Listing 6.7: Non-optimized assembler code of program 2

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $4
        call    malloc
        addl    $16, %esp
        movl    %eax, __lb
        movl    __lb, %eax
        movl    $0, (%eax)
        movl    __lb, %eax
        movl    $0, (%eax)
        jmp     .L2
.L3:
        movl    __lb, %eax
        movl    (%eax), %edx
        addl    $2, %edx
        movl    __lb, %eax
        movl    %edx, (%eax)
.L2:
        movl    __lb, %eax
        cmpl    $10, (%eax)
        jg      .L6
        subl    $8, %esp
        pushl   $0
        movl    __lb, %eax
        pushl   (%eax)
        call    OUTINT
        addl    $16, %esp
        call    OUTIMAGE
        jmp     .L3
.L6:
        subl    $12, %esp
        pushl   __lb
        call    free
        addl    $16, %esp
        movl    $0, __lb
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Program 2 uses a variable of type INTEGER which is 32 bits, i.e. 4 bytes, wide. This variable is allocated in a heap-based frame. It is the only element needed, since the outermost block has no need for static and dynamic links. Hence, the total size of the frame equals 4 bytes. The frame is allocated on lines 5 and 6 in listing 6.8. The return value from malloc (held in the %eax register) is a pointer to the allocated area and is assigned to the global run-time system variable, __lb, which continuously points to the active frame at any given time during the program execution. I is initialized to zero on line 9. This is required because variables should always be initialized in Simula.

The FOR statement is then ready to execute. The variable I needs to

Listing 6.8: Optimized assembler code of program 2

```
1  main:
2          pushl  %ebp
3          movl   %esp, %ebp
4          subl   $20, %esp
5          pushl  $4
6          call   malloc
7          addl   $16, %esp
8          movl   %eax, __lb
9          movl   $0, (%eax)
10 .L2:
11         movl   __lb, %eax
12         cmpl   $10, (%eax)
13         jg     .L6
14         pushl  %edx
15         pushl  %edx
16         pushl  $0
17         movl   (%eax), %eax
18         pushl  %eax
19         call   OUTINT
20         call   OUTIMAGE
21         addl   $16, %esp
22         movl   __lb, %eax
23         movl   (%eax), %ecx
24         addl   $2, %ecx
25         movl   %ecx, (%eax)
26         jmp    .L2
27 .L6:
28         subl   $12, %esp
29         pushl  %eax
30         call   free
31         movl   $0, __lb
32         movl   %ebp, %esp
33         xorl   %eax, %eax
34         popl   %ebp
35         ret
```

be set to the initial value of the FOR statement, which is zero. In the non-optimized version, this initialization is credulously generated by the back-end as instructed by the front-end. This explains why I is set to zero twice in the non-optimized version, as commented above. In the optimized version, however, the back-end omits the redundant assignment to I.

Returning to the non-optimized version, control leaps from line 14 to label .L2 on line 21, whereupon the loop test is evaluated on lines 22–24. When I is greater than 10, the FOR statement is finished and control leaves by a jump to label .L6 on line 33. However, as long as I is less than or equal to 10, program control proceeds to the controlled statement which is comprised by the lines 25–31. Control then returns to label .L3 where the incrementation operation increases I by two (lines 16–20), whereupon program control, once again, reaches evaluation of the loop test on lines 22–24. Note that the incrementation operation was initially omitted by a jump from line 14 to line 21.

Again, the optimized version in listing 6.8 is clearly more compact. The GCC back-end has rationalized the .L3 label away by moving the incrementation operation to the end of the controlled statement. The controlled statement comprises lines 14–21 and the incrementation operation constitutes lines 22–25.

6.2.3 Program 3

Program 3 a is shown in listing 6.9. This program has been compiled using both Cim and GSC. Cim produces a C file which in turn is compiled by the GCC C compiler.

The Cim program contains code to initialize and set up the run-time system and environment. This initialization is not relevant in the comparison since GSC does not have a run-time system to initialize. Hence, it is only

Listing 6.9: Program 3 a

```
BEGIN
  INTEGER I;

  FOR I := 1 STEP 1 UNTIL 10 DO
     BEGIN
       INTEGER J, K;

       FOR J := 1 STEP 1 UNTIL 1000000 * I DO
          BEGIN
            K := K + 1;
          END;

       OutInt (K, 0);
       OutImage;
     END;
END;
```

interesting to compare the internal parts of the program produced by Cim. Listing 6.10 shows the relevant part of the assembler code produced by Cim, while listing 6.11 shows the corresponding code generated by GSC.

It is worth noting that even though the program generated by GSC does not perform any run-time system initialization, the semantics of the inspected area can be regarded as complete, so this comparison is pretty fair. The only potential change in a complete implementation would be that `malloc` and `free` are replaced by a proper garbage collecting memory manager.

Here is a summary of the assembler code produced by Cim and shown in listing 6.10:

- Lines 1–3 initialize the frame for the program block.
- Line 6 initializes `I` to 1.
- Lines 7–12 evaluate the loop test of the outer `FOR` statement. If it is true, control jumps to label `.L10`.
- Lines 13–15 constitute the end of the program block.
- Lines 19–20 allocate and enter the inner block, containing `J` and `K`.
- Lines 23–24 initialize `J` to 1.
- Lines 25–33 read `I` and multiply it by 1 000 000.
- Lines 34–41 read `J` and evaluate the loop test of the inner `FOR` statement. If it is true, control jumps to label `.L16`.
- Lines 42–55 execute `OutInt` and `OutImage`.
- Line 56 deallocates and leaves the inner block.

- Lines 58–61 increment `I` by one.
- Lines 62–68 evaluate the loop test of the outer `FOR` statement. If it is false, control jumps to label `.L11`. If it is true, control jumps to label `.L10`.
- Lines 69–77 contain the body and incrementation statement of the inner `FOR` statement, which involves increasing `K` and `J` by one.
- Lines 78–89 read `I` and multiply it by 1 000 000.
- Lines 90–92 read `J` and proceed to label `.L33` to evaluate the loop test of the inner `FOR` statement.

This is the summary of the GSC-generated code shown in listing 6.11:

- Lines 1–2 allocate the frame for the program block.
- Lines 4–5 initialize `I` to 1.
- Lines 6–9 evaluate the loop test of the outer `FOR` statement. If it is false, control jumps to label `.L6`.
- Lines 10–11 initialize `K` to zero and `J` to one.
- Lines 12–22 read `I` and multiply it by 1 000 000.
- Lines 23–24 evaluate the loop test of the inner `FOR` statement. If it is false, control jumps to label `.L11`.
- Lines 25–27 increase `K` by one.
- Lines 28–32 increase `J` by one and proceed to `.L7` to evaluate the loop test of the inner `FOR` statement.
- Lines 33–40 execute `OutInt` and `OutImage`.
- Lines 41–46 increase `I` by one and proceed to `.L2` to evaluate the loop test of outer `FOR` statement.
- Lines 47–50 deallocate the frame of the program block.

The only directly comparable code fragments that span several instructions, are the multiplications by one million on lines 26–33 and 81–88 in listing 6.10 and lines 15–22 in listing 6.11. Other than that, the two programs are quite different. The variable storage and access are not done the same way and the structure also differs moderately.

An important difference between the two programs is that the GSC version preallocates the inner sub-block's frame (containing `J` and `K`) within

Listing 6.10: Essential parts of code produced by Cim for program 3 a

```
        movl    $__blokk206, __sto
        movl    $__p206, (%esp)
        call    __rb
        popl    %ebx
        popl    %eax
        movl    $1, __blokk206+28
        pushl   $10
        pushl   $1
        call    __rsigndi
        addl    $16, %esp
        testl   %eax, %eax
        jle     .L10
.L11:
        call    __rbe
        jmp     .L6
.L10:
.L14:
        subl    $12, %esp
        pushl   $__p208
        call    __rb
        popl    %ecx
        popl    %ebx
        movl    __lb, %ebx
        movl    $1, 28(%ebx)
        movl    __blokk206+28, %ecx
        movl    %ecx, %edx
        sall    $5, %edx
        subl    %ecx, %edx
        movl    %edx, %eax
        sall    $6, %eax
        subl    %edx, %eax
        leal    (%ecx,%eax,8), %eax
        sall    $6, %eax
        pushl   %eax
        movl    28(%ebx), %edx
        pushl   %edx
.L33:
        call    __rsigndi
        addl    $16, %esp
        testl   %eax, %eax
        jle     .L16
.L17:
.L12:
        movl    __lb, %eax
        pushl   %ecx
        pushl   $0
        movl    32(%eax), %edx
        movl    __blokk0FILE+32, %eax
        pushl   %edx
        pushl   %eax
        call    __rooutint
        popl    %ebx
        movl    __blokk0FILE+32, %ecx
        pushl   %ecx
        call    __rpoutimage
        call    __rbe
        popl    %eax
        movl    __blokk206+28, %eax
        popl    %edx
        incl    %eax
        movl    %eax, __blokk206+28
        pushl   $10
        pushl   %eax
        call    __rsigndi
        addl    $16, %esp
        testl   %eax, %eax
        jg      .L11
        jmp     .L10
.L16:
.L18:
        movl    __lb, %ecx
        movl    32(%ecx), %eax
        movl    28(%ecx), %ebx
        incl    %eax
        incl    %ebx
        movl    %eax, 32(%ecx)
        movl    %ebx, 28(%ecx)
        movl    __blokk206+28, %ebx
        pushl   %edx
        pushl   %edx
        movl    %ebx, %edx
        sall    $5, %edx
        subl    %ebx, %edx
        movl    %edx, %eax
        sall    $6, %eax
        subl    %edx, %eax
        leal    (%ebx,%eax,8), %eax
        sall    $6, %eax
        pushl   %eax
        movl    28(%ecx), %eax
        pushl   %eax
        jmp     .L33
.L6:
```

the program block's frame. Consequently, this frame needs space to store three INTEGERs (I, J, and K), i.e. 12 bytes.

Cim uses an optimization technique where the frame of the program block is statically allocated at load-time. Since there is always only one instance of the program block, this can be done, and should probably also be included in a future version of GSC.

Cim does, however, not preallocate sub-blocks. Hence, the Cim program performs ten sub-block frame allocations, one for each iteration of the FOR statement. These 10 allocations do not constitute a measurable impact on

Listing 6.11: Code generated by GSC for program 3 a

```
        pushl  $12
        call   malloc
        addl   $16, %esp
        movl   %eax, __lb
        movl   $1, (%eax)
.L2:
        movl   __lb, %eax
        cmpl   $10, (%eax)
        jg     .L6
        movl   $0, 8(%eax)
        movl   $1, 4(%eax)
.L7:
        movl   __lb, %ebx
        movl   (%ebx), %ecx
        movl   %ecx, %edx
        sall   $5, %edx
        subl   %ecx, %edx
        movl   %edx, %eax
        sall   $6, %eax
        subl   %edx, %eax
        leal   (%ecx,%eax,8), %eax
        sall   $6, %eax
        cmpl   %eax, 4(%ebx)
        jg     .L11
        movl   8(%ebx), %eax
        incl   %eax
        movl   %eax, 8(%ebx)
        movl   __lb, %eax
        movl   4(%eax), %edx
        incl   %edx
        movl   %edx, 4(%eax)
        jmp    .L7
.L11:
        pushl  %eax
        pushl  %eax
        pushl  $0
        movl   8(%ebx), %ebx
        pushl  %ebx
        call   OUTINT
        call   OUTIMAGE
        movl   __lb, %eax
        addl   $16, %esp
        movl   (%eax), %ecx
        incl   %ecx
        movl   %ecx, (%eax)
        jmp    .L2
.L6:
        subl   $12, %esp
        pushl  %eax
        call   free
```

the execution time. However, if the innermost block is extended to include local variables, the effect becomes evident. Program 3 b, shown in listing 6.12, contains such an extension. Since the inner block now contains local variable, `__rb` and `__rbe` will be called for each iteration of the inner block as well. However, the GSC-generated program preallocates this block as well.

Table 6.1 presents the execution times observed by both program 3 a and program 3 b for Cim as well as GSC. The time measurements were done on a 200 MHz INTEL PENTIUM processor.

Listing 6.12: Program 3 b which includes an extra inner integer declaration

```
BEGIN
  INTEGER I;

  FOR I := 1 STEP 1 UNTIL 10 DO
     BEGIN
       INTEGER J, K;

       FOR J := 1 STEP 1 UNTIL 1000000 * I DO
          BEGIN
            INTEGER L;
            K := K + 1;
          END;

       OutInt (K, 0);
       OutImage;
     END;
END;
```

	Cim	GSC
Program 3 a	11 s	5.8 s
Program 3 b	53 s	6.1 s
Program 4	3.9 s	1.0 s

Table 6.1: Execution times on an INTEL PENTIUM processor

6.2.4 Program 4

Listing 6.13 shows program 4. It contains two procedures, `P` and `Q`. The main program activates `Q` once, and `Q` activates `P` one million times.

Listing 6.13: Program 4

```
BEGIN
  INTEGER PROCEDURE P (A, B); INTEGER A, B;
  BEGIN
    P := A - B;
  END;

  INTEGER PROCEDURE Q;
  BEGIN
    INTEGER I, O, S;

    FOR I := 0 STEP 1 UNTIL 1000000 DO
       BEGIN
         S := S + P (I, O);
         O := I;
       END;

    Q := S;
  END;

  OutInt (Q, 0);
  OutImage;
END;
```

Listing 6.14 shows the assembler code generated by GSC. Here is a summary of the most important things to notice in the generated code.

- Taking the program block (`main`) first, lines 88 and 89 allocates 16 bytes of frame. This space is used by the preallocated parameter frame which includes space for a static link, a dynamic link, an exit address, and the `INTEGER` return value—each 4 bytes wide.
- Lines 93–98 initialize and execute the activation of `Q`. Lines 93 and 94 initialize the static link and dynamic link, respectively. Line 95 sets `__lb` to the parameter block (which in this case coincides with the containing frame). Line 96 initializes the exit address and line 97 activates `Q`. Note that the exit address is set incorrectly, as will be commented in the next chapter.

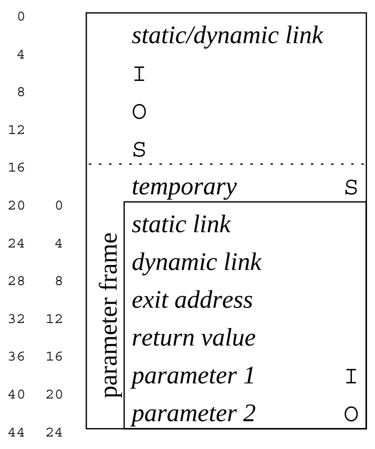

Figure 6.1: Frame of procedure `Q` in program 4

- After return from `Q`, lines 100–102 set `__lb` to what was stored in the dynamic link. Since the containing frame coincides with the parameter frame in this case, `__lb` remains unchanged in this case.
- Line 103 extracts the return value of `Q`, which is then printed on lines 104–106.
- In procedure `Q`, on lines 27 and 28, the return value is set to default value, zero. A local frame of 44 bytes is allocated on lines 29 and 30. The structure of this frame is shown in figure 6.1.
- The local block pointer, `__lb`, is updated, and the dynamic / static link of the new frame is set to the old value on lines 32–34. Lines 35–37 set the local variables to zero.
- Lines 38–42 contain the loop test.
- Lines 43–59 contain the activation of `P`. Line 45 sets the first parameter (`I`). Lines 44 and 46 store a temporary copy of `S`. Line 47 sets the dynamic link. Lines 48–49 set the second parameter (`O`). Lines 50–51 set the static link. Lines 52–53 set `__lb` to the parameter block. Line 54 sets the return address (note that since this is done relative to the new `__lb` value, the offset is different—otherwise it would be on byte offset 28). Lines 56–58 set `__lb` back and line 59 extracts the return value.
- Lines 60–62 read the temporary copy of `S`, which was stored on line 46, and then add it to the result value of `P` whereupon the sum is stored in `S`.
- Lines 63–69 assign `I` to `O` and increase `I` by one before jumping to the loop test.

	Lund	Cim	GSC
Program 3 a	27 s	13 s	2.7 s
Program 3 b	150 s	46 s	2.7 s
Program 4	2.4 s	3.3 s	0.85 s

Table 6.2: Execution times on a SUN SPARC processor

- Line 71 loads the static/dynamic link to `%eax`, line 72 reads `S` and line 74 stores it in the result value position of the dynamically enclosed frame (pointed to by the static/dynamic link loaded on line 71).

When compiled with Cim, program 4 finishes in 3.9 seconds. The Cim version also makes a concluding remark that 61 garbage collections were performed in 1.3 seconds. The GSC version completes in 1.0 seconds.

It is worth noting that the program produced by GSC does not utilize the optimization discussed in section 4.2.2 on page 30. Even though the local frame of procedure `P` is empty and therefore never needs to be allocated, the produced code calls for such an allocation. The frame which is allocated contains only 4 bytes for static/dynamic link. This inefficiency is a result of incomplete implementation of GSC. The allocation may, however, be removed manually to reveal the gain in efficiency that this technique would have caused. When lines 7–11 and 15–18 are removed, the program completes in 0.2 seconds.

The GSC implementation does not provide a garbage collector. Contrary to the version generated by Cim, the GSC version does not need a garbage collector in this case, because the frames are deallocated explicitly when exiting the corresponding procedure. Hence, the GSC program would not have invoked the garbage collector even if there had been one. Compared to Cim, it is beneficial to explicitly deallocate frames because the Cim-generated program had to spend 1.3 seconds performing 61 garbage collections.

Finally, table 6.2 shows the execution times of programs 3 a, 3 b and 4 on a SUN SPARC machine. Since, Lund Software House's Simula compiler was available on this platform, it has also been included.

6.3 Quality and efficiency compared to Cim

As mentioned in the introduction, it is important to distinguish between quality and speed achieved as a result of the fact that GSC has a more flexible interface than C, and the gain that stems from the revised run-time organization of GSC (like preallocation).

In the programs presented in this chapter, the C language does not seem to be the main reason why the programs generated by Cim tend to be slower than those generated by GSC. Instead, the main factor seems to stem from the differing run-time organizations that the two compilers use. The GSC programs gain much from the reduced number of activation frame allocations due to preallocation.

Listing 6.14: Code generated by GSC for program 4

```
P:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        movl    __lb, %eax
        movl    $0, 12(%eax)
        pushl   $4
        call    malloc
        movl    __lb, %ecx
        movl    %ecx, (%eax)
        movl    %eax, __lb
        movl    16(%ecx), %edx
        subl    20(%ecx), %edx
        movl    %edx, 12(%ecx)
        movl    (%eax), %edx
        movl    %eax, (%esp)
        movl    %edx, __lb
        call    free
        movl    %ebp, %esp
        popl    %ebp
        ret
Q:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $16, %esp
        movl    __lb, %eax
        movl    $0, 12(%eax)
        pushl   $44
        call    malloc
        addl    $16, %esp
        movl    __lb, %edx
        movl    %edx, (%eax)
        movl    %eax, __lb
        movl    $0, 8(%eax)
        movl    $0, 12(%eax)
        movl    $0, 4(%eax)
.L3:
        movl    __lb, %ebx
        movl    4(%ebx), %edx
        cmpl    $1000000, %edx
        jg      .L7
.L8:
        movl    12(%ebx), %eax
        movl    %edx, 36(%ebx)
        movl    %eax, 16(%ebx)
        movl    %ebx, 24(%ebx)
        movl    8(%ebx), %eax
        movl    %eax, 40(%ebx)
        movl    (%ebx), %eax
        movl    %eax, 20(%ebx)
        leal    20(%ebx), %eax
        movl    %eax, __lb
        movl    $.L8, 8(%eax)
        call    P
        movl    __lb, %eax
        movl    4(%eax), %edx
        movl    %edx, __lb
        movl    32(%edx), %eax
        movl    16(%edx), %ecx
        addl    %ecx, %eax
        movl    %eax, 12(%ebx)
        movl    4(%edx), %eax
        movl    %eax, 8(%edx)
        movl    __lb, %eax
        movl    4(%eax), %ebx
        incl    %ebx
        movl    %ebx, 4(%eax)
        jmp     .L3
.L7:
        movl    (%ebx), %eax
        movl    12(%ebx), %edx
        subl    $12, %esp
        movl    %edx, 12(%eax)
        movl    (%ebx), %eax
        pushl   %ebx
        movl    %eax, __lb
        call    free
        movl    -4(%ebp), %ebx
        movl    %ebp, %esp
        popl    %ebp
        ret
main:
.L10:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $20, %esp
        pushl   $16
        call    malloc
        popl    %ecx
        popl    %edx
        pushl   $0
        movl    %eax, (%eax)
        movl    %eax, 4(%eax)
        movl    %eax, __lb
        movl    $.L10, 8(%eax)
        pushl   %edx
        call    Q
        popl    %eax
        movl    __lb, %eax
        movl    4(%eax), %eax
        movl    %eax, __lb
        movl    12(%eax), %ecx
        pushl   %ecx
        call    OUTINT
        call    OUTIMAGE
        movl    __lb, %eax
        popl    %edx
        pushl   %eax
        call    free
        movl    $0, __lb
        movl    %ebp, %esp
        xorl    %eax, %eax
        popl    %ebp
        ret
```

Chapter 7

Summary and concluding remarks

This concluding chapter points out further considerations and potential refinements regarding GSC in light of the results of the previous chapter. The thesis is summarized at the end of the chapter.

7.1 Challenges

The following sections present a few apparent challenges that have become evident and have to be solved in future development of GSC.

7.1.1 Keeping the machine stack balanced

The assembler code listings of the previous chapter, contained a few instructions that were not commented in particular. These were mainly instructions that manipulate the machine stack in order to set up the parameters in conjunction with calls to the external stack-based functions, like `OUTINT` and `OUTIMAGE`.

As stated throughout this report, the machine stack has to be kept in "balance" whenever a coroutine switch may occur, i.e. the stack should be empty or the content should remain unchanged. In a final implementation of GSC, it will therefore be important that the front-end has full control over the back-end's tendency to generate stack-oriented instructions.

First and foremost, this issue is relevant in conjunction with the standard stack-based prologues and epilogue generated by the back-end, as will be discussed in the next section. In addition, the front-end must inhibit the back-end from performing optimizations that may leave the stack out of balanced at places where it should not be. One such potentially harmful optimization technique involves deferred pop-operations of stack elements until the end of a function, right before the return instruction. When appropriate, this technique decreases the number of interior `popl` instructions and, therefore, speeds up execution. This optimization can and should be disabled by the front-end (cf. the `-fno-defer-pop` command-line argument to the GCC C compiler).

7.1.2 Disabling the standard prologue and epilogue

The standard stack-based prologue and epilogue use the machine stack to store values like the return address, dynamic link, and so on. Listing 7.1 demonstrates the standard stack-oriented function prologue and epilogue that are generated by the GCC back-end for the INTEL X86 processors. This standard call / return sequence can be summarized as follows:

Listing 7.1: Traditional call / return sequence of INTEL X86

```
                ⋮
    call    P
                ⋮
--------------------------
P:  pushl   %ebp
    movl    %esp, %ebp
    subl    frame size, %esp
                ⋮
            interior of P
                ⋮
    movl    %ebp, %esp
    popl    %ebp
    ret
```

- The caller uses `pushl` instructions (or instructions with a similar effect) to push the actual parameters onto the machine stack. Since this implicitly moves the stack pointer (the `%esp` register) toward lower addresses, the stack pointer value decreases with an amount equal to the size of the parameter being pushed onto the stack.
- When all parameters are in place on the stack, the caller performs the activation through a `call` instruction. This instruction implicitly pushes the return address onto the stack.
- The callee starts off by pushing the address of the caller's frame onto the stack, so that it can put the address of its own frame into the frame pointer (`%ebp`) register.

- The callee then subtracts a certain amount from the stack pointer, to reserve space for its own use (like local variables), before proceeding to execute its interior instructions.
- When the callee has finished its operation, the standard epilogue restores the old stack pointer value where the old frame pointer was stored.
- Finally, the `ret` instruction performs the reverse operation of the `call` instruction, i.e. pooping the return address off the stack and jumping to that address.

As demonstrated by the assembler code in the previous chapter, the existing version of GSC produces call/return sequences that are partially stack-independent. The scheme is, however, not complete yet. The programs still use the `call` and `ret` instructions. Particularly, it is worth noting that although GSC generates an instruction that supposedly stores the return address into the heap-allocated frame, this address is not used when the callee returns. The reason is that the address stored there, is not always correct due to an unresolved discrepancy between the GSC front-end and the back-end.

In order to keep the stack in balance, the net effect imposed on the stack by a call or return sequence, has to be zero. This can be achieved either by refraining from using instructions like `push`, `pop`, `call`, and `ret`, which modify the stack, or by adding extra “counter-instructions” which moves the contents from the stack to its appropriate position in a heap-based frame and zeroes out the effect caused to the stack.

Listing 7.2 shows a finalized version of the call/return sequences that the current GSC implementation tries to follow. This scheme only uses instructions that do not involve the stack (except when calling the stack-based run-time library functions). Also, the scheme uses the `%ebp` register as the local block pointer, analogous to how it is utilized when activation frames are stack-based. This potential optimization is discussed shortly.

7.2 Potential improvements

The following sections presents suggestions to future improvements of the GSC implementation.

7.2.1 Rationalizing the local block pointer

As mentioned, listing 7.2 uses the `%ebp` register instead of the local block pointer variable `__lb` which is currently included in programs generated by GSC.

Listing 7.2: The caller's part of the revised call / return-sequence

```
    pushl   parameter frame size
    call    heap allocation routine
    addl    $4, %esp
    movl    %ebp, 4(%eax)
    evaluate parameter 1
    movl    evaluation result, p1(%eax)
                  ⋮
    evaluate parameter n
    movl    evaluation result, pn(%eax)
    movl    (%ebp), %ebp
                  ⋮
    movl    (%ebp), %ebp
    movl    %ebp, (%eax)
    movl    return address, 8(%eax)
    movl    %eax, %ebp
    jmp     procedure entry point
  point of return address
    pushl   %ebp
    movl    4(%ebp), %ebp
    call    heap deallocation routine
    addl    $4, %esp
```

As described earlier in this chapter, the active frame in a stack-based run-time environment, is pointed to by the base pointer register (`%ebp`). Hence, accesses to local variables and parameters are done relative to the address stored in this register.

The introduction of an explicit local block pointer variable like `__lb` is the result of having no back-end-provided and platform-independent way to operate the machine-specific frame-pointer (which on the INTEL X86 platform, is the `%ebp` processor register). Ideally, `__lb` is unnecessary as long as the machine in question offers a register designated to the purpose. If the front-end had been able to access the frame-pointer register directly, the need for a dedicated `__lb` variable would have been eliminated. Furthermore, eliminating this dedicated variable will remove the alias problem, which currently forces the generated programs to spend time reloading the value of `__lb` in case it has been changed through a pointer.

Under this scheme, the `%esp` register can continue to point to the top of the stack. Although this stack-pointer register will be used less frequently by programs generated by GSC, it will be ready for use by stack-based routines whose prologue will appropriately set `%ebp` to pointing to the stack on entry and restore its old value (pointing to a heap-based frame) on exit.

This will be convenient when activating run-time routines or external procedures that are stack-based and contains the traditional prologue and epilogue.

7.2.2 Adapting the back-end

So far, this and the previous chapter has focused much on the assembler code of a particular machine platform. One of the fundamental motivations for using the GCC back-end, is, however, platform-independence. Therefore, it is important to realize that the `%esp` and `%ebp` registers on the INTEL X86 platform, may have totally different counterparts, or maybe no counterparts at all, on other platforms. The question is how the GCC back-end can be modified to provide an ideal, but still platform-independent interface through which a front-end can handle the native frame and stack pointers and use them for purposesăwhich differ from what the back-end currently designates them to.

Of course, an even more ideal situation would be that the back-end also was able to handle heap-based frames itself. In that case, the back-end could apply the same logic that is already present for stack-based frames, when laying out their structures. This would relieve the front-end of this troublesome and conceptually low-level functionality. The back-end could even be given the responsibility of deciding when and how to preallocate space in heap-based frames.

Such a modified back-end, being aware of heap-based frames, would naturally also know how to generate appropriate procedure prologues and epilogues, where the front-end only provides the names of the allocation and deallocation routines to call.

7.2.3 Other optimizations

The opportunities for other optimizations are countless. For instance, if the frame of the outermost block is allocated prior to execution (i.e. at load-time), accesses to this (global) block-level will not have to traverse the static chain. Furthermore, the activation frames belonging to procedure and class declarations defined in this outermost block-level, will not have to include a static link, because it will never be traversed when the outermost block is accessed by other means.

This report has focused much on arranging for coroutines by allocating activation frames on the heap. However, many Simula programs never make use of the coroutine feature. These programs may equally well be stack-based, so a future version of GSC should be able to identify and exploit this situation. Such “single-stack” programs should potentially be able to compete with their C counterparts, except for Simula-specific aspects like range-checking of array accesses, class membership tests, garbage collections, and so on.

7.3 Suitability of the GCC back-end

The most apparent advantage of using the GCC back-end interface rather than C, is the ability to acquire the address of program labels and the byte offset of attributes (fields) inside compound data structures. Other than this, C is well-suited as a platform-independent intermediate language, and if these two features were included, it would be almost as expressive as the back-end interface.

In fact, the C front-end of GCC does support an extension of C where the address of a label may be acquired at run-time and stored in a pointer. It also supports an extension where the byte offset of a compound structure's field can be determined at compile-time.

However, there are still certain properties of Simula programs, that is not expressible, even through GCC's extended C variant. The generality of pointers (including pointer arithmetics) in C, gives rise to pointer aliasing which implies that the code optimizer has to account for the possibility that any memory location may (potentially) be affected when an assignment is made through a pointer. Simula does not have the same degree of pointer aliasing, but this fact is lost in the transition to C, and therefore cannot favor code optimization.

7.4 Summary

Chapter 6 revealed that code produced by GSC is more efficient than the corresponding code produced by Cim, at least for programs of limited complexity. The gain in efficiency stems partly from better low-level code optimization, but first and foremost, from the structural differences, of which *preallocation* appears to be the most important.

It is clear that a theoretically optimal Simula implementation cannot be based on C as intermediate representation. Still, the code efficiency and quality achievable through C, is fairly good. The GCC back-end interface enables a more expressive representation than C. Hence, the GCC back-end ultimately provides a better basis than C with respect to run-time efficiency. It is evident that interfacing the GCC back-end also arranges for a more efficient compilation process since generation and syntax analysis of C code is avoided.

On the other hand, the GCC interface is quite complex and it takes a considerable amount of effort to comprehend and master the necessary elements of it, in order to potentially exploit the expressive benefits provided by the GCC back-end interface.

Also, the back-end shares C's bias toward stack-oriented procedure activations, and has to be tricked by the front-end in order to refrain from utilizing the stack. Since this also applies to C, it does not disfavor the GCC back-end interface per se. However, it indicates that the back-end is not as well-suited for the Simula language as might have been hoped for.

Furthermore, C is more flexible in the sense that it can be used to interface an entirely different code optimizer and generator than GCC's back-end. Any decent C compiler will suffice.

The preallocation techniques could have been employed equally well by Cim using C, and therefore most of the gain in run-time efficiency cannot be ascribed to the GCC back-end interface. Because of this, the advantages of interfacing the GCC back-end rather than C, seem moderate. Improved compile-time efficiency is evident, but the improvements observed during run-time are subtle.

There are, however, several potential improvements that can be made, especially if the back-end itself is subjected to minor modifications. Rationalization of the local block pointer is a highly desirable optimization that is hopefully achievable by limited modifications.

In a more ambitious context, it is also desirable to modify the back-end so that it comprehends and handles heap-based frames as natively as it currently does with stack-based frames. This would enable the Simula front-end to elevate to a higher level of abstraction where most of the technical details concerning heap-based frames, are kept in the back-end.

All in all, the complete access to the entire GCC project's source code, provides numerous possibilities. Provided that enough effort is put into it, I am fairly confident that a very good, and complete, Simula can be made from the current version of GSC.

Bibliography

[1] Free Software Foundation (FSF). *The GNU project*. 1984–2001. http://www.gnu.org/.

[2] Free Software Foundation (FSF). *The GNU Compiler Collection (GCC)*. 1984–2001. http://gcc.gnu.org/.

[3] Richard M. Stallman et al. *Using and Porting GCC*. Free Software Foundation, 1991–2001. http://gcc.gnu.org/onlinedocs/gcc.html.

[4] Jack W. Davidson and Christopher W. Fraser. *Register Allocation and Exhaustive Peephole Optimization*. University of Arizona, 1984.

[5] Simula Standards Group. *Standard Simula*. 1986.

[6] Ole-Johan Dahl. *Runtime organisasjon for Algol/Simula*. Department of Informatics, University of Oslo, 1980.

[7] Sverre Hvammen Johansen, Terje Mjøs, and Stein Krogdahl. *Cim. A Simula-to-C compiler*. Department of Informatics, University of Oslo, 1987–2000. http://www.ifi.uio.no/~cim/cim.html

[8] Sverre Hvammen Johansen. *Et portabelt Simula-system bygget på C*. Department of Informatics, University of Oslo, 1987.

[9] Terje Mjøs. *Videreføring og testing av et portabelt Simula-system*. Department of Informatics, University of Oslo, 1989.

[10] Hans-Juergen Boehm, Alan Demers, and Mark Weiser. *A garbage collector for C and C++*. 1988–1999. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[11] Free Software Foundation (FSF). *GNU Bison. A YACC-compatible parser generator*. http://www.gnu.org/software/bison/.

[12] Free Software Foundation (FSF). *GNU gperf. A perfect hash function generator*. http://www.gnu.org/software/gperf/.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, USA, revised edition, 1988.

[14] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, Inc., third edition, 1998.