POLITECHNIKA CZĘSTOCHOWSKA
WYDZIAŁ ELEKTRYCZNY

Tomasz Borowik

Nr albumu: 92808

Alternatywny format zapisu kodu źródłowego

Praca Inżynierska

Promotor:
dr inż. Grzegorz Dudek

Częstochowa 2010

POLITECHNIKA CZĘSTOCHOWSKA
WYDZIAŁ ELEKTRYCZNY

Tomasz Borowik

Register number: 92808

Alternative source code storage format

BS Thesis

Promotor:
dr inż. Grzegorz Dudek

Częstochowa 2010

Table of Contents

Glossary

GCC

GNU Compiler Collection with front-ends for C, C++, Objective C and more, along with back-ends for x86, x86_64, IA-64, PowerPC, SPARC, ARM and more.

LLVM

Low Level Virtual Machine is project to develop a virtual instruction set and a corresponding compiler infrastructure that enables “effective optimization at compile time, link-time (particularly interprocedural), run-time and offline”[1].

Clang

A C/C++, Objective C/C++, front-end for the LLVM compiler.

AST

Abstract syntax tree is a “tree representation of a simplified syntactic structure” [1].

RTL

Register transfer language, an intermediate representation close to assembly language, that is well suited for operations performed by a compiler back-end.

CFG

Context-free grammar is a “grammar which naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap.” [2]

SC

The name of the language presented in this thesis.

SC file

A file that contains source code in the SC language.

SCF

The “file” representation of SC source code and any functions or structures related to it.

SCM

The “memory” representation of SC source code and any functions or structures related to it.

SCEdit

The editor application for SC files.

libsc

The main library for working with SC files.

F2M

The “File to Memory” subsystem of libsc which translates from SCF into SCM.

M2F

The “Memory to File” subsystem of libsc which translates from SCM to SCF.

OSCalc

The subsystem of libsc that calculates the position and size of objects as they will appear in the SCF format.

Pointer

A variable that holds an address of something in the applications address space

Offset

In the scope of SCF a number that is the position of something relative to the beginning of a file, or a variable that holds such a number.

---

1 http://llvm.org

Introduction

In recent years in the field of computer programming there is a noticeable trend towards a wider use of more high-level, type-safe, type-dynamic and memory-safe languages. These boast benefits like: lower barrier of entry, easier code management, safer code, faster program development and many more. There are also people who refute the advantages of these languages based on the conjecture that such languages cause programmers to have a worse understanding of the underlying system, which results in a misunderstanding of the performance and security implications of their code.

This thesis will present the SC programming language that aims to simplify programming and source code maintenance not through simplifying the language (the meaning), increasing abstraction, and taking over certain tasks like memory management, but through simplifying the form and providing a more advanced programming environment. SC tries to accomplish this goal mainly through the use of a special file format and an editor that "understands" the form and meaning of the language. Calling SC a programming language is somewhat inappropriate since it is not a language, however it fulfills a similar role. The C programming language forms the basis from which SC is derived, and the possibility of translation from C with binary equivalence (or at least strong similarity) is an important feature of SC. The practical part of this thesis includes:

- a library for reading and interpreting SC files
- an SC file editor, called SCEdit or scedit
- a front-end for GCC for compiling SC files
- a translator between different versions of SC files
- an AST consumer for Clang for translating C files to SC files

Chapter 1 presents the standard process of analyzing and compiling a text based language.

Chapter 2 presents the basic differences between SC and a standard text based language, as well as highlights some of the defining characteristics. It also contains an introduction to all the major parts of SC.

Chapter 3 is a through description of the file format used for SC files.

Chapter 4 is a description of some of the characteristics of the memory format along with libsc and translating between different versions of SC source code.

Chapter 5 presents the language from the perspective of the editor and consequently the programmer.

Chapter 6 is a summary of the current state of the project along with some concepts for future development and final thoughts.

1 Programming language and compiler theory

A programming language is “an artificial language designed to express computations that can be performed by a machine” [3]. The term “language” is somewhat inappropriate as there are also “visual programming languages”, which do not use words numbers, symbols, punctuation and grammar like a natural language. Therefore the term “programming language” in the scope of this thesis should be interpreted more as method of storing and transfering information that can be presented and modified by a human using software built for that purpose.

The most common classification is by how much a given language abstracts the underlying machine (these classes and examples are relative and the boundaries are ambiguous and arguable):

- high-level: Java, Lisp
- medium-level: C, C++
- low-level: assembly

There are a few types of programs that are associated with programming languages:

- compiler – translates from a higher-level language to a lower-level language
- decompiler – translates from a lower-level language to a higher-level language
- translator – translates between different languages of a similar level
- interpreter – executes source code written in a programming language
- assembler – creates machine code from assembly code

A compiler that translates a higher level language into assembly code will be of focus as it is the central part in most program production environments as shown in Figure 1.1. However, it is not the only one employed as creating, managing and generating machine code for complex applications would be impossible.

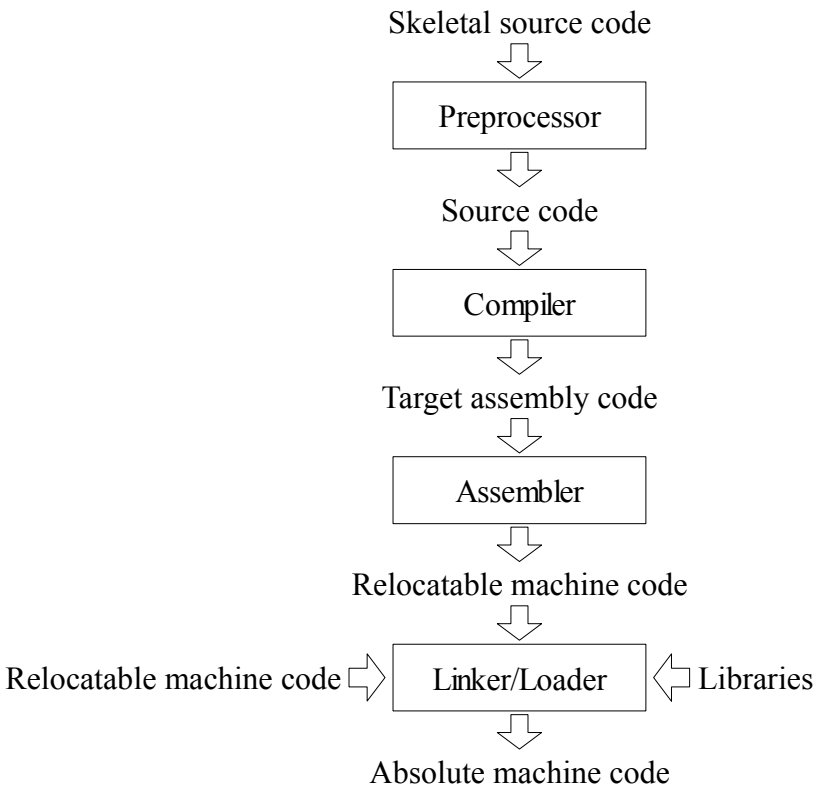


Figure 1.1: Program compilation stages in a standard production environment

A preprocessor is a program that performs (in the case of C) relatively simple transformation of its input like: text file inclusion, macro expansion and conditional inclusion or exclusion of text. This kind of a preprocessor is called a "lexical preprocessor" as it operates on tokenized character sequences and doesn't analyze the underlying language.

An assembler is also relatively simple as assembly code has a very straightforward structure and most work is simple translation of mnemonics into opcodes. The main abstraction feature is resolving of symbolic names into memory addresses.

A linker can take many object files containing relocatable machine code along with libraries, join them together resolving symbolic references, and using configuration scripts produce absolute machine code or alternatively again relocatable machine code.

A compiler is the most complex as it has to not only perform complex source code analysis, but also optimizations and then generate code that optimally fits a given target machine and produces expected results. Conceptually "a compiler operates in phases, each of which transforms the source program from one representation to another" [4], as shown in Figure 1.2. These phases can be grouped into a front-end and a back-end with an intermediate representation in-between often called a "middle-end". The reason behind this division is the possibility to reuse most of the source code when building compilers of the same language for different target architectures and vice versa.

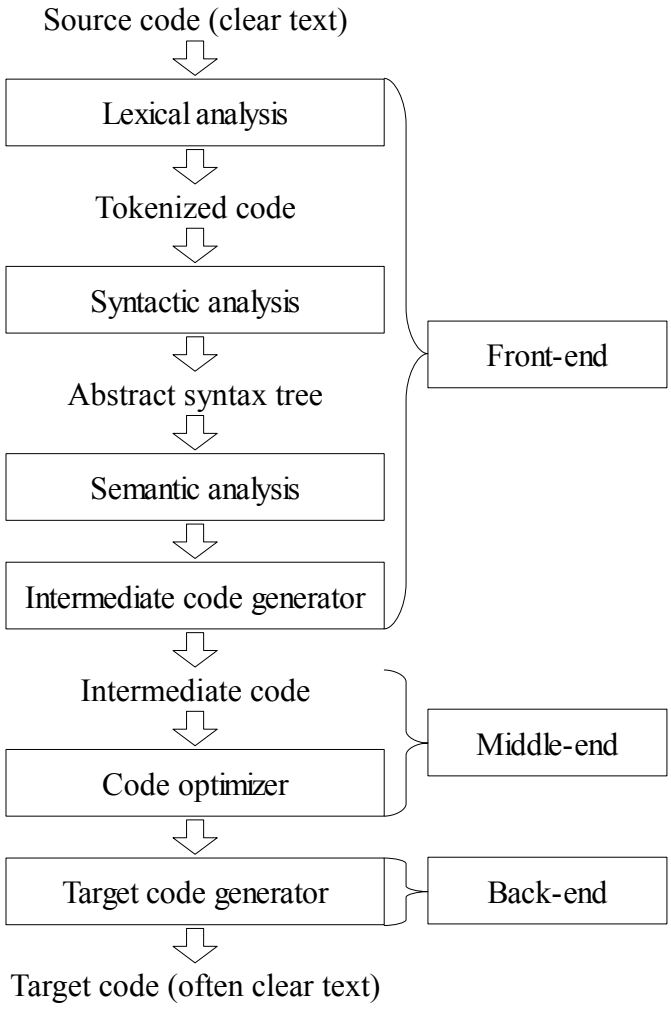


Figure 1.2: Phases of a compiler

Lexical analysis transforms clear text into tokens (usually with attributes). A token is a string of characters that is categorized as a certain symbol like: Identifier, Number, Assignment, Left_parenthesis etc. Text such as spaces, tabs and comments is usually eliminated from the output as keeping track of it is unnecessary.

Syntactic analysis builds a hierarchical structure usually called an AST (abstract syntax tree) from a series of tokens according to a formal grammar. An AST is "a tree representation of the abstract (simplified) syntactic structure of source code" [1]. In cases where a given series of tokens doesn't form a structurally correct sentence in a given language it reports an error. It is the most complicated and algorithmically intensive phase of the front-end.

Semantic analysis checks whether a given AST has a correct meaning in a given language. Commonly this means type checking and reporting any errors or warnings. However, explicitly stating everything in source code would often be too time consuming, which leads to the addition of implicit casting in cases where semi-compatible types are encountered.

An intermediate code generator will transform the syntactically and semantically correct AST passed to it into intermediate code. The complexity of this process depends mostly on the complexity of the source language and its discord with the intermediate representation. However, it is in most cases relatively straightforward as the process can generate excess code that will later be removed by the optimization phase.

Intermediate code is usually formed similarly to assembly code, however, instead of operating on registers and memory locations it operates on symbols and vaiables. The most common format is called “three-address code” in which each instruction has one destination location and an operation to perform on at most two source locations. However, intermediate code doesn't necessarily have to be limited to only such instructions, as it could also contain for example a function call instruction with any number of source locations (for arguments). Furthermore intermediate code could also be multiphase as certain abstractions may simplify certain optimization passes.

Code optimization can be the most complex part of the whole compiler. Optimizations are not limited to only the intermediate code, as an AST is also a good target for certain higher level optimizations. Intermediate code, however, is usually the most suitable since it is efficient to analyze and often contains substantial amounts of code produced by the intermediate code generator that is easy to optimize. Additionally any optimizations done on intermediate code will work for other target architectures and source languages. It is important to note that some algorithms do not guarantee that the resulting code is more efficient, or even that it will produce expected results as there are factors that cannot be accounted for by the compiler.

Target code generation is in many ways the most important phase as the quality of the output code strongly depends on it. The most important tasks that are performed in this phase are register allocation, instruction selection, and instruction ordering. Making optimal choices in this phase often requires through knowledge about the target CPU microarchitecture as the superscalar, out-of-order, and register renaming capabilities often differ substantially and although their aim is to optimize code on the fly they have limitations. To facilitate these tasks another representation is often used called a Register Transfer Language.

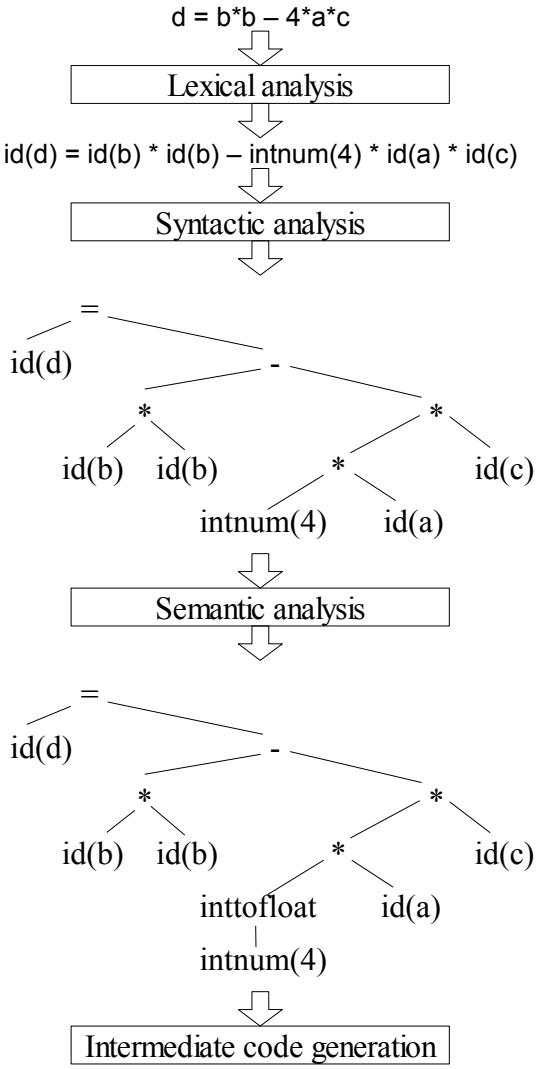


Figure 1.3: Example source code transformation in a compiler front-end

The transformation of code performed by a compiler front-end is best shown in Figure 1.3. First we have a simple sequence of characters with spaces. After lexical analysis we have a sequence of tokens (with attributes), which are in reality numbers and can be passed in a pipeline manner and processed on the fly. Then syntactic analysis creates an AST where we can observe a correct order of operators. Semantic analysis notices that all variables are of type float and the constant number is an integer therefore it adds an implicit conversion from int to float.

Continuing further an intermediate code generator could for example produce code similar to (1), which after the optimization phase could be transformed into much shorter code (2). The target code (3) and (4) in this example is real output from GCC for the x86_64 target architecture. Code (3) is compiled without any optimizations and it is immediately clear how many unnecessary operations are performed as compared to the optimized code (4). Also the ordering of instructions is such as to alternate between loads and operations.

```
tmp0 = id(b) * id(b)
tmp1 = intnum(4)
tmp2 = inttofloat(tmp1)
tmp3 = tmp2 * id(a)
tmp4 = tmp3 * id(c)
tmp5 = tmp0 – tmp4
id(d) = tmp5
```
(1)

```
tmp0 = id(b) * id(b)
tmp1 = intfloat(-4.0) * id(a)
tmp2 = tmp1 * id(c)
id(d) = tmp0 + tmp2
```
(2)

```
movss  b(%rip), %xmm1
movss  b(%rip), %xmm0
mulss   %xmm0, %xmm1
movss  a(%rip), %xmm2
movss  .LC0(%rip), %xmm0
mulss   %xmm2, %xmm0
movss  c(%rip), %xmm2
mulss   %xmm2, %xmm0
addss   %xmm1, %xmm0
```
(3)

```
movss  .LC0(%rip), %xmm1
mulss   a(%rip), %xmm1
movss  b(%rip), %xmm0
mulss   %xmm0, %xmm0
mulss   c(%rip), %xmm1
addss   %xmm1, %xmm0
```
(4)

Syntactic analysis

To perform syntactic analysis it is first necessary to define the formal grammar, which is commonly achieved using a notation called context-free grammar (CFG). In a CFG "clauses can be nested inside clauses arbitrarily deeply, but grammatical structures are not allowed to overlap" [2]. The simplest example of this is matching two types of parentheses, in "[(([])) [( )]]" the logical units nest within each other but do not overlap therefore it can be generated by CFG, whereas in "([)]"even though the the parentheses are balanced, they overlap, meaning a unit starts within another unit but ends within a different unit. CFG is built using rules (productions) that have the following form:

nonterminal → string of terminals and/or nonterminals, or an empty string

The character "→" has the meaning "can have the form". A terminal symbol is simply a token, meaning an indivisible element of the language that cannot be broken into smaller elements without loosing its meaning. A nonterminal symbol consist of terminal and nonterminal symbols (they can self reference) and therefore is represented by the production. To shorten the grammar definitions these two sets of rules are equivalent:

Expr → Expr + Num
Expr → Expr – Num
Expr → Num

Expr → Expr + Num | Expr – Num | Num

A CFG naturally describes the tree structure of most programming language constructs. In Figure 1.4 is an example grammar definition for analyzing a list of simple expressions (Є denotes an empty string), while in Figure 1.5 is an example sequence of tokens with a parse tree for that grammar.

```
Start  →  List EOF
List   →  Expr ; list
       |  Є
Expr   →  Expr +  Expr1
       |  Expr –  Expr1
       |  Expr1
Expr1  →  Expr1 * Term
       |  Expr1 / Term
       |  Term
Term   →  ( Expr )
       |  ID
       |  NUM
```

Figure 1.4: Left-recursive CFG for simple expressions [4]

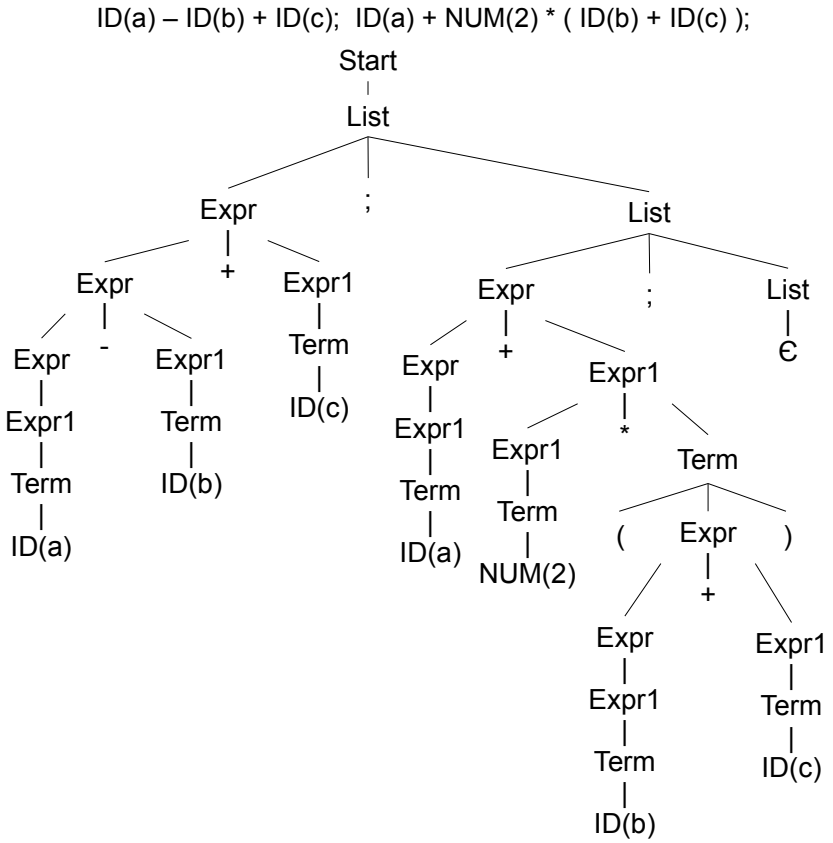


Figure 1.5: An example expression with a parse tree

The act of determining whether a sequence of tokens is a correct sentence and what its structure is, is called parsing. This process doesn't require the construction of a parse tree, however, it must be possible or else it is impossible to guarantee that the analysis is correct. There are many methods of accomplishing this, however, the simplest to imagine and program by hand is a top-down recursive-descent method. It works by recursively expanding nonterminals (starting with the Start nonterminal) into the right side of one of their productions until no more unexpanded nonterminals exist. Unfortunately this method cannot handle left-recursive productions like “Expr→Expr+Expr1” as it would expand the

nonterminal Expr into itself until a stack-overflow occurs. This can be fixed by turning such a left recursive production into an intermediate production paired with a right recursive production, as shown in Figure 1.6. The example grammar presented bellow is not equivalent to the previous as it will build a parse tree in which "+ - * /" are right associative. However, it is still possible to attain left associativity by inlining handling of _Expr into Expr and correctly "linking" consecutive "+/- Expr1" together.

Start	→	List EOF
List	→	Expr ; list
	|	Є
Expr	→	Expr1 Expr_more
Expr_more	→	+ Expr1 Expr_more
	|	– Expr1 Expr_more
	|	Є
Expr1	→	Term Expr1_more
Expr1_more	→	* Term Expr1_more
	|	/ Term Expr1_more
	|	Є
Term	→	(Expr)
	|	ID
	|	NUM

Figure 1.6: Right-recursive CFG for simple expressions [4]

A parser built on the above grammar would be called a "predictive parser" because the look-ahead symbol (the next token in the sequence) unambiguously determines which production is used for any given nonterminal. This is an important characteristic as it greatly increases the efficiency of syntactic analysis. Additionally it is not necessary to perform lexical analysis in whole and then perform syntactic analysis on the output, as syntactic analysis can actually "drive" the process by querying the lexical analyzer for the look-ahead token and consuming it when it has been matched (pushing lexical analysis forward). This has many benefits like: lower memory consumption, less memory allocation/deallocation and better cache utilization.

2 SC programming language

Comparing SC to other text based languages is hard due to its very different nature. The initial (admittedly uneducated) idea behind SC was to create a binary format that would significantly reduce compilation time. After further analysis and reflection the idea evolved into “reducing the barrier between the programmer and the compiler” and reduced compilation time became a secondary benefit. Therefore the file format became not a goal but actually a means to accomplish a goal.

The file format itself is relatively similar to an AST tree after syntactic analysis. However, this does not mean that lexical and syntactic analysis is just moved into a source code editor, as the whole process of compilation is divided and organized differently. In general a compiler front-end has to accomplish these tasks:

- preprocessing, compile time metaprogramming,
- creation of a global symbol table,
- creation of an AST,
- symbol reference disambiguation (analyzing symbol scope),
- checking for errors,
- adding implicit casts.

In SC a source code file already contains a symbol table, all statements and expressions are stored as a syntax tree, and most symbol references (internal to the file) are already established. Therefore the only actions the compiler front-end has to perform are: read the file and all the included files, perform compile time metaprogramming, resolve soft and inter-file symbol references, check for semantic errors and add implicit casts (this could be moved into the editor), and generate an intermediate representation. This should result in greatly improved efficiency of the compiler front-end, which can be anywhere from 2% to 20% of compilation time (tested using -ftime-report GCC option) depending on the source code, optimizations, and state of the disk cache.

Moving onto the source code editor, it has to at least build a global symbol table, properly create the syntax trees of statements and expressions, and (in part) perform symbol disambiguation. Although this could be done using lexical and syntactic analysis similar to that in a standard compiler, that would defeat the goal of SC as it would just be moving a barrier from the compiler into the editor and introducing more complexity to an already complex problem. Instead the editor exposes the syntax tree and structure of the code to the programmer in a more direct fashion. All of this obviously requires the editor to “understand” substantial parts of the code. Having all of this infrastructure it is a natural progression to extend the capabilities of the editor to give as much additional feedback as is needed by the programmer and to introduce as much flexibility in code presentation as can be useful.

Although full lexical analysis does not exist in SC, the editor or compiler does for example need to transform a sequence of characters into a number, however, this action should not be considered lexical analysis as it is already known what “token” the sequence is.

Semantic analysis can be done by both the editor and the compiler. Currently it is done in parts by both, as the editor gives feedback about types to the user and filters symbols, while the compiler does implicit casts.

Looking from the side of contemporary text based languages there are many “tricks” used like keywords or using the same symbol to denote two different operations. Keywords are words which have a particular meaning in a language. In C “a keyword is a reserved word, which identifies a syntactic form” [5] and it cannot be used to denote types, variables or functions. The symbol “-” is used for both binary subtraction and unary negation, which is unambiguous due to how syntactic analysis works. Both of these techniques are completely inapplicable to SC, which will become clear after reading Chapter 3 The file format.

Source code representations

There are currently two main storing methods: "file" and "memory". The file method is the format used in SC files meant for storing data in a continuous buffer, it's optimized for small size without too much hindering of reading/analysis speed. However, it is impossible to efficiently modify it, due to the complexity of relocations. The memory method uses standard dynamic memory management, static sized structures and pointers. This means that the memory needed to hold such a tree is up to a few orders of magnitude larger. However, it can be efficiently modified and should be faster for advanced analysis.

Internally all functions and types associated with the "file" method are prefixed with SCF, and those associated with the "memory" method are prefixed with SCM, and I will henceforth refer to these representations as SCF and SCM.

Main library

The central part of the SC language is libsc. Its duty is to simplify access to information stored in files and handle or provide templates for handling code analysis. It is currently divided into three main parts, one to deal exclusively with the “file” representation of SC code, one to deal with the “memory” representation, and one that translates between these representations. There are currently many exceptions and inconsistencies due to an early stage of design and development. The library itself does not currently form a full layer of abstraction and probably never will since the structure of the file is in many cases as important as the structure of the language.

The translation from SCM to SCF is handled in two stages, the first is to calculate the sizes and offsets of certain objects (the OSCalc subsystem), the second is the process of writing the data into a buffer (the M2F subsystem).

The translation from SCF to SCM can be much more granular and “on demand”. It is currently handled by functions called SC_F2M_(object_class_name) that take a pointer to an SCF object, see if it was previously translated (by checking an associative array), if yes they return the retrieved pointer to it, if not they translate it into SCM also calling an F2M function for every reference to other objects, and then save the pointer to the complete object in an associative array and return it. The main disadvantage of this method is that the translation will jump around the file, which will hinder caching and the possibility of concurrent file reading and translating.

The associative array is actually implemented by allocating a buffer with a scaled size depending on the size of a pointer on the host system and using the offset (also scaled) of the SCF object to index into it. This method has O(1) complexity at the expense of wasting space. It can also be useful for other situations such as holding temporary data tied to SCF objects, which could enable some applications to work on just SCF.

Some parts of libsc are a code template or require static linking while others do not, therefore libsc will probably be divided into separate parts in the future.

Source code editing

SC files contain many offsets and sizes, which need to be changed whenever any part of the file is modified. Therefore it is impossible for a human using a hex editor to modify them efficiently. However, debugging the file or extracting information is possible although time consuming.

SC file editor has to not only present the information in a correct and readable fashion but also create a means for efficient editing. There are also many additional complications resulting from the architecture of the language like: variable scope, operator priority, handling "derived" types (pointers, arrays), proper behavior upon object deletion. Many of these are handled by a compiler front-end in a text based language and therefore in SC an editor should be considered a part of the language specification.

On one hand this can be considered a needless complication since a language developer has more parts to design and maintain. On the other hand this creates many possibilities for features that would be unfeasible in text based languages, it also gives a programmer a guarantee that another programmer working on the code has the same capabilities in his editor.

GCC front-end

GCC has a few internal code representations, as shown in Figure 2.1. The two of most interest are GENERIC and GIMPLE. GENERIC is most equivalent to an AST and is classified as a part of the front-end. However, it is not language specific and is actually shared between the C, C++, Java and Fortran front-ends. GIMPLE is a simplified version of GENERIC, which uses a three-address representation and is equivalent to intermediate code.

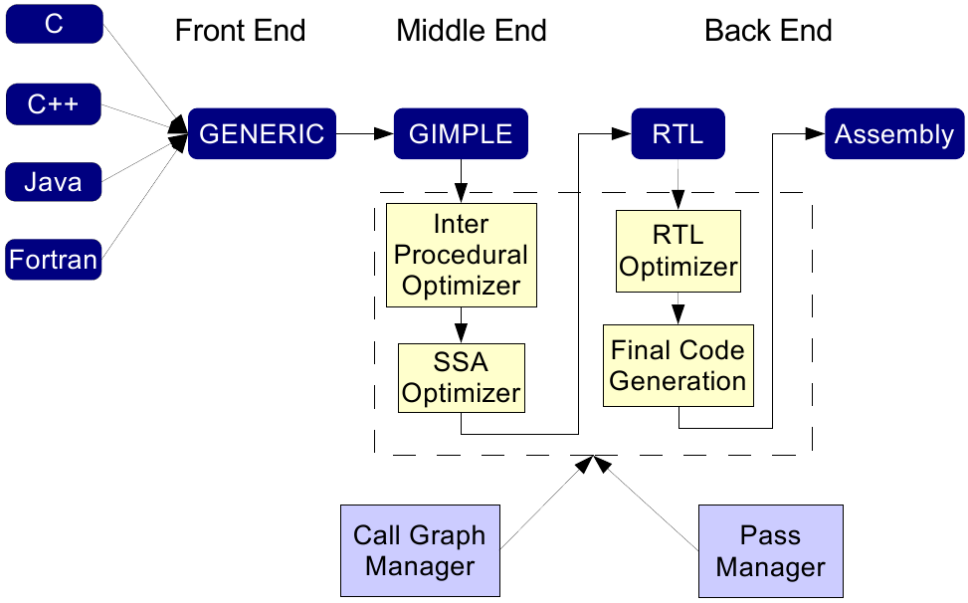


Figure 2.1: GCC pipeline [6]

The current implementation of the SC GCC front-end is in an early stage of experimentation. It reads an SC file and all the included files, translates them into SCM, performs inter file symbol reference disambiguation, and then translates SCM into GENERIC performing implicit casts on the fly. The target idea is to translate SCF directly into GENERIC code. However, such an implementation strongly depends on the internal

structure of the file, which would hinder any major experimentation with the format itself. It may be possible to even translate SCF directly to GIMPLE code, however, as SC aims for translation from C to SC with binary equivalence it is much simpler to work with GENERIC code.

Clang AST consumer

The main purpose of this tool in its current form is to aid with the development of the language. A source C file is first preprocessed, then an abstract syntax tree is created, and that AST is translated into SCM and then into SCF, which is written into an output file. The clang AST consumer has been used mostly to produce flattened SC headers from standard C headers.

2.1 Features

Some of the more notable SC features and characteristics, some of which are potential and unproven, those marked with "*" are not yet implemented:

- faster compilation/interpretation,
- impossibility of writing a syntax error,
- eliminated problems like too many keywords, and not enough symbols,
- ease of extending and further development,
- potentially smaller files,
- less duplicated information,
- *advanced syntactic find & replace,
- separation of code structure and code presentation,
- many different views of a single file, for example an objective and functional grouping of functions, or imaginary splitting into multiple files,
- very advanced code presentation capabilities, syntactic assistance:
  - code highlighting,
  - code folding,
  - code completion,
  - tree view of expressions,
  - automatic aligning,
  - *automatic smart line wrapping,
  - filtering of unneeded information, for example hiding long comments or explicit type casts,
- very advanced semantic assistance:
  - code highlighting,
  - inlining of additional information:
    - implicit casts,

    - expression types,
    - inferred types,
    - both weak and strong typing at the same time,
- code presentation is editor dependent, for example people working on the same code can have different indentation width,
- source code itself can become the full documentation,
- immediate editor feedback about the code like errors, types, casts etc,
- less time wasted on needless compilation just to fix syntactic and semantic errors.

Information duplication hinders experimentation, modification and refactoring. It can take many forms the simplest being the need to repeat the name of a variable or a type when referring to it, which is a requisite of text languages. In SC variables, types and other objects are (when possible) referred to directly using a pointer.

A different example is the need to specify a function type both in the prototype and the implementation, which forces the programmer to change it twice in different files. There is no technical reason why a compiler should require the function type in an implementation if it has a previous prototype (assuming function parameters were named). In practice, however, not having the function type along the implementation makes the code extremely unreadable. SCEdit can solve this by simply inlining the function type declared in the prototype (which could even be in a different file).

A similar case happens with inferred variable types that depend on the type of the first assignment. These are most useful for expressions without side effects that get evaluated only once but are used a few times in a block, where remembering or searching for the exact type of the expression would be more troublesome than simply copy & pasting the expression 5 times. But again such variables can cause confusion when proof reading, analyzing or learning new code. SCEdit can again solve this problem by simply inlining the inferred type of the variable.

These examples may not seem like much, however, given hundreds of thousands of lines of code managing duplicated information can become very time consuming. This is especially visible when doing code derivation (meaning a copy & paste & modify of existing code) where one has to do a lot of simple name modifying and just a bit of behaviour modifying. A good example is SCEdit's source code where there are many polymorphic objects. When a programmer needs to add another he just copies one that is as close as possible in structure to the new one, does a find & replace on selection to change the names and proceeds to modifying the behaviour and structure. Due to a very objective nature of the code, objective grouping and proper naming, a single find & replace will handle most of the work. However, this is not the case for libsc where functions are grouped by the subsystem. Also find & replace can cause much harm if not done with care.

Unfortunately not all is in favor of SC, the first problem is the lack of find & replace in current implementation which basically makes any mass name modify a much more time consuming task. There also exists a less obvious problem with a field reference like fnc.pName in which there is a pointer to the exact field pName it the exact class that was the type of fnc. When one changes the type of fnc to something else that also has a field named pName, the field reference still points to the wrong pName, which is an error and the code will not work whereas in C at that moment it would work. This can be fixed in a

few ways like a shortcut for reevaluation of field references in the selected code, which is the simplest solution. Alternatively a syntactic find & replace could be used which would search for every field reference like expr_of_type(tSCM_Var*)->tSCM_Fnc:pName and change it to expr_of_type(tSCM_Var*)->tSCM_Var:pName. This find & replace is however still just an initial concept.

The separation of code structure and presentation is a key component of the language and the basis of many other features. In general it should be interpreted in the sense that the way the source code is presented to the programmer can be different than the way it is stored and structured. For example a function is presented as a whole with its name type and body, whereas in the file these 3 objects are actually stored separately in different parts of the file, or expressions which are stored in a prefix format but are presented infix or as trees. The general benefit of this is that the code can be presented in a programmer friendly manner and at the same time be organized in a compiler friendly manner. Unfortunately it also means that the editor application has to be much more complex.

While code folding, highlighting and completion are not considered advanced features, their implementation is often complicated, requires background indexing, can consume much memory and can cause editor sluggishness. Whereas in SC implementing code indenting, folding and syntax highlighting is quite trivial and doesn't require much more overhead or memory. Code completion is even more interesting because it is actually (partially) necessary for editing of the code.

Code aligning in standard programming languages is the act of adding tabs or spaces to make parts of similar consecutive lines align where they did not align due to different lengths of symbol names. It is most often used in structs, enums, constant table initilizers or consecutive function prototypes and defines. The benefit is greater readability, the drawback is that the programmer has to waste time inserting tabs/spaces, which for files with a few thousands of defines or enums or hundreds of functions is very time consuming. Also when adding new lines if one happens to require more space in a column than was previously assumed, it is necessary to insert more characters into every other line, or leave that one line misaligned. The same problem can happen with name changes. The use of tabs also depends on a certain tab width and the alignment will in some places break if a different tab width is used.

In SC the placement of parts is calculated by the editor, therefore it is possible to do forward or backwards scanning to automatically establish a smallest necessary alignment. In the case of struct and enum fields it is already implemented due to the simplicity of it. However, with separate objects there is a need for a common mechanism that calculates the alignment once for a range of objects and not once per object. The reason for that is that without such an optimization the complexity would be $O(n^2)$, which could cause major slowdowns.

Semantic assistance, in the context of SC, is the act of inlining any additional information that helps with understanding of the meaning of source code and not just the structure. The simplest and most useful form of this is displaying the type of an expression, and the further extensions of it like displaying what types a function call expects, displaying implicit casts and marking semantic errors. There are also many smaller features that go along these like the ability to jump to the type declaration or displaying dealiased types. It is hard to overestimate the usefulness of these, however, that is only assuming the

given information is 100% correct in all cases, which although hard to achieve should be possible in SC.

What is meant by “both weak and strong typing at the same time” is that the editor can inline information whether an implicit cast is acceptable only in weak typing and not strong typing. This gives the programmer a granular choice whether he accepts unsafe casts. Although in C it was also theoretically possible since a warning was printed, the reality is that given more than small amounts of code even noticing errors between those warnings can become a burden.

In C this can be solved either by writing explicit casts or disabling warnings. The first solution forces the programmer to write more code, which not only limits his time for experimentation but also makes it harder as there is more duplicated information, it can also make the source code less readable. The second solution has the disadvantage of disabling these warnings everywhere, so a programmer writing new code has to rely more on his memory and understanding of the given code.

SCEdit can also further remedy the problem by a shortcut that will insert an explicit cast same as the implicit one. This will allow a programmer to quickly get rid of the unsafe implicit cast marker.

2.2 Drawbacks

These are the most notable drawbacks, those marked with “*” are just deficiencies in the current implementation and not limitations of the language itself:

- lack of lexical preprocessing,
- retrieving information from corrupted files is much harder,
- special editor is necessary,
- layout calculation is complex and could cause slowdowns/sluggishness,
- potentially larger files,
- no support from current applications for:
  - file comparing,
  - patching,
  - repositories,
- *lack of history in editor,
- *lack of copy & paste between different files, and editor instances,
- lack of lexical find & replace,
- *lack of syntactic find & replace.

Retrieval of information from corrupted files is not of much concern as practically all proper production environments have automatic backups with safeguards against corruption.

The need for a specialized editor shouldn't be a considerable adoption barrier because such software is already used in most production environments. Also SCEdit is almost completely GUI library and OS independent, which makes it very easy to port.

The complexity of layout calculation is hard to predict. However, even assuming it exceeds a reasonable amount of time in its current concept (the layout is calculated for the whole view space), there are many relatively simple techniques (which should not hinder ease of editing) that can be used to limit the scope of recalculation only to the currently viewable items and their closest surroundings.

Comparing file size is not yet possible on any real world code since there is none, and the automatic C to SC translator works on preprocessed code so all the headers are included. Another difficulty is that the size difference also depends on the code itself, names with common long prefixes work in favor of SC.

The lack of history in SCEdit (undo, redo) is a very serious and frustrating deficiency since it's currently quite easy to delete or replace a whole branch of an expression with a single wrong key-press. Therefore it is considered necessary to implement it before this language can be usable for standard programming. Unfortunately it will be complicated to implement and may exhibit high memory usage along with poor granularity.

Copy & paste between different files requires reevaluation of symbol references on the fly, which hasn't been implemented due to low importance from a design and development perspective. Implementing copy and paste between different editor instances may require another storing method, which would be a variant of SCF. This makes this feature undesirable from a maintenance perspective. Also due to the plan to make SCEdit flexible with multiproject, multiwindow and tabbed editing, this feature will hopefully be of little importance.

A standard implementation of find & replace operates on pure text, which, same as lexical preprocessing, is impossible in SC. Syntactic find & replace should be possible, however, there are many difficulties in designing such a feature.

3 The file format

The SC file format can be described as:

- binary - not human readable text, written as a sequence of bytes,
- non-linear - it is not treated as a sequence of bytes and does not have to be interpreted as such, certain parts can be skipped without interpretation,
- objective - the data is written more or less as structures (many have a dynamic size) which can contain arrays, lists or offsets to other objects,
- dynamically typed - these objects have a 16bit “Code” field which unambiguously describes the meaning and format of the following bytes which correspond to this object.

The codes used for dynamic typing are actually multipurpose as they also designate language constructs. Some objects can actually contain just the 2 byte Code. It is also possible to embed flags in them, for example the most significant bit is currently used to distinguish between objects stored in SCF and SCM (used by SCEdit). Codes can be optimized by aligning them in such a way that instead of range checking it's possible to use mask and compare to distinguish between classes/groups of objects.

The file format went through many significant changes since the initial implementation like for example the introduction of radix trees for symbol tables. During the development time it was always possible to relatively easily build an automatic translator between these formats by exploiting the common ground of the SCM representation, which is further discussed in Chapter 4 The memory format and SC library.

The main parts of a file in its current form are (respectively to the storing order):

- header which contains the format version and pointers to other parts,
- include/import table,
- macro symbol table,
- named type symbol table,
- derived type list,
- node symbol table,
- main heap which contains the private data of symbols,
- list with views of this file.

There are currently 6 main classes of objects:

- Meta/Macro: expression_equivalent
- Types: builtin, struct, union, functions, enum_type, pointer, array
- Nodes: variable, function, enum_field
- Statements: block, if, loops, switch
- Expressions: = + - * /, function call, dereferences, type_cast
- Views: blocks of global declarations

Codes, structures and functions are usually named in an object oriented style, with unneeded fields omitted: “NameSpace_SubSystem_Class_Object_SubObject_Action”. The class is usually shortened to a single letter (M, T, N, S, E, V).

The codes used for dynamic typing are:

```
enum {
dSC_NULL = 0,

dSC_SymTab,                 //an intermediate object in a radix symbol tree

dSC_M_START,
        dSC_M_E,            //a macro that evaluates to an expression
dSC_M_END,

dSC_T_START,
        dSC_T_BuIn,         //a builtin type like: 32bit signet integer, or 64bit float
        dSC_T_Enum,         //an enum type, it containes a list of N_Enum objects
        dSC_T_Class,        //currently same as a struct in C, containes a list of T_Class_Fld objects
        dSC_T_Class_Fld,    //a single field within a class, struct, union. has a pointer to a type and a name
        dSC_T_Struct,       //structure and meaning same as class
        dSC_T_Union,        //structure same as class but fields use the same space (same as union in C)
        dSC_T_Fnc,          //a function type, has a return type and a list of T_Fnc_Par objects
        dSC_T_Fnc_Par,      //a parameter to a function type, has a pointer to a type and a name
        dSC_T_Ptr,          //derived type: pointer to a type
        dSC_T_Array,        //derived type: constant array of size (constant expression) of a type

        dSC_T_Imp,          //an import type, structure similar to a class, has a list of T_Imp_Fld objects
        dSC_T_Imp_Fld,      //a field in an import that can be elem dereferenced
        dSC_T_Alias,        //alias type, similar to typedef type_a type_b;
dSC_T_END,


dSC_N_START,
        dSC_N_NEW,

        dSC_N_Var,          //global variable declaration, has a pointer to a type, a name, and a pointer to an
                            //initializing constant expression
        dSC_N_Var__Init,    //used only in SCF to distinguish a global variable with an initializing expression
        dSC_N_Enum,         //an enum field
        dSC_N_Enum__Init,   //used only in SCF to distinguish an enum field with an explicit value (constant
                            //expression)

        dSC_N_Fnc,          //a function declaration with a pointer to the type, and S_Block statement
        dSC_N_Fnc_Ext,      //a function prototype without a pointer to S_Block

        dSC_N_Imp,          //a node import, contains only the name of the object
        dSC_N_Alias,        //alias node

        dSC_N_Cond,         //first experimentation with a conditional node, contains a list of constant
                            //expressions and corresponding Node objects
dSC_N_END,


dSC_S_START,
        dSC_S_Label,        //equivalent to label_name: in C, just a stub atm
        dSC_S_GoTo,         //equivalent to goto label_name; in C, just a stub atm

        dSC_S_Ret,          //equivalent to return expression; contains a pointer to an expression

        dSC_S_If,           //equivalent to if () {} else {}; contains an expression and 2 pointers to S_Block
                            //objects; if the second S_Block pointer is null it's equivalent to if () {}
        dSC_S_If__El,       //used only in SCF to distinguish an if statement with an else block

        dSC_S_Loop_0,       //while () {}; loop, pontires to an expression and S_Block
        dSC_S_Loop_1,       //do {} while(); loop, same structure as S_Loop_0

        dSC_S_Switch,       //currently not fully equivalent to a switch () {}, contains a list of S_Switch_Case
                            //objects
        dSC_S_Switch_Case,  //contains a constant expression and a pointer to S_Block

        dSC_S_Brk,          //equivalent to break; in C
        dSC_S_Cnt,          //equivalent to continue; in C

        dSC_S_Block,        //equivalent to { }, contains a list of local variable declarations and statement
                            //objects.
```

```
	dSC_S_W_Ign,	//a white ignore statement which holds a pointer to a statement object, can be inserted to
			//ignore a given statement, equivalent to using // or /**/ to comment out code
	dSC_S_W_V,		//a white statement, enables embedding of View objects within S_Block
dSC_S_END,


dSC_E_START,
	dSC_E_NEW,		//designates a new expression has a pointer to an expression since it can be
				//inserted before a certain expression
	dSC_E_Sym,		//a soft symbol reference, contains the name of the symbol, when a node or a local
				//variable is deleted each reference to it is replaced by an equivalent E_Sym object
	dSC_E_Cst_BuIn,		//a constant builtin value (a number)
	dSC_E_Cst_Str,		//a constant string value equivalent to "" in C
	dSC_E_N,		//a reference to a node, contains a pointer to a Node object
	dSC_E_Par,		//a reference to a function parameter, contains pointer to T_Fnc_Par or T_Imp_Fld
				//object

	dSC_E_Cast,	//a cast, contains a pointer to a type and an expression, equivalent to (type)expression
	dSC_E_Conv,		//a convert, currently equivalent to cast

	dSC_E_Arg1_START,	//single argument operators
		dSC_E_Ptr,	//equivalent to &expression
		dSC_E_DePtr,	//equivalent to *expression

		dSC_E_Neg,	//equivalent to -expression

		dSC_E_IncPre,	//equivalent to ++expression
		dSC_E_IncPost,	//equivalent to expression++
		dSC_E_DecPre,	//equivalent to --expression
		dSC_E_DecPost,	//equivalent to expression--
	dSC_E_Arg1_END,

	dSC_E_Arg2_START,	//two argument operators
		dSC_E_Ass,	//equivalent to expr = expr
		dSC_E_Add,	//equivalent to expr + expr
		dSC_E_Sub,	//equivalent to expr - expr
		dSC_E_Mul,	//equivalent to expr * expr
		dSC_E_Div,	//equivalent to expr / expr

		dSC_E_And,	//equivalent to expr & expr
		dSC_E_Or,	//equivalent to expr | expr
		dSC_E_XOr,	//equivalent to expr ^ expr

		dSC_E_ShL,	//equivalent to expr << expr
		dSC_E_ShR,	//equivalent to expr >> expr


		dSC_E_LAnd,	//equivalent to expr && expr
		dSC_E_LOr,	//equivalent to expr || expr

		dSC_E_EQ,	//equivalent to expr == expr
		dSC_E_NE,	//equivalent to expr != expr
		dSC_E_LT,	//equivalent to expr < expr
		dSC_E_GT,	//equivalent to expr > expr
		dSC_E_LE,	//equivalent to expr <= expr
		dSC_E_GE,	//equivalent to expr >= expr

		dSC_E_DeArr,	//equivalent to expr[expr]
	dSC_E_Arg2_END,

	dSC_E_Call,	//equivalent to function_name(expression,expression,etc.), contains an array of pointers to
			//expressions
	dSC_E_Elem,	//equivalent to expression.field_name or expression->field_name, contains a pointer to
			//T_Class_Fld or T_Imp_Fld, and an expression

	dSC_E_M,	//similar to using a MACRO or MACRO(expression, expression), used to call M_E

	dSC_E_V,	//used as a root object of every expression can also directly be used in a statement list
dSC_E_END,
};
```

It is important to note that a list in SCF is not a standard linked list. There are no additional pointers to the next or previous objects as the objects are simply stored one after the other. The reason why it's not called an array is because they can have different types and sizes. Which means that accessing an n-th element requires going through all the previous objects. Another implication is that it is necessary to be able to calculate the size of an object to move to the next one and it's also impossible to move backwards. Therefore even though the format is not that of a linked list the usage constraints are closer to a singly-linked list than an array.

The format itself can store any characters in names, including utf-8 (although it's currently not supported). This means that one could theoretically have a type named “tSCF E +” or a variable named “-width” and it can be stored and operated on by the editor and compiler front-end correctly. Obviously naming a variable “-width” or a type “*tSCE” apart from potentially not working with the rest of the compiler and linkers could cause a lot of misinterpretations and should be blocked or warned against by the editor. On the other hand certain variables or fields are meant to store values like “1/m” (m being mass of a rigid body) and if the editor could make it visually distinctive that it's a whole variable then such a name might be better than “oneoverm” or “oom” or “invm”.

3.1 The symbol table

Most named structures have a similar format to simplify and speed up processing of symbol tables and symbol references. This unfortunately forces many objects to store their data on a heap, which has the downside of less compression or even inflation as compared to a list of complete objects with full names.

```
struct tSCF_Sym
{
        tSC_code Code;          //2 bytes
        tSC_off Priv;           //4 bytes
        u08 Len;                //1 byte
        u08 Name[0];            //0-255 bytes
}
```

The Len field stores the number of bytes in the Name field. Name holds the postfix added to form the complete name of this object or its children, this string is not null terminated. The Priv field is used for private data, which can be for example a Type and Size of a builtin type, an offset to another type, or an offset into the heap where the real private data is stored (used in cases where 4B isn't enough to store it).

When the Code is SymTab an object is an intermediate node in the radix tree, meaning it only holds children nodes and isn't an object used in the source code. The Priv field then contains an offset within the symbol table space that is the first object in a sequence of child nodes terminated by code NULL. The order of storing the sequences of child nodes is important since it allows more efficient Name retrieval when having the objects offset.

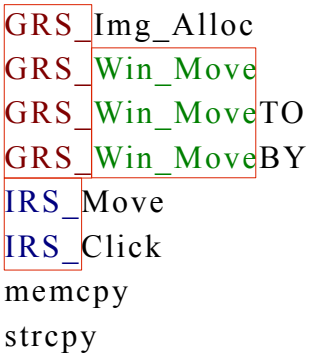


Figure 3.1.1: Function names with marked prefixes

An example symbol table, containing functions from Figure 3.1.1, will have three intermediate SymTab objects as highlighted. These symbols will be stored in a continuous buffer as presented in Table 3.1.1. Objects with Code N_Fnc have "__" in their Priv field which stands for an offset outside of this symbol table. Also an object can have the Name field empty like the one at offset 83.

Offset in buffer	Code	Priv	Len	Name
0	SymTab	50	4	GRS_
11	SymTab	110	4	IRS_
22	N_Fnc	__	6	memcpy
35	N_Fnc	__	6	strcpy
48	NULL			
50	N_Fnc	__	9	Img_Alloc
66	SymTab	83	8	Win_Move
81	NULL			
83	N_Fnc	__	0	
90	N_Fnc	__	2	TO
99	N_Fnc	__	2	BY
108	NULL			
110	N_Fnc	__	4	Move
121	N_Fnc	__	5	Click
133	NULL			

Table 3.1.1: Radix symbol tree

Looking at objects under offsets 0 and 11 we have two intermediate nodes and any object that has the first one "GRS_" as a prefix is stored between the offsets 50 and 110. Any object having IRS_ as a prefix is contained between offset 110 and the end of the whole table which is 135. This means that it is possible to skip whole branches of the tree when retrieving the name of an object, something that is impossible when not using this kind of ordered storing or additional numbering.

In order to retrieve an objects full name (having the offset) it is necessary to first establish in which table it is stored. This can be done in a few ways. If the code is a of class Macro then the node is in the global macro symbol table or if it's a T_Class_Fld the Name field is the full name of this object. Alternatively it can be done by checking the range of a symbol table to see if the offset of an object falls within that table.

3.2 Types

There are two kinds of types: named and derived. Pointers and arrays of constant size are derived, which means that their “name” is directly derived from their structure. Therefore their names are not stored in SCF, and if one needs the full name (which for a pointer to type “tSC” would be “*tSC”) one has to call a function to write the name into a buffer.

Classified as named types are: built-in, enum, class, class_field, struct, union, function, function_parameter, alias. They are all based on the symbol structure and, except for class_field and function_parameter, are stored in a radix tree. Those that contain a variable number of elements store them in the main heap as a list (enum, class, function). Class struct and union types have exactly the same structure and all use class_field, the difference lies only in how the compiler treats them. The class and struct types currently have the same meaning, neither has methods, inheritance or any other C++ features the existence of both was only for future proofing.

3.3 Expressions

In a source file most space is consumed by expressions. In C a single binary operation should be on average 3B (1B for the op, 2B for spaces). Storing them as trees where each operation holds two offsets to its operands would cause tremendous file size inflation (2B code, 8B for offsets). Therefore they are stored in a prefix format where only 2B are needed for the Code.

Standard C code (14bytes):

```
d = a * b + c;
```

Prefix format:

```
= d + * a b c;
```

SC code (30bytes):

Field type	Code	Code	Offset	Code	Code	Code	Offset	Code	Offset	Code	Offset
Size in bytes	2	2	4	2	2	2	4	2	4	2	4
Data	E_Ass	E_N	d	E_Add	E_Mul	E_N	a	E_N	b	E_N	c

Figure 3.3.1: An example expression in C and SC

The difference in size as shown in Figure 3.3.1 depends mostly on the length of identifiers, if they are on average around 5-6B the used space should be on a similar level. Translation of a tree stored in this format requires a recursive approach where the size of the operation and its operands is returned (or a global offset is used). An alternative format would be to arrange an expression in a postfix manner. This could prove faster when interpreting such an expression for immediate execution. However, it would make the source code and libsc less objective since having the offset to E_Add in the example above enables the interpretation of that whole branch without any additional information.

3.4 Imports

The use of offsets/pointers is only possible within the scope of a single file. Therefore to access a symbol defined in a different file an Imp (short for Import) object is used. It's an object similar to other named objects in its class, there are currently only Type

Imps and Node Imps. Such an object has the same Name as the object that it "imports" and is transparent in the sense that the compiler will act as if the imported object was referred.

A Node_Imp holds just a Name whereas a Type_Imp is much more complicated since it has to handle importing a struct/union or a function type. This is handled by holding Fields similar to those in a struct so if a field of an external struct is used its name is added into the Fields and the expression points at it.

3.5 Views

Views are stored last in a file because they are not needed by the compiler and can be easily not read. There are currently only two view objects V_NEW and V_Block. V_Block stores an array of offsets to global declarations and other view objects. An offset to the same object can occur any number of times, however, circular referencing between V_Block objects is forbidden. V_NEW represents an empty line, which can be changed into other View objects or a global declaration. Since it isn't meant to store any meaningful data, and can only exist within other View objects, it is symbolized by offset 0, and is not stored anywhere in the file.

3.6 Compile-time conditional code

In practically all languages there is a possibility to create code that depends on certain compile time parameters. In C it's accomplished in most cases using preprocessing directives:

```
struct _tSCM_N {
        u16 Code;
        u08* pName;
        #if dSCM_N_Priv != 0
        void* Priv;
        #endif
}
#if dSCM_FnRR != 0
ui SCM_S_Any_FnRR (tSC* psc, tSCM_S_Any* psany, tSCM_Any* pfrom, tSCM_Any* pto);
ui SCM_E_Any_FnRR (tSC* psc, tSCM_E_Any* peany, tSCM_Any* pfrom, tSCM_Any* pto);
ui SCM_Any_FnRR (tSC* psc, tSCM_Any* pany, tSCM_Any* pfrom, tSCM_Any* pto);
#endif
```

These present a few obstacles in SC. First of all there will be offsets throughout the code that will “point” at these objects and they have to point at the condition along with the object. This is solvable using a new “Conditional” object, which will hold the possible forms of the object and the conditions for them. However, an object of a Type class has a different structure than an object of a Node class, which forces the existence of as many Cond objects as there are classes. This also means that in SC conditional forms are object oriented and in the example above each function would have a separate conditional assigned to it. This subject will be further analyzed in Chapter 5.6 Views.

4 The memory format and SC library

As has been previously stated, the memory format is based on static sized structures and dynamic memory management, therefore in general the code is straightforward and self-explanatory. Same as in SCF every structure that is part of the language has a code field that identifies what language construct it is as well as the type/format of the structure. This code is also used for dynamic dispatching of common methods either through switch-case statements, or function pointer arrays.

SCM is currently a semi template, it can be configured with macros to for example include a Priv field in some structures. The template functionality will probably be further extended as it gives more possibilities to save memory and increase efficiency. For example some apps may require a pointer to a parent object to achieve acceptable performance, while other apps may benefit from less memory usage and better cache utilization.

Because most allocations are very small, between 4B-40B on 32bit systems and 4B-80B on 64bit, the use of standard dynamic memory management can be very inefficient. Depending on the file the average allocation size (counting only <400B requests) is around 18B-21B on 32bits and 29B-38B on 64bits. This means that just storing the size (which is 4B or 8B [7]) of the requested block we get about 20% more memory consumed. This is excluding any additional overhead caused by alignment, paging and fragmentation. The overhead can be lowered by allocating larger blocks only for certain sizes of requests with bitmaps of which slots in the block are free/used. Which should (assuming low fragmentation and no alignment) give an overhead of around a bit per allocation. Unfortunately such a solution requires more work when deallocating, either on the part of the app by requiring the size of the deallocation, or the library requiring more block range checking. Considering the memory present in today's computers and the fact that SCM is not yet aimed for storing huge amounts of code there is no need for such optimizations at this moment and the standard libc malloc and free is used.

As has been mentioned in Chapter 3 The file format SCM proved very useful for translating between different versions of SC files. Due to correct naming it is possible to link against two different versions of libsc by performing find & replace on the source code from “SC” to for example “SC0”. This will be replaced by proper namespace support using macros in the future. Therefore the translation between two SCF representations boils down to translating between two SCM representations, which is incomparably simpler. The exact method used to accomplish this is strongly case dependent.

For example the addition of a new language construct starts with the addition of new Codes for it, and building the translator from these two libraries (before Code changes and after). This part does not require any additional code to be written as neither the structure of SCF or SCM was modified. The programmer usually then proceeds to write the SCM and SCEdit portion of the construct and tests it within SCEdit to ensure no mistakes were made, the SCEdit code is usually a very simple “just so it works” implementation. Following this are the OSCalc and M2F subsystem functions necessary to transform the new object from SCM to SCF, during this phase the programmer usually ascertains the correctness of the translation by manually analyzing an output file with a hex editor. Then the new F2M functions are written and tested on the output files with the new construct within them. Lastly assuming no serious design problems surfaced the SCEdit functions are finalized and fine-tuned. This process might appear complicated at first but in practice it is straightforward and can take anywhere between 15 minutes and a few days. The most complicated part is a good editing mechanism for SCEdit, which can be implemented

parallel to any other work e.g. on the compiler.

A slightly modified subset of the above process is used for SCF only changes. Up to now the most profound of these was the introduction of radix trees for symbol tables, which practically meant a complete change of the ordering and format of data in files. The process can be carried out either based on SCEdit (like in the previous example) or on the translator. Using SCEdit has the disadvantage that it is necessary to not break F2M functionality, whereas using the translator does not require a working F2M subsystem in the target libsc. Either way after the needed types exist OSCalc and M2F functions are modified and tested by analyzing output with a hex editor. Next the new F2M functions are written and tested this time in SCEdit as it enables easy checking whether the SCM trees are correct. After everything was proven correct the translator is build using the old (before the process was started) and new fully functional libsc. If there were no changes to SCM (which should be the case) then again no additional code is necessary.

Finally there are also language structure changes that require a modification of not only SCF, SCM and SCEdit, but also the modification of the translated source code. There was only one such change performed up to now, which is the move of local variable declarations from a statement list (C style) into an array in a block (almost Pascal style). This was performed in almost exactly the same manner as the introduction of a new construct as it was possible to maintain both types of local variables at the same time. The main difference was in the translator, which upon encountering an “old style” local variable had to add it into an array in the parent block and put the initializing expression in place of the declaration.

While these cases to some extent prove it is possible to relatively easily perform complex changes to the language and format, they are not representative of real world development. The first issue is that libsc currently is tied to a single version of the language and format, this means that a newer editor/compiler cannot work on older source code without it being first translated, and translating code to a newer version renders it incompatible with older systems. Introducing flexibility in this regard can be quite complex and it forces the maintenance of much more code than is directly needed making the language “bloated”. A development model with periods of compatibility separated by incompatible cleanup releases (with the automatic translator) is probably the most reasonable compromise. Another issue is the parallelization of development, all of the mentioned changes couldn't easily be performed by more than two people, and performing two separate changes at the same time can introduce many issues when merging them.

Summarizing project management and introduction into real world applications will probably be a substantial challenge for SC, and may seriously hinder development efficiency. Furthermore the scalability of development with increased manpower may also be low.

5 SCEdit - The SC file editor

SCEdit is without doubt the most important part of SC as the usefulness of this language completely depends on how efficiently it can be edited. Although it is not a trivial task to accomplish, the lack of many limitations that result from the format and language should enable unparalleled capabilities.

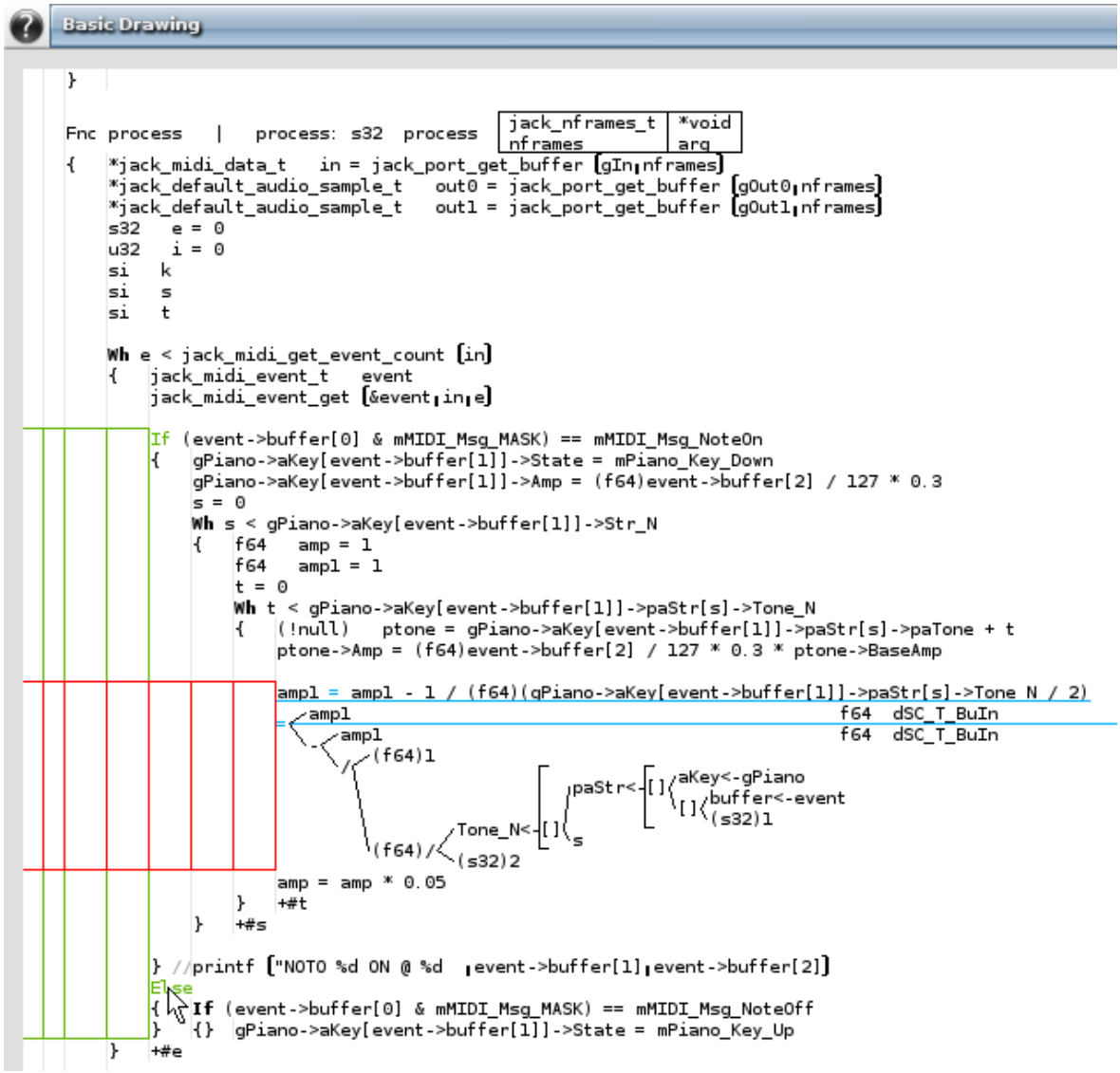
In a standard programming language a programmer creates a linear string of characters, for which the writing techniques are well established. There are also more complex techniques like those used in Vim, users of which boast they can write/edit faster. This linear string of characters is then interpreted by the compiler using certain grammar rules into a syntax tree, which is translated into an intermediate representation and then into output code. A good programmer often knows what general output code he needs, and has to devise a syntax tree that would most likely produce the needed code, and then project it into a linear string of characters. This linear description of a syntax tree can cause many problems, mostly to beginners but they can sometimes needlessly slow down advanced programmers:

- implicit casts are not always obvious
- operator priority, *pint++ is *(pint++) and not (*pint)++
- complex expressions and parentheses counting
- not always obvious which object __attribute__(()) corresponds to and what the behavior is

There are also source code styling issues:

- aligning of similar lines (when using tabs forces a certain tabwidth)
- managing proper and consistent indenting

Looking at Screenshot 5.1 the first immediately noticeable characteristic is the unconventional red and green frames on the left side. These are “markers” that highlight the currently selected object (red) and the currently “hovered over with the cursor” object (green). Light blue color marks the “keyboard hover”, which shows which object or field will be modified upon a key press. The exact way keyboard hover is marked is different for different objects but it is always the same color. One of the most important benefits of these markers is that it's easy to tell which object will be selected upon a mouse button click, which greatly helps when working with expressions in tree form.



Screenshot 5.1: SCEdit example code

Another important characteristic of the code is that it is non-linear, which is visible at the end of blocks. In standard C the printf call above the mouse cursor (if it was uncommented) would be after the block expression whereas in SC it is still part of it. The general rule is that anything that is on the right is a child of or is referenced by something that is on the left of it. Or to be more specific given an object A, any object that is completely within the vertical size of the marker of object A is referenced by it or is its child or a child of its child (any level of depth).

The lack of color syntax highlighting is in part due to the target being graphical and spatial distinction. The other reason is that semantic highlighting is far more important and it hasn't been completely thought out yet.

Moving onto the general concept of movement and editing. After initial analysis and experimentation it has been concluded that a more object oriented multimode mechanism should be more efficient than inline style editing. What is meant by inline editing is a mechanism more equivalent to standard text editors where most movement is per character and any keypress inserts a character at the current position. Object oriented means that for example a symbol reference is treated as an atomic object and in the “root” mode one can only move to or select the whole reference. Then after pressing “enter” SCEdit enters a “symbol reference edit” keyboard grabbing mode, which allows the programmer to change

the name of the symbol in a textbox-like element and presents completion possibilities along with any additional information that can be useful. When the user hits enter and the current symbol name is equivalent to a symbol that fits the given "slot" that symbol will become referenced by it and the editor returns to "root" mode. Currently there is only a "root" mode along with keyboard grabbing modes for all the atomic values like a name, symbol reference, number, field reference. Therefore it is only possible to change from the root mode to a keyboard grab mode and vice versa. However, as the language becomes more complicated it will be necessary to implement additional modes that will enable for example targeting of objects, which will require more advanced stacking of mode changes.

5.1 Internal architecture

When SCEdit opens a file for editing it is translated into SCM and all the files included by it are located and read into buffers (meaning they are kept in SCF form). The first idea was for SCEdit to work directly on SCM for the edited file, unfortunately that proved hard to implement mostly due to the possibility to view the same object a few times in the same canvas. Therefore the realization mechanism has been introduced. It allocates structures tSCE_Real that conceptually mean "That SCM object should be printed at X,Y takes W,H space (including its children), is a child of this tSCE_Real object and has these children".

```
struct _tSCE_Real
{
        tSCM_Any* pAny;
        tSCE_Real *pPar, *pCh, *pP, *pN;
        si X, Y, W, H;
        ui bFold:1;
};
```

The 8 main mechanisms that form the basis of printing and editing files are:

1. Realize – creates tSCE_Real structures based on SCM.
2. CalcXYWH – calculates the layout positions and sizes of objects.
3. Paint – draws the object on the screen.
4. Select – called when changing selection, to handle initializing of temporary private data.
5. KB_P – called upon a keyboard key press
6. M_Move – called upon a mouse movement
7. M_Click – called upon a mouse button click
8. Lin – used for changing selection upon certain key presses

SCEdit has a certain "editing state" data like a pointer to the currently selected object, its temporary private data and more. Only a tSCE_Real object can be selected, any internal selection of fields is handled by the object and is stored in its temporary private data.

CalcXYWH is one of the more complicated mechanisms and could prove troublesome in the future. It currently calculates the layout for the whole file, so unlike other mechanisms the calculation time increases substantially with the amount of code. Also as more advanced alignment features are added, the complexity of certain calculations will increase. Which may force certain changes into SCEdit's architecture like limiting recalculation to visible objects.

KB_P passes a key press to the currently selected object, which can ignore the event or react to it. If an object ignores the event it gets passed to its parent object, and so on, until one object accepts the event or the top level object is reached.

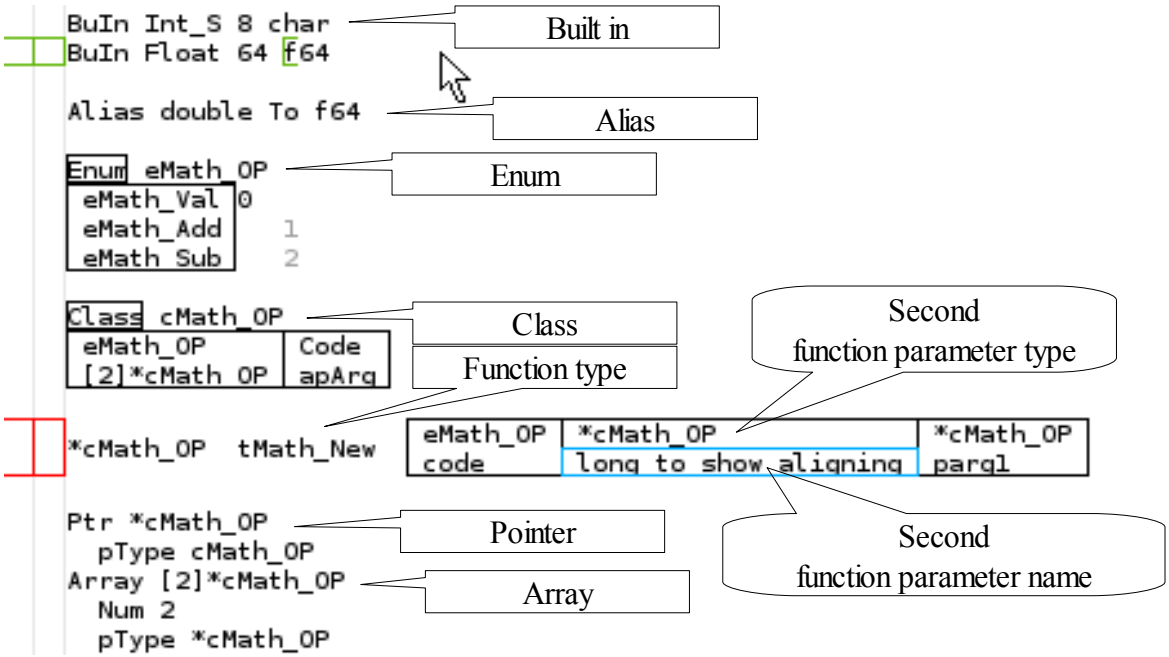
M_Move's main goal is to establish which tSCE_Real object is “hovered over” and also to set any temporary private data about which field is “hovered over”. The methods are issued recursively by each object if the given object calculates that the mouse points at one of its children but not itself.

M_Click first checks whether the currently “hovered over” object is the same as the selected one, if not it selects the “hovered over” object by issuing Select with a M_Click event. If the “hovered over” and selected objects are the same, then M_Click is issued to the selected object, which usually reacts by changing internal “keyboard hover” or entering an atomic value editing mode.

Lin handles inter tSCE_Real object movement, it currently has 2 modes: per object and per field/line. Per field/line is issued with an arrow key, but only if no object accepted the event. So for example in a class an arrow up will move through the fields (the object handles everything on its own), when it reaches the top of the class the object ignores the event and if all of its parents ignore it, the Lin mechanism is issued with a One_Field_Up event, which returns the object that will be selected, then a Select with a DeSelect event is called to the old object, and a Select with a One_Field_Up is issued to the new object, which (if applicable) will set its temporary private data so that the bottom most field/line is selected. Per object movement immediately moves between objects (entering into the right most one), and the internal field marker gets set to a default position. This event is issued with shift+arrow keys, it is never passed using KB_P but gets immediately issued as Lin. There will most probably be another mode “same level object”, which will not enter into Block, If, and other statements but just select them and then move past them.

5.2 Types

Types are relatively straightforward to print and edit as shown in Screenshot 5.2.1. All movements between atomic values are orthogonal and therefore map intuitively into arrow keys.



Screenshot 5.2.1: Types

The enum type aligns the explicit initializers, which are standard expressions and are modified using the same mechanism as expressions in function bodies. The gray numbers 1 and 2 are calculated values of these enum nodes. In the final implementation these will be printed as another column for even those nodes that have explicit initializers because they could contain macros and it might not be obvious what the number is.

The class type by default displays types on the left and names on the right. This behavior can easily be changed to display names on the left and types on the right, or to align text in the left column to the right.

Next is the function type, which is a standard named type like any other in SC. It displays the return type, its name, and the parameters it takes. A parameter takes two lines, the top is the type and the bottom is the name of the parameter. This follows the ideology of spatial differentiation and alignment. Likewise the borders in enum, class and function types follow the ideology of graphical highlighting.

There are two main drawbacks to the structure and presentation of enum and class types. For example in C it is common to write code like:

```
enum {
        dSC_E_START,
                dSC_E_NEW,
                dSC_E_Sym,

                dSC_E_Arg1_START,
                        dSC_E_Ptr,
                        dSC_E_DePtr,
                dSC_E_Arg1_END,

                dSC_E_V_IL,
        dSC_E_END,
};
struct _tSCM_E_Cst_BuIn
{
        tSCM_E E;
        u08 Type, Size;
        union {
                u08 V_u08;
                u16 V_u16;
                u32 V_u32;
                u64 V_u64;
        };
};
```

The problematic elements are empty lines, custom indentation, and multiple fields in the same line. All of these can be solved (on the data format side) by for example exploiting that pointers and offsets will never naturally have certain values. Although this solution is not the cleanest its implementation should be relatively simple and it will not introduce additional Codes or variables to objects, which would waste space even when not used.

The anonymous union type in _tSCM_E_Cst_BuIn is at this moment impossible to do in SC. However, that is mostly due to time constraints as it doesn't require much work on the part of data structures and storage. On the side of SCEdit it requires a bit more work to print everything readably and to give an efficient and as intuitive as possible editing mechanism.

The last two types are classified as “derived”, which means that their name is is directly derived from their structure. These should be handled automatically but again due to time constraints they aren't completely automatic.

5.3 Statements

Statements in SC are objects that do not return a type like a Block, If-Else, While, Switch, Break, Return. They most often hold a Block and a few expressions and cannot exist on their own (globally).

It is important to note that for example a block that is the body of a while loop is its child and is stored within that “while loop” object and not within the block that contains the while. This has many implications related to editing, the first is that deleting or commenting the while statement deletes or comments the while statement with its body. This has a few disadvantages as it's impossible to do something along these lines:

```
while (expression_A)
//while (expression_B)
{
        statements_A;
}
```

Which allows quick and easy switching between the while expressions or even omitting it to leave just the statements. Alternative while statements are still possible in SC with either (exprA && 1) || (exprB && 0) or the (not yet implemented) ignore expression. The main advantage (in the context of SCEdit) is that it eliminates some possible syntax errors and misinterpretations due to wrong indentation like:

```
if (exprA)
        if (exprB)
                stmtA;
else
        stmtB;
while (a_very_long_expression_A); //<- note the semicolon
        stmtsA;
```

The standard Block, which is a collection of statements, expressions and local variables, is the most important and complex statement in SC. It is printed as a pair of braces “{ }”, it indents its child objects, and can be folded. In SC Blocks are mandatory for If-Else, While, and other statements because of easier management.

```
Wh k < mPiano_Key_N
{   If gPiano->aKey[k]->State
    {   s = 0
        Wh s < gPiano->aKey[k]->Str_N
        {   t = 0
            Wh t < gPiano->aKey[k]->paStr[s]->Tone_N
            {   (!null)   ptone = gPiano->aKey[k]->paStr[s]->paTone + t

                *out0 = *out0 + (jack_default_audio_sample_t)(ptone->Amp * sin (ptone->Phase)
                ptone->Phase = ptone->Phase + m2PI / gRate * ptone->Freq
                If ptone->Phase > m2PI
                {}  ptone->Phase = ptone->Phase - m2PI
            }   +#t
    }   }   +#s
}   +#k

If *out0 >= 1
{}  *out0 = 0.999000013
If *out0 <= -1
{}  *out0 = -0.999000013
*out1 = *out0
```

Screenshot 5.3.1: Nested constructs with blocks

As has been previously explained the statements that are within the block aren't those that are between “{“ and “}” but those that are to the right of the vertical space from “{“ to “}”. The reasons behind this are: the rule is consistent with a one line block, less wasted lines, it doesn't exclude the possibility to still add the extra line if a programmer finds it

confusing (could be an editor dependent option). A “one line block” holds a single statement, which takes only a single line. In C the block is usually skipped and just the statement is written:

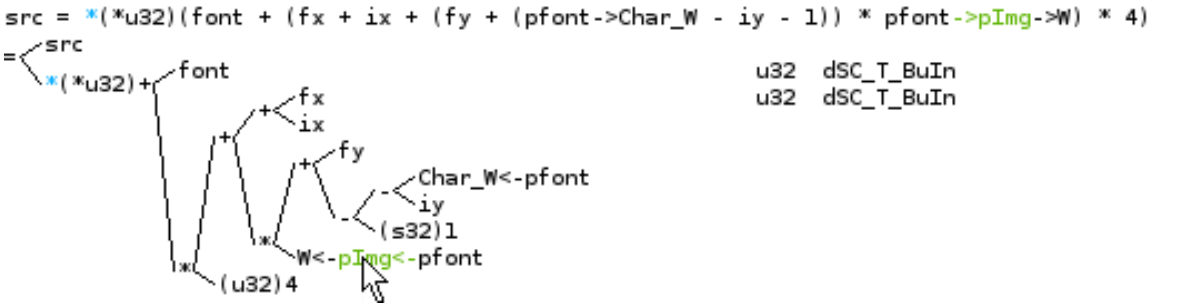
```
if (exprA)
        stmtA;
stmtsB;
```

The benefits are less typing required and more vertical compression, which often increases readability (or decreases depending on the given situation). The downside is that once the programmer wants to add another statement to the else condition he has to add those missing “{ }”, which forces the “waste” of a line for the ending “}”. In SCEdit due to the flexibility of Blocks it is possible to keep all the benefits without the drawbacks, as shown in Screenshot 5.3.1.

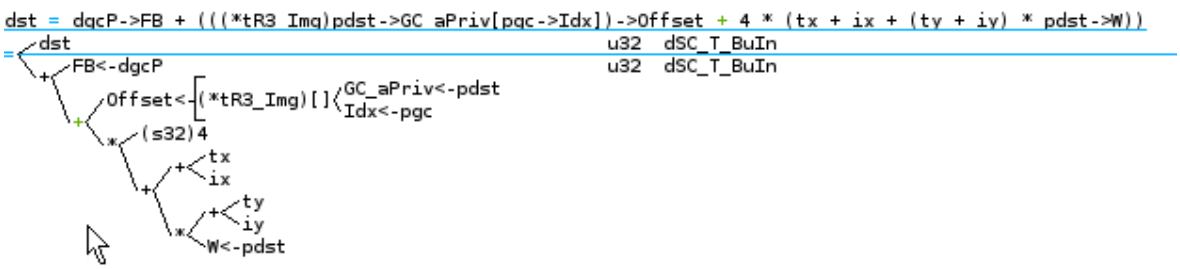
5.4 Expressions

In SCM every operator or L/R-value that is a part of an expression is a separate object. If every one of these objects was realized separately the memory consumption would increase by a few orders of magnitude. For this reason an expression is realized as a whole (one tSCE_Real object), which has the additional advantage of simplifying: memory management and custom presentation and editing techniques.

There are currently only 2 presentation methods: infix and tree, both shown in Screenshots 5.4.1 and 5.4.2. The infix form is read only, although it is possible to implement an editing scheme for it (requires simplified syntactic analysis), which should still provide benefits to readability and syntactic/semantic assistance over standard text. The tree form is, however, far more interesting as it solves the problems of operator priority and parentheses counting since both are implicit in the tree structure. Unfortunately it's used only for the selected expression due to much larger vertical footprint. Another method could be a mathematical view. However, editing it would be even less intuitive than trees so it would also be view only.

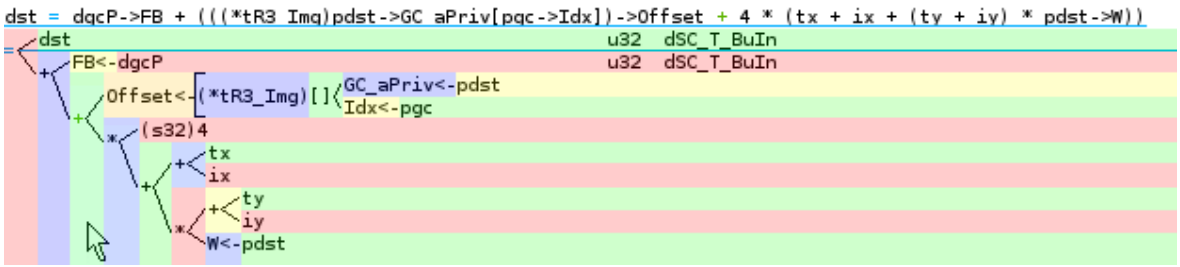


Screenshot 5.4.1: Example complex expression 1



Screenshot 5.4.2: Example complex expression 2

Modifying trees is a bit more complex than modifying linear statements because they are two dimensional, the movement directions are not orthogonal and they are always kept in syntactically correct state. First lets analyze the mouse hover regions that are marked with colored squares in Screenshot 5.4.3.



Screenshot 5.4.3: Hover regions

Each region corresponds to a single object, when the mouse enters that region the object gets mouse hover, and when the user clicks, that object gets keyboard hover. These regions are fairly intuitive for the user and the implementation is very straightforward.

Movement with the keyboard is much less intuitive. There are a few general movement concepts like hierarchical, line based and canvas position based. It is possible to implement many methods and enable switching between them but at this stage it's impractical (for maintenance reasons) and after a bit of testing a line based algorithm has been chosen as the default.

It works on the principal that most expression objects will either be a “line” (e.g. variable reference) or “between lines” (e.g. a two argument operation). Therefore it's possible to simply move with up and down arrow keys between these objects. The left arrow always moves to the parent expression, which is always directly to the left. Single argument operations are ignored by vertical movement and are accessed by moving to a child of it and pressing the left arrow. The right arrow works only for single argument operations and moves to the child, which is always directly to the right.

The most major inconsistency arises from a structure field reference that operates on a multi-line expression. To be consistent it would have to be treated as a single argument operation, however, in practice it was oftentimes more useful if it was treated as a “line”, which means that up/down arrows would move to it instead of its child.

Modifications of trees are based on a few actions:

- space key – inserts a NEW operation which will have the current object as its child,
- backspace key – deletes the parent object and moves the current object in its place,
- delete key – deletes the whole branch inserting a NEW operation in its place,
- tab key – in case of a 2 argument operation switches places of the two children,
- other keys – change the current operation (+,-,*,/ etc.), does not change the structure of the tree.

These techniques have been developed mostly for modifying existing expressions and they aren't very convenient for creating new expressions as they require much keyboard movement. Therefore another mechanism will be developed that will in many ways overlap with how expressions are written in text languages. The general concept is based on linearization, which means that any input is inserted almost as if it was written at the end of a standard infix presentation. However, as the specifics are not designed yet it is not presented in this thesis.

In a text based language when a programmer refers to a named object he simply writes its name and the compiler will evaluate which object to use. SC uses a pointer to the object, which can be compared to saying "use THAT object" instead of "use object named ABC". In practice this means that having a function:

```
int div_int (int a, int b) {
        return a / b;
}
```

If we were to switch the names of parameters a and b in a text based language we get:

```
int div_int (int b, int a) {
        return a / b;
}
```

Whereas in SC we get:

```
int div_int (int b, int a) {
        return b / a;
}
```

Therefore it can be said that SC is far more objective and direct in its description of the language, storing objects and the relations between them instead of storing declarations and requests.

5.5 Imports

Imports have quite a few benefits even though they are a necessity. The first one is a clear view of dependencies between files. The second is the benefit that a name change has to be done only on a per file basis. Also in case of class/struct/union type imports the storing of only used fields gives information on which fields of an external structure a given file depends. Therefore imports can be very useful when manually analyzing compatibility, however, it should also be possible to build scripts that will find every place in the source code where something was used that is now obsolete and point the programmer to it or (if possible) fix it automatically. This could also greatly help with correctly gauging whether breaking compatibility in certain places is “worth it” or not.

Another part of imports is how they are handled by the editor. First of all SCEdit has to properly provide any semantic feedback as in some cases it is necessary for correct editing. Also preferably they should be as automatic and transparent as possible, meaning the programmer shouldn't need to specifically create imports and add fields to them, as they should be added when the programmer uses them. To prevent false dependencies the editor should also delete any unused imports. All of this is already implemented to some extent.

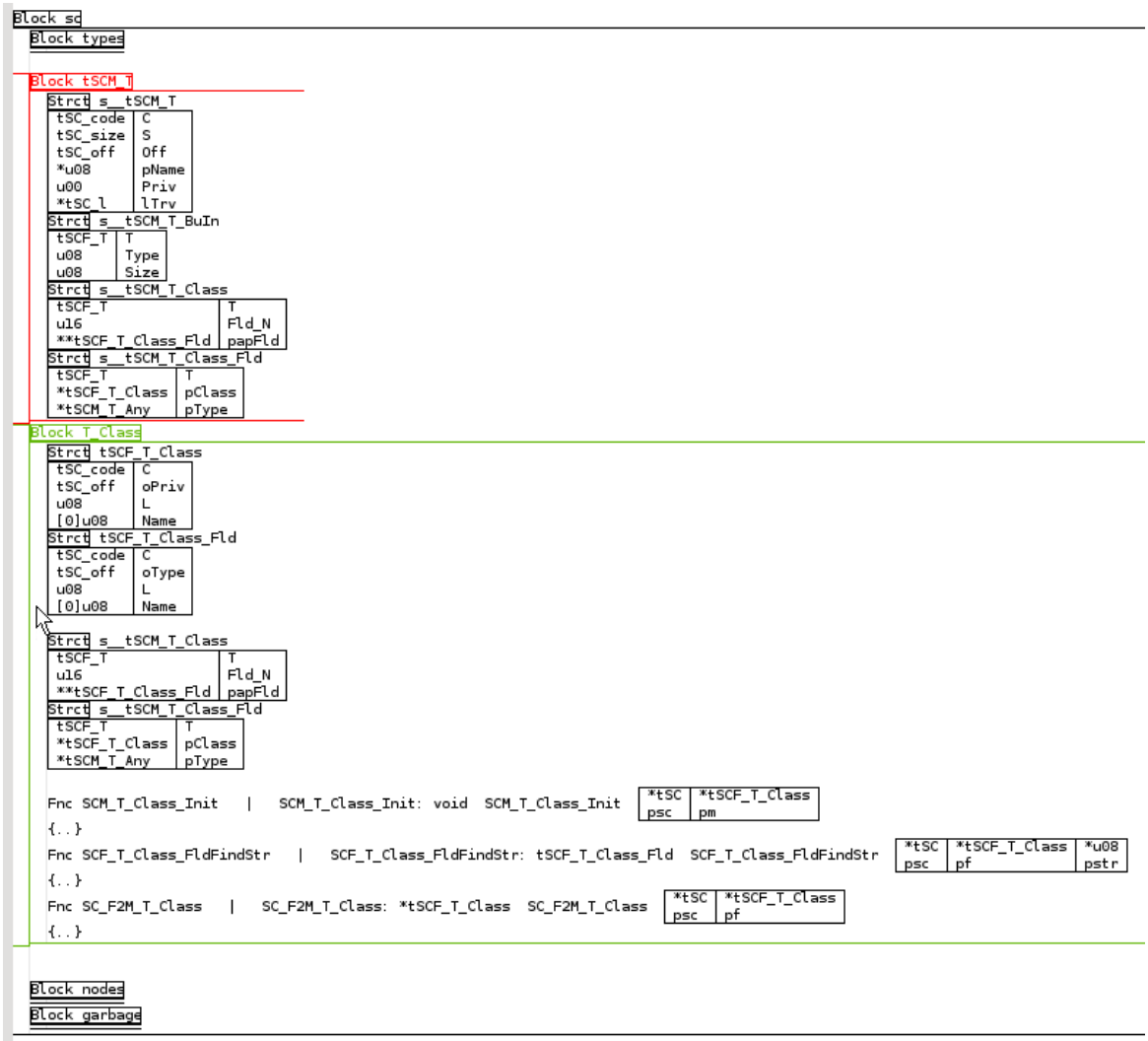
5.6 Views

Views exist mostly for use in SCEdit as a means to help the programmer manage source code. Conceptually they can be considered a projection of the contents of a file, or a perspective from which the file can be viewed.

The first most important characteristic of these is that they can present the same data in a different manner at the same time. This allows the programmer to quickly switch between views depending on what is needed and also temporarily create new ones to simplify some modifications.

The most basic view object is V_Block. It gives almost the same layout as standard text source code, which is a list of global declarations (or other View objects) printed

vertically one after another as shown in Screenshot 5.6.1. It can be named, folded, and it also indents its contents. It is similar to a pair "//BEGIN //END" in C/C++ which many editors recognize and mark as foldable.



Screenshot 5.6.1: Example use of V_Blocks using a part of libsc code

The same object can exist any number of times within a V_Block. Which allows for example to place a struct type at the top with all other types and again with functions that operate on it. This is the simplest manifestation of "different presentations at the same time", the "difference" being the grouping attribute, and "at the same" meaning that given large enough screen one could see both groups. This is shown (in a shortened manner) in Screenshot 5.6.1 where "Block tSCM_T" holds structs that are classified as "SCM_T", and "Block T_Class" holds structs and functions that deal with the conceptual object "T_Class". Blocks can also be "filled" with declarations using the name of the block as a pattern, which was used to create the example Screenshot 5.6.1.

In Chapter 3.6 the problem of objectiveness of compile time conditional code was mentioned. In real C headers there are often hundreds of objects within a single condition. The duplication of the expression itself can be mitigated by using a separate macro, however, there still remains the fact that when using V_Block it's impossible to achieve a presentation similar to what is in C headers (meaning grouping by condition and not object). It should be possible to create a new V_Cond object that would itself hold an expression and present all the Conditional object forms that depend on the same expression. There are however quite a few complications and it is hard to predict how

much limitations this will force on both the structure of the code and its maintenance.

Another benefit of views is that they allow virtual splitting into multiple “files”. Many projects are often split even though the code itself is not modular, and the split forces writing of additional headers and duplicating information. The reasons behind such actions are easier code editing/management and more granular recompilation. The first reason is of focus here as it is directly solved by views. In a standard text editor quickly moving around a file containing just 30k lines of code is troublesome due to the scroll bar loosing precision, and the inability to quickly switch between certain groups of functions (though this can be helped with bookmarks). In SCEdit any View object can be selected as the “root” to which the canvas gets limited to. Therefore having a file with a “top most” V_Block in which there are many mid-sized V_Blocks, even if the total size exceeds usable limits the canvas can easily be limited to one of the mid-sized V_Blocks. Furthermore the editor could easily create separate tabs from the most useful V_Blocks giving the programmer a means to quickly switch between them as if they were different files.

Currently there are no other view objects, but some concepts are presented in the following paragraphs. First of all it is possible to embed any content in an SC file like graphs, html pages, pictures, sounds, movies. The downside would be needlessly larger files, however, because views are stored last in a file, the compiler can simply not read them. Alternatively it's possible to store content separately and just include it.

Graphs are the most interesting, especially if they could be automatically generated from the contents of the file like for example a call graph or a class/structure reference graph. In such a case they would take very little space within the file (just the information on what to generate) and they could be quickly auto-updated.

There is also space for variants of V_Block. Like a dual column view (similar to V_Block), which could prove useful for comparing or more efficient use of horizontal space. Although use of it would probably be very rare, as it is “strange”, would require additional shortcuts for movement between columns and could cause some view space issues which SC aims to solve.

All of these views still fit within the concept of a canvas, where each element takes a bit of space and none overlap. SCEdit could introduce another concept of “floats”. These would be views that float above the main canvas either trying to stay in view (within certain boundaries), or are pinned to window coordinates. The most common use case would probably be to hold a few structures or classes while editing or analyzing function bodies, or doing a quick lookup of a type or function. The reasons why such floating views should be sometimes more convenient than switching between views or files are:

- most lines in source code tend to not take more than half of the view space, giving ample space on the right side to fit a bit of code without occluding the main contents,
- a single eye movement is enough to put the fixation point on the other information whereas changing views or files creates much more distractions,
- a quick lookup (assuming one knows the name of the object) doesn't require searching for the information (file/view change, scrolling).

6 Summary

The current implementation of SCEdit takes 12k lines[2] of code, libsc is 7k, the main part of the gcc front-end is 1,1k, and the main part of the clang AST consumer is 1,2k. Considering some of the present features, and the fact that many more can be implemented with very little effort, it is a very small amount of code. For comparison katepart-4.3.4[3] which is a standard text editor for KDE[4] with history, find & replace, code highlighting, folding (with a serious bug[5]), limited auto indenting, limited code completion, plugins and more is around 50k lines of code. On the other hand it is true that the implementation of for example history in SCEdit may consume a lot of work and there are also many other peculiarities that can dramatically increase the size of SCEdit and libsc.

SCEdit was able to efficiently perform code completion with 468k nodes total spread between 10 header files that were loaded 2^8 times, which is equivalent to around 39MB of SC source code. The current implementation of code completion doesn't perform any indexing or memory allocations, it simply traverses the SCF symbol trees, assembling names in a small buffer on the fly and ignoring any branches that do not fit the text written by the programmer. This also means that even in case when those 10 headers were loaded 2^{10} times (about 155MB), and there were 1874k nodes total, when the programmer typed “s”, which limited the amount of nodes to 175k, the performance was again acceptable. It is important to note that these nodes appeared sorted only on a per file basis (which in some cases could be desirable), because code completion doesn't perform any sorting and symbol tables in files are already presorted. One shouldn't put too much into these numbers as the current implementation is very wasteful, the headers do not contain any objects that are within an equivalent of “#ifdef #endif” and no human could search visually through a list of even 40k names. The most important conclusion here is that SCEdit can efficiently filter out a million of nodes, and has access to all information regarding the nodes of interest without any indexing, and memory consumption roughly equal to the amount of code opened for assistance purposes.

Comparing compilation time or memory consumption is unfortunately pointless due to there not being enough similar code, and the gcc front-end being in an extremely experimental stage with a lot of debugging code. However, it does produce working code for a x86 32bit and 64bit target that in many cases is the same as equivalent C code.

Using all the mentioned tools it was possible to write (halfway through the project) a working piano synthesizer which used libc, libm, libjack and libncurses. The synthesizer code also proved a good initial testing ground for many aspects of the editor. This code survived many serious format changes like those mentioned in Chapter 4. Proving that it is possible to at least introduce such changes to the format and still provide an automatic translator.

Preprocessing and compile time metaprogramming

The lack of lexical preprocessing and the impossibility of implementing it in a standard fashion is the most major drawback. This forces the creation of substitutes that work within the structure of the language (syntactic preprocessing). These substitutes are often inferior in capabilities and are harder to implement, however, they are often much easier to understand, use, manage and debug. Ironically it has been a trend among newer

2 Counted using SLOCCount http://www.dwheeler.com/sloccount/
3 http://www.kate-editor.org/katepart
4 K Desktop Environment http://kde.org/
5 https://bugs.kde.org/show_bug.cgi?id=107988

languages to abandon lexical preprocessing as much as possible.

Syntactic preprocessing could theoretically be extended above what is present in other languages as SC doesn't have some of the limitations of other languages like too many keywords, unreadable syntax and designing unambiguous syntax. But has an advantage that the editor can handle management of certain aspects of the source code. For example it should be possible to implement function templating/derivation in a way that would present the programmer with the basic implementation (highlighted as gray), the user would modify it more or less like any other function, and SC would store just the differences. Something similar can be implemented in C by heavy use of macros unfortunately it's often more time consuming than simple copy & paste & modify and can be very hard to read, manage and debug.

The most interesting concept is to build a specialized SC code interpreter, which could not only allow very advanced syntactic preprocessing and compile time metaprogramming, but also code analysis like additional warnings about api misuse or version incompatibility. However, such a solution would probably prove very complex, especially on the part of SCEdit which would have to intelligently manage data to not stutter or consume to much memory. Also it could introduce many compatibility issues as such metaprogramming code would probably be dependent to some extent on the internal structure of SC code.

Assuming such a system is feasible, it could enable for example multilevel programming where the programmer creates a high-level design (e.g. block diagrams) based on which most lower-level data, functions and types are generated, and then proceed to designing certain lower-level parts (stored as “patches/differences”) that cannot be expressed in the high-level representation. While this isn't anything very new as many IDEs already have similar capabilities, it would be much more general and open in nature since it would be more of an infrastructure to build such systems than a very specific and narrow system. Also the whole process would be part of the source code and wouldn't depend on any external systems, making it possible to allow full insight into the whole process from SCEdit.

Taking into account the current implementation with it's many limitations and the theoretical possibilities, preprocessing and compile time metaprogramming will most probably be either the strongest or the weakest point of SC.

SC and LLVM

The SC language could also theoretically yield itself well to “just in time compilation” and certain “self modifying code” techniques. Although this would require extensive modifications to GCCs internal representation and back-ends, which admittedly is far more complicated then this whole project. However, there exists a project named LLVM (Low Level Virtual Machine), which could provide the needed capabilities, “LLVM is a low-level object code representation that uses simple RISC-like instructions, but provides rich, language-independent, type information and dataflow (SSA) information about operands. This combination enables sophisticated transformations on object code, while remaining light-weight enough to be attached to the executable. This combination is key to allowing link-time, run-time, and offline transformations”[6].

The combination of SC and LLVM should provide many benefits even without the run-time features as it would further diminish the barrier between the programmer and the compiler. For example SCEdit could present the output intermediate-like LLVM code. Or

6 http://llvm.org

give the possibility to inline LLVM code, which would have the benefit of greater portability over inline assembler code. All of this would, assuming the previously mentioned metaprogramming features were included, give an almost full very-high-level to low-level programming environment with complete insight into the whole process from SCEdit.

The conceptual and technical difficulties of such a system are obviously hard to imagine at this point, and these claims should not be taken too seriously as they are not based on any through analysis.

Vector operations

More on the presentation side there is the possibility to implement vector variables and operations graphically as blocks. This should greatly help beginners in this field as most pack/unpack, shuffle, interleave operations are hard to imagine and remember at first, while a graphical representation is immediately understandable. It should also be possible to build a general solution where a programmer builds his operations (based on the same blocks) and the compiler has to build code that accomplishes them. Obviously not all operations would be vectorizable, and the compiler might translate some into serial operations. However, that would still at least encourage more attempts at vectorization and organizing data in a parallel friendly way.

Whichever implementation would be chosen it would require a certain virtual model of a vector unit. There are unfortunately many differences between these units in different architectures that make it very hard to create a fully general yet efficient solution. For example the vector unit in Intel's Larrabee has a mask register and supports scatter/gather operations, which greatly simplify vectorization and can increase performance [8]. However, code relying on these features wouldn't be translatable into efficient code for a vector unit without them.

Probably the best solution would be to create the most advanced and general system and provide information about the target unit based on which the programmer can create different variants of an algorithm if necessary. However, it is important to remember that most of the programming work required to make such a system is again on the part of the compiler.

Views, recompilation granularity and inter-file optimizations

In Chapter 5.6 Views splitting source code into multiple files was mentioned and how SC enables “imaginary” splitting to maintain ease of programming. The other issue here is of recompilation granularity and inter-file optimizations. In most C production environments every source file is compiled separately into an object file, and then all the object files are linked together into the executable application. Whenever a programmer makes a change in the source code, only those files that were changed (and those depending on them in case of headers) are recompiled. This provides a huge benefit to productivity as the recompilation time is significantly reduced, however, it also limits optimizations as the compiler does not have access to information from other files. There is also the issue that compiling a single huge source file can result in very high memory consumption, which in turn can severely impact compilation performance. While SC currently works in the same manner, it should be possible to implement a per function compilation strategy, where a single input file is compiled into many object files each with a single or a few functions. Whenever a programmer modifies the source code the editor appends to a file which objects were changed so the compiler can recompile only those functions that are changed or influenced by the change. This would solve both problems of

compilation granularity, and inter-file optimizations, assuming the files were merged, or SC could compile multiple files together (should be possible). As for the memory consumption issue, as was mentioned in Chapter 2, an object can be translated into SCM "on demand" which should greatly diminish the problem. Assuming the GCC fronted achieved its goal of direct SCF to GENERIC or GIMPLE translation it should still be possible to perform it "on demand" in mostly the same way it is currently performed.

Target audience

In its current implementation the SC language is not yet usable for anything more than experimentation, mostly due to incomplete preprocessing capabilities and the unfinished state of SCEdit. However, assuming an equivalence in capabilities to C is reached along with at least semi automatic translation of headers, SC should prove most useful to these groups of programmers:

- beginners who want the freedom, straightforwardness and clarity offered by C, but find it too hard to deal with,
- C programmers who want more concise source code with a more advanced source code editing environment.

Much depends on the direction SC takes in terms of the language. For example the inclusion of certain higher-level abstractions should be possible without sacrificing clarity and straightforwardness, inferred typing being one that is already implemented. Therefore the inclusion of such features could be acceptable for a system programming language. Which would also add a third group:

- other language programmers who want lower-level control without loosing certain productivity benefits of higher-level languages.

Bibliography

[1] Wikipedia, Abstract_syntax_tree, http://en.wikipedia.org/wiki/Abstract_syntax_tree

[2] Wikipedia, Context-free grammar, http://en.wikipedia.org/wiki/Context_free_grammar

[3] Wikipedia, Programming language, http://en.wikipedia.org/wiki/Programming_language

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Pearson Education, 1986

[5] Wikipedia, Keyword (computer programming), http://en.wikipedia.org/wiki/Keyword_(computer_programming)

[6] Diego Novillo, GCC Internals, http://www.airs.com/dnovillo/200711-GCC-Internals/200711-GCC-Internals-1-condensed.pdf

[7] GNU C Library 2.11, malloc/malloc.c: line 105: Minimum overhead per allocated chunk, http://ftp.gnu.org/gnu/glibc/glibc-2.11.tar.bz2

[8] Michael Abrash, A First Look at the Larrabee New Instructions (LRBni), 2009, http://www.ddj.com/hpc-high-performance-computing/216402188?pgno=1