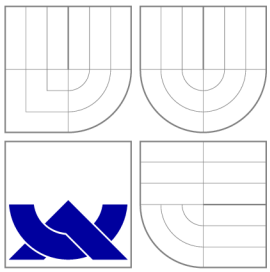
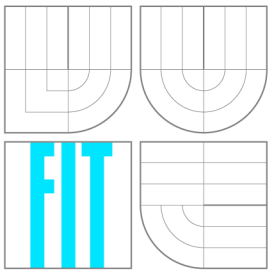


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

METHODOLOGY OF CONSTRUCTION COMPILER FRONT-END AND ITS INTEGRATION INTO THE GNU COMPILER COLLECTION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PETR MACHATA,

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MILOŠ EYSSELT, CSc.

BRNO 2007

Abstrakt

Vstupní bariéra pro vývoj uvnitř GCC se během posledních let znatelně snížila. Na konferencích, v časopisech a na webu se objevují články s architektonickými přehledy a návody. Věci se nadále zjednodušují použitím oficiálního vnitřního jazyka GENERIC: pro komunikaci mezi přední částí a zbytkem překladače již není nutné zabývat se obtížným a nepřehledným RTL.

Přesto je práce se souborem zdrojových kódů velikosti GCC nutně složitá. Je třeba napsat určité soubory a provést určitá nastavení, oboje jen s poměrně malým množstvím dokumentace.

Cílem této práce je pomoci s posledním zmíněným bodem. Práce popisuje ukázkovou přední část: vše od vytvoření zdrojových souborů, přes různé konstrukce jazyka GENERIC, až k problémům s kompilací běhové podpory nebo používání nativního preprocesoru.

Klíčová slova

GCC, GNU Compiler Collection, přední část, Algol 60, kompilátor

Abstract

The entry barrier to the development for GCC got considerably lower during the last years. Articles with various architectural overviews and how-to documents pop up in magazines, websites, and on conferences. With official intermediate language, GENERIC, used for communication between front end and the rest of the compiler, things are yet easier: it's no more necessary to bear the tedium of RTL when one writes new front end.

Yet there is a complexity inherent in handling a source base the size of GCC. There are files to be written, peculiar options to be set up, and this all with relatively thin documentation. This work is written to help with this last point. An example front end is described, with everything from the source base setup, through various GENERIC constructs, up to compilation of runtime library, or using GCC native preprocessor.

Keywords

GCC, GNU Compiler Collection, frontend, front end, Algol 60, compiler

Citace

Petr Machata: , diplomová práce, Brno, FIT VUT v Brně, 2007

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Miloš Eysselta, CSc., pod technickým dozorem pana Ing. Lukáš Szemly. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Machata
22nd May 2007

Poděkování

I owe thanks for patience and advices to the thesis' supervisor, Ing. Lukáš Szemla, who was bombarded by my status reports monthly; and Professor Jan van Katwijk of Delft University of Technology, for advices regarding odds and ends of ALGOL 60.

Contents

Chapter 1

Introduction

The grok of GCC is comprised of a very vast array of assorted information bits. There is no chance one person could achieve it in course of one academic year. In fact, software products with a complexity similar to GCC usually surround themselves with a shroud of mystery, and recognizing the behavior as designed from mere coincidence or a bug becomes a difficult task. Writing for GCC, as simple as it became, is still an adventure, in both senses of the word.

In this work, albeit inevitably incomplete given what was just written, I strive to give the implementer a sort of kick start on writing new language support for GCC. It is by no means a reference work, but it's useful "how to", that gives at least a glimpse of all what there is to writing GCC front end.

This work is only partially designed to be read sequentially. It's more of a collection of how-to themes.

The first two chapters (not including this) are basically sequential: some arguments for GCC are presented in chapter 2, and an overview of GCC architecture is provided in chapter 3.

The chapter 4 is a tour through a minimal GCC front end. Files are examined one at a time, and an explanation is provided on various constructs. This chapter should be read before all the other chapters.

The chapter 5 gives a glimpse of Algol 60 the language, and high-level organization of its compiler.

The chapter 6 describes GENERIC, a tree-based language that GCC uses to describe program structure. You will want to read selected parts of this chapter, as you will implement various features of your language. In a last section (6.5), emitting debugger information is described.

The chapter 7 collects descriptions of various features of GCC, that are not directly related to the translation of source language. These include use of preprocessor (7.1), processing command line options (7.2), emitting diagnostic messages (7.3), integration of language run time library (7.4), and integration of test suite (7.5).

Enjoy.

Chapter 2

Why GCC, Why Now

When facing a task of engineering a processor [1] of a given language, you have several options.

2.1 Language Processors Breakdown

You could write an interpreter, a tool that, given a program, emulates its actions token by token, possibly using some form of compilation into intermediate code. Interpreter has the advantage of being rather simple to write. And if the language still isn't sorted out completely, it will be simple to adjust the processor.

You could write a compiler that emits other high-level language, such as C. Compilation via C is quite popular, but it has drawbacks. E.g. programs in C will contain artifacts introduced during the compilation, and those will be visible in debugger. C may not support all the necessary constructs that your language needs, and you may have to use an even higher-level language. You will typically have to give artificial names to various thunks of code, and mangle program identifiers. On the other hand, C is well understood, with ubiquitous compilers, and tons of documentation.

Another option is producing virtual machine instructions, such as JVM or CLR. This can be advantageous, if you can count on having the target machine on binary host site. Virtual machines typically do just in time compilation, so you can experience near-native speed of programs. And they can be ported to several platforms, which means your compiler will be portable for free.

2.2 Processing With GCC

This thesis describes yet another option: writing the processor as a part of a well known compiler suite, GCC.

Crafting GCC front ends used to be hard. One had to understand quirks of RTL, GCC's intermediate language, which was neither easy, nor high-level. But things changed: GCC team took tree language used as an AST representation in C and C++ front ends, and generalized it into an official intermediate language called GENERIC. Work with GENERIC is rather easy [8], on par, I'd say, with emitting C. Unlike C, there are all GCC bells and whistles: attributes, inline assembly, OpenMP. So it's very powerful platform, compared to C.

[1] In a general sense of a tool that allows direct or indirect execution of a program in a given language.

Unlike C, however, the documentation is rather thin. Quite often I found myself scanning through other front ends in hunt for particular usage of some feature. Quite often GCC died on me with assertion error, and I had to look up what went wrong, and figure out why. This is actually easier than it sounds, but the fact is, GCC would benefit from better documentation.

GCC is work in progress. For some twenty years as of now. Things change. If you won't get your front end into GCC trunk, you will have to deal with those changes yourself. And as far as I know, new languages are not exactly the priority of GCC team, supposedly for exactly the reason that they would have to maintain them. It would have to be very high-impact language for that to happen. If you won't carry your compiler forward, it will become irrelevant as time passes. Just as are GCC-2.9x slowly drifting towards irrelevance today[2].

Given GCC's strong C heritage, it's still best fit for compilation of C-like languages. Currently the GCC family contains C, C++, their Objective variants, Java, FORTRAN, and Ada. Those are all rather C-like languages. I can imagine compiling something like Python through GCC, and Mercury, a declarative logic/functional language was implemented as GCC front end. But still the best match will be for imperative C-like languages.

On the other hand, GCC is a mature, even if underdocumented compiler. Big companies depend on GCC, on both the vendor and the consumer sides. It is ported to a huge number of platforms, and has the necessary drive and impact. Lots of people know how to use it, build it, package it, distribute it, and that means lots of people will know how to work with your front end from the day one. This is very important. Clever usage of GCC features will make your processor another option for wide range of tasks, from high-level parallel computations to systems programming.

2.3 What's Ahead

I have written an Algol 60 [1] front end for GCC [3] to get myself familiar with the platform. I was expecting days[4] spent in GCC's internals, digging in code knee deep. Much to my surprise, no such thing happened. I had my share of cursing, but overall I was pleasantly surprised. I doubt going via C would make the work significantly easier.

What's ahead is description of my experience with development of Algol 60 front end. It's a mix of things from GCC Internals documentation [3], things cut'n'pasted from other people's code and later analyzed, and things either found in GCC comments or tried by chance and found to work, all wrapped up and delivered as a continuous, and hopefully coherent text.

[2] I'll risk upsetting some people, and will claim that for purposes of mainstream use, GCC 2.9x already is irrelevant. I know of several developers that stick with it, e.g. for reasons of binary compatibility, hardware, OS or other dependencies, and typical corporation won't change the core compiler in a moment's thought. But these are niche uses from mainstream perspective.

[3] `http://projects.almad.net/gcc-algol`

[4] Or rather nights. With lots of tea.

Chapter 3

GCC Architecture

GCC can translate from variety of source languages into variety of assemblers. With one command, decorated with various flags, it's possible to do preprocessing, compilation, assembly and linking. How does it achieve this?

In following sections, we delve into successively deeper levels of overall GCC architecture.

3.1 Compilation Driver and Compiler Proper

On the outermost level, GCC is divided into a *compilation driver*, and a *compiler proper*. Besides this, GCC uses several host tool chain components, an assembler and a linker.

Compilation driver is a user-interfacing application. It knows about all languages that GCC supports. Given a source file, it can guess what to do based on file's suffix. Then it launches various other tools, most prominently compiler proper, but also preprocessor, assembler, and, eventually, linker.

Whole compilation works as a pipeline on an application level. After optional preprocessing, the source file is fed into compiler proper, that emits assembly. This is then assembled into object file. Finally, object file is handed over to linker to produce final executable. This process is managed by compilation driver. Depending on command line switches in effect it can be cut short in any of the stages: after preprocessing (`-E`), after compilation (`-S`), and after assembly (`-c`).

From our standpoint, the critical component is the compiler proper. There is one such component for each language, each in separated executable binary. You can run the binary yourself if you so wish, just as driver does. Let's look at that component closer in next section.

3.2 Front End, Middle End, and Back End

The compiler proper itself is composed from three components: a *front end*, a *back end*, and a *middle end*. While the front end is suited specifically to each language, the two other ends are the same in all various compiler propers. Front end is the part that encapsulates all the logic special to your language.

Just like the compilation process done by the driver, the compilation of a source file can be viewed as a pipeline that converts one program representation into another. Source code enters the front end and flows through the pipeline, being converted at each stage

into successively lower-level representation forms until final code generation in the form of assembly code [6].

There are two intermediate languages that are used on the interfaces between the three GCC's ends. The higher level one, used between front end and middle end, is called *GENERIC*. The lower level one, used between middle end and back end, is called RTL, or Register Transfer Language. Both middle end and back end do various optimizations on their intermediate representation before they turn it into yet lower level one.

Both interfaces mentioned are *unidirectional*: front end feeds GENERIC into middle end, middle end feeds RTL into back end. But sometimes the other direction is also necessary. For example, during alias analysis, middle end has to know whether two objects of different data types may occupy the same memory location [6]. Each language has its own rules for that, and front end is the place where language-dependent things happen. For this purpose, GCC has a mechanism of *language hooks* or *langhooks*, which provide a way to front end's participation in lower layers of compilation process.

The goal of front end is to analyze source program, and ensure all types are correct and all constraints required by the language definition hold. If everything is OK, it has to provide GENERIC representation of the program. For this reason, a bulk of this thesis deals with GENERIC. You don't have to know anything about RTL to write front end—at least I don't.

3.3 GENERIC and GIMPLE

The intermediate form that is important to us is called GENERIC. From expressiveness point of view, it's similar to C. From notation point of view, it's similar to Lisp. GENERIC is capable of representing whole functions, i.e. it supports everything there is to represent in a typical C function: variables, loops, conditionals, function calls, etc.

GENERIC is a tree language (hence the Lisp qualities). As any well behaving tree, it's recursive in nature, having both internal and leaf nodes, with internal nodes capable of holding other internal nodes. Typical leaves are identifier references, integer numbers, etc. Internal nodes are then unary or binary operations, block containers, etc.

For optimization purposes, GENERIC is still too high level a representation. During a course of compilation, it's lowered. The intermediate code that it's lowered into is called *GIMPLE*. The process of lowering is thus inevitably called *gimplification*. GIMPLE is a subset of GENERIC. Nesting structures are still represented as containers in GIMPLE, but all expressions are broken down to three address code, using temporaries to store intermediate results[5]. There are actually two GIMPLE forms: high GIMPLE and low GIMPLE. In low GIMPLE containers are further transformed into `goto`s and labels[6].

Apart from predefined nodes, GCC provides a mechanism to define your own GENERIC node types. Of course it wouldn't know how to gimplify them, and a langhook is necessary for this purpose. C and C++ front ends actually don't use GENERIC for their AST representation, but extend it with their own node types, and then provide means of gimplification.

3.4 Fronted ASTs

While it is possible to use GENERIC for representation of programs in your front end, it is recommended not to do so [6] [8]. Your own AST representation can suit the language

in hand better, and furthermore you are better shielded from the changes in GCC core. Besides, the language analysis tools that you write are then shielded from *GCC itself*, which makes them reusable in other tasks: e.g. as a syntax checker and pretty printer in smart editors.

This thesis aims very strongly towards the direction of your own AST. Under such a scenario, there are four layers actually: front end is further divided into two. The top one is a generic language processing library; the bottom one is actual GCC front end, and it translates one tree form (your AST) into other (GCC's GENERIC).

3.5 Source Base

Unfortunately, all GCC front ends have to be built inside the GCC tree. There is no way to use e.g. public headers and link against GCC libraries that would implement lower layers of compilation process. You will have to store your source files into places where GCC expects them and adhere to the overall GCC build policy. In following text, directories are referred to relatively to the directory where you unpacked a distribution tar file. E.g. gcc/ refers to actual directory gcc-4.2.0/.

The core of compiler is stored in directory gcc/gcc/. Directly in this directory are the files that compose a C Front End, general GCC services, whole middle end, and a machine independent parts of back end. Besides this, there is a number of directories with front ends. E.g. gcc/gcc/cp/ contains a C++ front end. One interesting front end is Treelang (gcc/gcc/treelang/), a toy language showing off how to write front ends. You will have to create one such directory, chapter 4 deals with this.

Another interesting directory is gcc/gcc/testsuite/, which contains automated DejaGNU test suite. You will most probably want to have your own files there. Section 7.5 is dedicated to subject of writing test suite.

In top level directory, a GCC runtime services are stored. This includes runtime libraries for various front ends, e.g. gcc/libstdc++-v3/ with C++ runtime library. Most probably you will want to write your own runtime library. Section 7.4 deals with this.

If you can afford depending on GCC, i.e. you don't want your front end to be completely independent, you can use data types and routines from a gcc/libiberty/ library.

3.6 Garbage Collector

Internally, GCC uses garbage collector [1] [3, chapter Memory Management and Type Information] for its memory management. The objects with indeterminable lifetime, which includes trees, are not managed explicitly, but instead garbage-collected. The collector used is of mark & sweep kind. Pointers (variables, fields, ...) that should be collected are explicitly tagged, the tags are gathered during the build, and marking and scanning routines are generated.

The garbage collector data are also used for implementation of precompiled headers. The precompiled header mechanism can only save static variables if they're scalar. Complex data structures must be allocated in garbage-collected memory to be saved in a precompiled header.

[1] `http://gcc.gnu.org/wiki/Memory_management`

3.7 Summary

In this chapter, I covered overall structure of GCC, as well as some internal details that will be useful later. Let's recap.

- The GCC as a whole is composed of a driver and a number of actual compilers, one for each language. Driver knows about all the compilers, and depending on its command line settings it preprocesses the file, hands it over to one of the compilers, and assembles and links its output.
- The compilers themselves have three layers each: a front end, where the language specific logic is located, platform independent middle end, and platform dependent back end.
- The language that is used to represent programs in middle end, and that front end emits, is called GENERIC. GENERIC is C-like tree language. Besides predefined node types, front end can register its own node types and use them at will. Subset of GENERIC, used for optimizations, is called GIMPLE, and the process of lowering GENERIC to GIMPLE a gimplification. You have to provide gimplifying routines for your front end's own tree nodes.
- While it is possible to use GENERIC as an AST representation, it is advisable to use it rather as a target language, and craft the AST to suit your front end's needs.
- GCC uses garbage collector. Garbage collected structures are tagged, and memory scanners are generated based on the tags during the build process.

Chapter 4

Hello, World!

In this section, we will create minimal GCC front end. The criteria are very simple: the front end has to be recognized and compiled by GCC, and when launched, it has to provide deterministic results; it must not fail. Note that we don't require the results to depend in any way on the source file being compiled. To cut the teeth, we will create a front end, whose only purpose is producing the code equivalent to the following C code:

```
int main (void) {
   return 7;
}
```

The return value of "7" is picked arbitrarily. I used to have "4" in its place, but found out that GCC uses 4 as a return value when it fails, and thus it's less clear when the GCC fails, and when it succeeds and the binary answers 4.

The prerequisite here is that you already know how to build GCC for your system. There is no chance experimenting with front ends if you never built GCC yourself.

Besides reading this chapter, you can find inspiring some other minimal front ends. I created a "Hello World" front end as part of work on gcc-algol [1]. Surprisingly enough, real "GCC Hello World" exists[2], as was brought to my attention later. Another possibility is to look at GCC's Treelang front end.

4.1 Agenda

The following will be covered in this chapter:

1. GCC source base set up; in section 4.2.
2. Mandatory files of front end are described. See 4.3, 4.4, and 4.5.
3. Few tips are given on joining minimal front end with existing language parser. Read 4.6.
4. Some versioning issues are resolved in 4.6.2.

[1] http://projects.almad.net/gcc-algol/browser/trunk/gcc/algol60?rev=58
[2] http://svn.gna.org/viewcvs/gsc/branches/hello-world/

4.2 Setting Up Source Base

4.2.1 GCC Reference Version

This work is being written with GCC version 4.2 in mind.

4.2.2 Create Front End Directory

Each front end lives in a subdirectory of its own. Decide on the name of your fronted and create a subdirectory in gcc/gcc/. For sake of demonstration, I will use the hypothetical language Überlang.

```
$ cd gcc/gcc/
$ mkdir uberlang
```

Then you will have to populate the front end directory with source files. This is covered in sections that follow.

4.3 uberlang1.c

While it is possible to use other languages than C to write GCC front end, most of GCC is written in C. There are exceptions, e.g. Java front end is written in C++, and Ada front end is written in Ada. For now, I'll assume that you will start up with simple C source base.

The file uberlang1.c will make up the compiler itself, or the front-end-specific part thereof. It has to contain all the necessary langhooks, init functions, and data structures that the rest of the compiler expect from your front end.

4.3.1 Custom Data

Each frontend can have its own data in various GCC data nodes. Since there are five kinds of these nodes, five structures will have to be defined:

```
// language-specific identifier data
struct lang_identifier;
// language-specific tree node data
union lang_tree_node;
// language-specific type data
struct lang_type
// language-specific declaration data
struct lang_decl
// language-specific function data
struct language_function
```

All structures may be empty, except for the lang_identifier, which has to contain common part of identifier definition:

```
struct lang_identifier GTY(()) {
   struct tree_identifier common;
};
```

This example shows also the GTY(()) tag, necessary for proper garbage collection scanners to be generated. Each structure has to have such a tag.

As the last thing regarding the custom data, you have to provide definitions of tree_code_type, tree_code_length, and tree_code_name arrays. These have to contain both the system node types, and your own types. This is done with the following trick:

```
#define DEFTREECODE(SYM, NAME, TYPE, LENGTH) TYPE,
const enum tree_code_class tree_code_type[] = {
#include "tree.def"
  tcc_exceptional
};
#undef DEFTREECODE
```

The file gcc/gcc/tree.def contains data ready for preprocessing, with records neatly defined in columns of DEFTREECODE calls. By including this file three times, each time with different definition of DEFTREECODE, you fill the three arrays.

The last member of array has the class of tcc_exceptional, which serves here as a sentinel. The name of the last node can be arbitrary, e.g. *"@@dummy"*, and the length will be 0.

4.3.2 Langhooks

The list of all various langhooks is long. You can find it in file gcc/gcc/langhooks.h together with comments. Default value of each langhook is in gcc/gcc/langhooks-def.h. Both files have to be #included in uberlang1.c. If you are not comfortable with value (i.e. name) of any given hook, the following mantra is used to redefine it:

```
#undef LANG_HOOKS_INIT
#define LANG_HOOKS_INIT uberlang_init
```

Many hooks are predefined in GCC. The file gcc/gcc/langhooks.c contains these definitions. Any langhooks not defined there have to be provided by the front end—even if that will be blind definition from the beginning. Review of the langhooks that have to be in minimal front end, together with the action that has to be taken, is in table 4.1. Review of the various actions follow.

4.3.3 Blinded Out Langhooks

By blinding langhook out, I have in mind something like this:

```
void insert_block (tree block ATTRIBUTE_UNUSED) {
  gcc_unreachable ();
}
```

Function gcc_unreachable aborts the compiler if it is ever hit by a thread of execution. If that happens, error message is printed out denoting the file and line where the offending gcc_unreachable appeared.

The ATTRIBUTE_UNUSED cookie tells GCC that given variable may be unused in the function body. GCC has a very pedantic flag settings during bootstrapping, and huge

Langhook	Action
LANG_HOOKS_GLOBAL_BINDINGS_P	gcc_unreachable
LANG_HOOKS_INSERT_BLOCK	gcc_unreachable
LANG_HOOKS_PUSHDECL	gcc_unreachable
LANG_HOOKS_BUILTIN_FUNCTION	gcc_unreachable
LANG_HOOKS_TYPE_FOR_MODE	gcc_unreachable
LANG_HOOKS_TYPE_FOR_SIZE	gcc_unreachable
LANG_HOOKS_UNSIGNED_TYPE	gcc_unreachable
LANG_HOOKS_SIGNED_TYPE	gcc_unreachable
LANG_HOOKS_SIGNED_OR_UNSIGNED_TYPE	gcc_unreachable
LANG_HOOKS_MARK_ADDRESSABLE	gcc_unreachable
tree convert (**tree** type, **tree** expr)	gcc_unreachable
LANG_HOOKS_INIT_OPTIONS	empty
LANG_HOOKS_HANDLE_OPTION	empty
LANG_HOOKS_POST_OPTIONS	empty
LANG_HOOKS_FINISH	empty
LANG_HOOKS_INIT	see 4.3.5
LANG_HOOKS_PARSE_FILE	see 4.3.6

Figure 4.1: Breakdown of langhooks in minimal front end.

amounts of warnings are printed out for various legal, yet suspicious C constructs. The ATTRIBUTE_ cookies work as GCC pacifiers, so that you are actually able to spot any new warnings.

The langhooks that will be blinded out could in fact be empty. But having them blinded out will make sure that GCC fails early and noisily[7] once it tries to use them. This condition is easier to track down than when GCC gets nonsensical value in return, or thinks that langhook did its job, and fails on its own internal assertion thousands of cycles later.

The function convert is not a langhook. It can be blinded out, too, but it has to be here, and as opposed to langhook functions, it must not be **static**, because GCC calls is directly.

4.3.4 Empty Langhooks

Some more langhooks need to contain non-failing code, because they are called always. They are not required to do anything useful however, and their bodies may be empty. This also includes langhooks that return NULL_TREE, one of the arguments, etc.

4.3.5 Initialization And Deinitialization

The setup and teardown are implemented by LANG_HOOKS_INIT and LANG_HOOKS_FINISH langhooks. The finish langhook is among the empty ones, because it doesn't have to contain any code just now. It is called after all compilation is done, and you can use it to clean up whatever is necessary. Initialization, however, can't be empty.

Most of GCC is self-initializing, but there are several functions that need to be called. Quite possibly your front end requires some initializations of its own, e.g. to build its own tree nodes for language standard types. Init hook is a convenient place to do so.

During initialization you want to call the following functions:

- build_common_tree_nodes, which creates nodes for all integer types.
- set_sizetype, which is used to set the type of the internal equivalent of **size_t**; it is simplest to set this always to long_unsigned_type_node, which was created in previous call.
- build_common_tree_nodes_2 uses size type to create few more tree types.

4.3.6 Parse File Langhook

This is the langhook that does all the parsing, semantic analysis and code generation needed for the input file. This is where our **return** 7; code resides.

The code itself is listed in figure 4.2. It's quite verbose, and of course by now nothing was written about the GCC framework, so the functions are not likely to tell you much. I'll just cover the basic points, and leave the full elaboration for the later chapters.

The argument debug of the langhook is nonzero, if debugging messages should be dumped to the standard error. See description of -dy in [4, chapter GCC Command Options].

The macros TREE_PUBLIC, DECL_CONTEXT and similar are used for runtime access into discriminated union that makes up node of GENERIC tree. They set various flags and auxiliary values of node in question. In a development tree, there is a runtime check whether using the macro on a given node is legal. This check goes away in final build. You will want to enable this checking (via configure flags) when doing development inside stable GCC tree, otherwise you will slowly go insane from all the ICEs that will pop up.

The functions build1, build2 and build3 are used to build, respectively, unary, binary, and ternary GENERIC nodes. The first argument is the node type, the second argument is the type of expression, and the remaining arguments are the children of the node being built.

Meaning of other functions can be more or less deduced from their name, and the details are not important just now.

4.4 Language Specific Components of the Driver

The lang-specs.h file describes your front end to the GCC driver. It tells the driver the file extensions that, when seen on the command line, should cause GCC to invoke your front end. It also gives the driver some instructions for what other programs must be run, such as whether the assembler should be run after your front end and how to pass or modify certain command-line options. It may take a while to write this file, as specs are their own strange language[8].

The contents of the file for the minimal front end will be pretty simple, along these lines:

```
{".ubl", "@ubl", 0, 0, 0},
{".UBL", "@ubl", 0, 0, 0},
{"@ubl", "%{!E:uberlang1 %i %(cc1_options) %{I*}"
         "%{!fsyntax-only:%(invoke_as)}}", 0, 0, 0},
```

```
void
algol60_parse_file (int debug ATTRIBUTE_UNUSED)
{
  /* Build declaration of ‘main’ functions */
  tree  main_type =
    build_function_type_list (integer_type_node, NULL_TREE);
  tree  decl = build_function_decl ("main", main_type);
  DECL_CONTEXT (decl) = NULL_TREE;
  TREE_PUBLIC (decl) = 1;
  DECL_ARTIFICIAL (fndecl) = 0;
  DECL_EXTERNAL (decl) = 0;
  TREE_STATIC (decl) = 0;
  DECL_ARGUMENTS (decl) = NULL_TREE;
  rest_of_decl_compilation (decl, 1, 0);

  /* Build RESULT DECLARATION, which is used for storing function
     return value. */
  tree resultdecl
    = build_decl (RESULT_DECL, NULL_TREE, integer_type_node);

  /* Build function BODY:
     ‘(<block> (return (init_expr resultdecl (int_cst 7))))’ */
  tree assign = build2 (INIT_EXPR, void_type_node, resultdecl,
                        build_int_cst (integer_type_node, 7));
  TREE_USED (assign) = 1;
  TREE_SIDE_EFFECTS (assign) = 1;
  tree body = build1 (RETURN_EXPR, void_type_node, assign);
  TREE_USED (body) = 1;
  tree block = build_block (NULL_TREE, NULL_TREE, NULL_TREE, NULL_TREE);
  DECL_SAVED_TREE (decl) = build3 (BIND_EXPR, void_type_node,
                                   NULL_TREE, body, block);
  DECL_INITIAL (decl) = block;
  TREE_USED (block) = 1;

  /* Emit code for the function */
  allocate_struct_function (decl);
  gimplify_function_tree (decl);
  cgraph_finalize_function (decl, false);
  cgraph_finalize_compilation_unit ();
  cgraph_optimize ();
}
```

Figure 4.2: Program listing of parse file langhook

This reads, roughly: if no -E is seen on commandline, invoke uberlang1, which is the name of frontend binary. Further, if -fsyntax-only is not specified, assemble the resulting file. Note how the second action is "embedded" in the first, thus providing the context under which it should be considered.

GCC's spec language is described in file gcc/gcc/gcc.c.

The lang.opt file is an option specification file. It can be empty for now. More about compiler command line options is written in section 7.2.

The spec.c file contains language-specific GCC driver components. In particular, two functions and one variable. Function lang_specific_driver is given a vector of command line arguments and is free to do any processing, including reordering and changing the vector, before the main GCC routines take over. This is used for linking in language runtime libraries, see 7.4 for details.

The other function, lang_specific_pre_link, is called before linking is done. Whatever language processing you need can be done here. The function has to return 0 on success and -1 on failure. We don't need this function, and will leave it empty.

The variable lang_specific_extra_outfiles is used in concert with lang_specific_pre_link, and keeps track of the number of extra output files that lang_specific_pre_link may generate.

Note that this special processing is only done, if you run the compilation through the language-specific driver, e.g. g++ for C++ or gubl for Überlang.

4.5 Build System Integration

The config-lang.in file is a sort of high-level descriptor of your front end. It contains variables for toplevel configure (i.e. it's written in a shell syntax). The file is very simple overall and only defines few variables:

language=*"uberlang"*
compilers=*"uberlang1$(exeext)"*
gtfiles=*"$(srcdir)/uberlang/uberlang1.c"*

The variable language defines the name of the front end as recognized by the build system. This is really a name of the language, it doesn't have to match name of the front end subdirectory. However the name will be used in make targets, and users will call this your language on configure command line when they'll wish to include it in GCC build. For this reason the name should be 7-bit ASCII clean.

Variable compilers contains list of all compilers created during the build.

Very important variable gtfiles contains the list of files that should be scanned for GTY(()) tags. Garbage collection scanners are built from these.

More is written about this file, and which variables are recognized, in [3, section Anatomy of a Language Front End].

The Make-lang.in file serves as a fragment of Makefile.in, from which configure will eventually create Makefile. GCC build machinery does calls into front end-specific Makefile, and you have to implement certain targets.

This file has to contain many targets, exactly which ones is specified in [3, section The Front End *language* Directory]. But like in case of C language hooks, most of these may be empty. The only ones necessary to make whole thing tick are listed in figure 4.3.

The compilation starts at the root component, named after your front end language. It is then dispatched to build the two components that make up language support in GCC: language-specific driver, and the compiler proper. The driver is called gubl in our case, like GCC Überlang [3]. The compiler proper is called uberlang1, which is standard naming of compiler component in GCC. The variables that this Makefile fragment uses, are inherited from parental make invocation.

Note that the fragment actually doesn't present hooks, but make rules. Only the first rule in the list is actually a hook. The first two rules, then, have not only the dependence list, but also a body. This doesn't hold for the other two rules, which only contain dependence list, and use mechanism of implicit rules to sort out the exact order of compilation.

Because other hooks are empty, including uberlang.install-* family, the minimal front end won't be installable. You will have to run it like this:

```
$ pwd
/path/to/build/
$ ./gcc/uberlang1 ./file.ubl &>/dev/null && cat ./file.s
        .file 1 "file.ubl"
        .abicalls
        .text
etc ...
```

4.6 Integration of Existing Parser

Chances are you already have the language processor written, and need to integrate it into the GCC. (By the way, this is the approach that I have taken. Parts of Algol compiler were written long before I started poking GCC.)

4.6.1 Joining In The Minimal Front End

At this point, you have two components in hand: a minimal GCC front end, and your compiler. The first step to get them joined should be inclusion of your compiler into the build system. If you don't want to port entire compiler over to GCC's Makefile machinery, you will have to invent some mechanism of transferring the build into your subdirectory. Basically $(MAKE) -C should be enough, but you will have to take care of passing necessary build/host/target trichotomy variables and paths over to your build system, and respecting them there. This in fact holds for all various variables that are passed from one recursive make instance to the other.

You may want to take advantage of certain GCC's services. E.g. error reporting, integrated test suite, building runtime library as part of overall build; or you may wish

[3] A more customary meaning of this abbreviation would be GNU Überlang. But there are lots of restrictions for one's project to become part of GNU.

```
# General hook
uberlang: gubl$(exeext) uberlang1$(exeext)
.PHONY: uberlang

# Compiler proper
uberlang1$(exeext): uberlang/uberlang1.o $(BACKEND) $(LIBSDEPS)
        $(CC) $(ALL_CFLAGS) $(LDFLAGS) -lm -o $@ \
          uberlang/uberlang1.o $(BACKEND) $(LIBS) attribs.o

# Language-specific driver
gubl$(exeext): gcc.o version.o prefix.o intl.o $(EXTRA_GCC_OBJS) \
        $(LIBDEPS) uberlang/spec.o
        $(CC) $(ALL_CFLAGS) $(LDFLAGS) -o $@ uberlang/spec.o \
          gcc.o version.o prefix.o intl.o $(EXTRA_GCC_OBJS) $(LIBS)

# Compiler front end
uberlang/uberlang1.o: uberlang/uberlang1.c $(CONFIG_H) $(SYSTEM_H) \
        coretypes.h $(TM_H) toplev.h $(GGC_H) $(CGRAPH_H) \
        $(TREE_DUMP_H) $(TREE_GIMPLE_H) $(LANGHOOKS_DEF_H) langhooks.h \
          tree.def gt-uberlang-uberlang1.h gtype-uberlang.h

# Language-specific driver component
uberlang/spec.o: algol60/spec.c $(CONFIG_H) $(SYSTEM_H) \
        coretypes.h diagnostic.h $(TREE_H) flags.h toplev.h langhooks.h $(TM_H)
```

Figure 4.3: Program listing of Make-lang.in essentials

to extend your language with various GCC-isms [4], such as C-like preprocessing or inline assembly.

For special-casing the code that's built as part of GCC, you can use IN_GCC preprocessor variable. It's #defined when the file is built as part of GCC build process.

It's not usually a good idea to mess GCC and an existing code. GCC's garbage collecting engine has infamous habit of poisoning certain memory-related functions (e.g. malloc, calloc, strdup). Using these symbols then leads to code that is classified by cpp:

```
cpp error: attempt to use poisoned malloc
```

Hand-managed code is OK with GCC, so it's best to keep the collector off already written code base. On the other hand, if you know you are writing what will become part of GCC, it is perhaps better to use garbage collector from the beginning. It saves some nerves.

4.6.2 Versioning Issues

Most probably, you will want to keep your front end versioned separately from the GCC itself. I will describe the approach that I'm taking. Toplevel trunk/ directory (I'm using subversion, but of course, you are free to keep your stuff versioned to your likings) contains the following subdirectories:

```
trunk/doc/
trunk/gcc/
trunk/gcc/libga60/
trunk/gcc/gcc/
trunk/gcc/gcc/algol60/
trunk/gcc/gcc/testsuite/
trunk/gcc/gcc/testsuite/lib/
trunk/gcc/gcc/testsuite/algol60.dg/
```

More will be written about the trunk/gcc/libga60/ and trunk/gcc/gcc/testsuite/ in their respective sections, 7.4 and 7.5. For now, you can skip their description, if you so wish.

The toplevel split to trunk/doc/ and trunk/gcc/ is here to keep my diploma thesis texts separated from the GCC development part. This may come in handy in your case, too, if you want to keep the GCC subtree clean, and store e.g. documentation, scripts, etc. separately.

The subtree rooted at trunk/gcc/ mimics GCC's distribution subtree (e.g. gcc-4.2.0/). This is advantageous for two reasons. First, you can just tar and zip the contents of directory trunk/gcc/, and distribution package is ready. User will just enter the GCC's distribution directory, unpack your front end here, do necessary patching, and GCC tree is ready to build your front end. Second, it keeps the things tidy.

Then I just symlink the files from GCC tree into my front end tree. This is only possible on systems that support symlinks, but with GCC being unixish compiler, it is reasonable requirement. In several cases, symlinking works as intended even on directory level:

[4] Yes, I know this term is pejorative. But this doesn't decrease usefulness of certain GCC-specific constructs.

```
gcc/gcc/algol60
  -> /a/path/to/gcc-algol/trunk/gcc/gcc/algol60
gcc/gcc/testsuite/algol60.dg
  -> /a/path/to/gcc-algol/trunk/gcc/gcc/testsuite/algol60.dg
```

But for test suite, you will have to add two files into subdirectory gcc/gcc/testsuite/lib/, and you have to symlink those two files explicitly (we need to "merge" the two directory subtrees, which is not simply done in most OSes):

```
gcc/gcc/testsuite/lib/algol60-dg.exp
  -> /a/path/to/gcc-algol/trunk/gcc/gcc/testsuite/lib/algol60-dg.exp
gcc/gcc/testsuite/lib/algol60.exp
  -> /a/path/to/gcc-algol/trunk/gcc/gcc/testsuite/lib/algol60.exp
```

The directory that really causes trouble is gcc/libga60/, a runtime library subdirectory. Build scripts will use the path in a relative manner: gcc/libga60/../. This doesn't work with symlinks, because the ../ component will point to the target directory's ../, not back to gcc/. For this reason I had to create gcc/libga60/ as an ordinary directory, and symlink there all the files from front end's gcc/libga60/. You will have to either take care and symlink back by hand when someone adds new files into version system (maybe with support of nice VCS hook), or store all source files in subdirectory e.g. gcc/libga60/src/, symlink that, and only keep in gcc/libga60/ the build machinery files that need to be there.

4.7 Summary

- Each front end lives in a directory of its own. Even for minimal front end, several files are necessary to honor requirements of GCC build system. They are lang-specs.h with the description of actions that should be taken depending on the source file suffix and command line switches; lang.opt with the description of command line switches specific to your front end (this file may be empty); config-lang.in with meta-data about your front end; Make-lang.in with the description of build itself; spec.c with language-specific components of GCC compilation process; and *your-language-name*1.c with the code itself.

- The file *your-language-name*1.c, in the past chapter conveniently called uberlang1.c, contains the GCC that is required by the rest of GCC stack. This includes definitions of custom data structures and language hooks. Most language hooks, though required to be defined, may be empty of blinded out. Some of them must contain a code, most prominently INIT and PARSE langhooks.

- To keep GCC garbage collector happy, you will have to annotate data structures with GTY(()) tags. Furthermore, all files that contain such annotations have to be included in a list gtfiles in config-lang.in.

Chapter 5

The Algol 60 GCC Front End

The Algol 60 GCC front end has been written in an attempt to get a first hand experience on work with GCC. Algol 60 itself turned out to be a kinky language, and as of now, is not yet fully implemented. This chapter presents overview of the compiler architecture and implementation, and ways it handles certain interesting Algol 60 constructs.

5.1 `gcc-algol` the Project

The programming language Algol 60 was picked purposefully, because language such as this is best fit for GCC. It is structured, with static typing, has arrays and functions, and will profit from support of runtime library. Lastly it has a rigorous standard, which is not that important in and on itself, but why invent new language?

The Algol 60 compiler was started as a project independent on GCC[1]. I knew I would plug it in GCC one day, but I specifically avoided any relation to GCC before the time came. This helped me simulate the situation where the company or an individual wants to port existing work to GCC back end[2].

The front end is written in a language C. As mentioned before, C is natural fit for GCC. If you have the choice, choose either C, or other GCC language that can compile transitively from C. Front end written in such a way is simpler to distribute and build at user sites.

I strove to avoid any extra dependencies of the front end, to the end of writing linked list and generic string modules from scratch just for the purpose of Algol parser. These two modules are the only library functionality beyond pure libc—even symbol tables are stored in linked lists [3].

5.2 Overall Architecture

From the bird's eye view, Algol compiler is a straightforward compiler: it has Flex-based lexical analyser, Yacc-based parser, and several C modules with AST and semantic analysis. The compiler itself consists of two main parts: a front end, whose work is to transform stream of bytes into AST; and back end, whose work is to convert AST into GENERIC. Note that under this scenario, the back end of the compiler is actually front

[1] Despite the language being picked to suit GCC!

[2] In the sense of GCC itself being the back end.

[3] I will fix this, I promise.

end of GCC. For the rest of this chapter, I will use the term *compiler* to denote the part of ga60, that transforms the stream of bytes into GENERIC.

Algol compiler uses encapsulation heavily. There is no (official) way to get inside structures, everything is passed as a pointer to undefined structure[4]. Only in the module that handles that particular type is the structure actually defined, so that the implementation has full, unrestricted access to bits.

The encapsulation zeal is exposed by use of visitors. There are certain polymorphic structures in Algol AST (more about polymorphism and subtyping in Algol 60 compiler will be written in section 5.2.2). For example expressions may be numbers, variable references, unary and binary expressions, function calls, etc. All expression kinds are passed around as the same opaque expression_t pointer. To do any action depending on exact type of expression, one has to define a *visitor* over the expression structure, and let the visitor dispatch on the desired expression object.

Other means of achieving safety include strong trend towards checking. Whatever can be checked statically, by compiler giving a warning, is checked so. If the warning can be turned to full error, perhaps with some extra coding, so much the better. If compiler can't check that, runtime will.

5.2.1 Visitors

The figure 5.1 presents an example of visitor creation and usage. You can see that the expression visitor is created by a dedicated function new_visitor_expr, which accepts one argument for each expression kind. The arguments have to be callback_t pointers: to convert function pointer to callback, you have to pass it through appropriate callback builder, a60_expr_callback in this case. Callback builder doesn't do any actual work, it just typecasts to pointer, but if you try to pass wrongly typed function pointer, it will bail out. Visitor builder, in turn, requires exact number of callback arguments.

In addition, the visitor builder knows it builds visitor for expression type, and when in debug mode, it records that information into visitor itself. Visitor dispatcher then *dynamically* checks that that visitor is dispatched over an expression, and not e.g. a statement. In release mode, the check goes away.

5.2.2 Dynamic Typing

To implement rigorous runtime checking, each polymorphic objects carries around two identifiers: a *kind*, which identifies subtype (e.g. "number"), and *signature*, which identifies type (e.g. "expression"). Kind is actually used to model RTTI, but not so signature. Given a void*, signature makes it possible to decide whether it points to (e.g.) expression or not. This property is not used to emulate RTTI, but rather to provide additional type safety, for example in a code like this:

```
void * mystery = slist_front (state->for_statements);
statement_t * parent_for_stmt = a60_as_statement (mystery);
//statement_t * parent_for_stmt = (statement_t *) mystery;
```

On first line, mysterious object is extracted from singly-linked list. Instead of typecasting it like in commented-out code, dedicated function is called, that *actually checks* exact

[4]Structure, not **void**. It is necessary to use void pointer at several places, but number of such instances is kept as low as possible.

```
/// Excerpt from compiler context.
struct struct_al60l_bind_state_t
{
  visitor_t * expression_build_generic;
};

/// An example callback function.
/// Builds GENERIC for integer constant node.
void *
expr_int_build_generic (expression_t * self, void * state)
{
  tree ret = build_int_cst (integer_type_node, expr_int_value (self));
  return ret;
}

/// Creates new compiler context.
al60l_bind_state_t *
new_bind_state (void)
{
  al60l_bind_state_t * ret = xmalloc (sizeof (al60l_bind_state_t));
  ret->expression_build_generic = new_visitor_expr (
      a60_expr_callback (expr_int_build_generic),
      a60_expr_callback (expr_real_build_generic),
      a60_expr_callback (expr_string_build_generic),
      a60_expr_callback (expr_bool_build_generic),
      a60_expr_callback (expr_idref_build_generic),
      a60_expr_callback (expr_if_build_generic),
      a60_expr_callback (expr_binary_build_generic),
      a60_expr_callback (expr_unary_build_generic),
      a60_expr_callback (expr_call_build_generic),
      a60_expr_callback (expr_subscript_build_generic)
  );
  return ret;
}

/// Wrapper function, uses visitor to build GENERIC from given expression.
tree
expr_build_generic (expression_t * expression, al60l_bind_state_t * state)
{
  return (tree)a60_visitor_dispatch (state->expression_build_generic,
                                     expression, expression, state);
}
```

Figure 5.1: Example use of visitors in Algol 60 front end.

```
static void *
private_check_expr_lvalue (void * ptr, void * data ATTRIBUTE_UNUSED)
{
  expression_t * expr = a60_as_expression (ptr);
  if (expr && !expr_is_lvalue (expr))
    expr = NULL;
  return expr;
}

statement_t *
new_stmt_assign (cursor_t * cursor, slist_t * lhss, expression_t * rhs)
{
  statement_t * ret = private_new_statement (sk_assign, cursor, NULL);
  slist_set_type (lhss, private_check_expr_lvalue, NULL);
  ret->assign.lhss = lhss;
  ret->assign.rhs = rhs;
  return ret;
}
```

Figure 5.2: Example use of typed slists in ALGOL 60 front end.

type, and if it doesn't match, aborts. You can't place a condition on the result of type checking, the mechanism is here to *prevent errors*, not to emulate dynamic typing systems.

In release mode, the checks go away, and a60_as_statement ends up being a macro that expands to direct C typecast.

This same dynamic typing mechanism is used for visitors. Visitor knows which object type it should dispatch on, and given a visitor and an object (passed in as void*), it can decide whether the visitor matches the object type.

Dynamic typing is used for implementation of *typed slists*, i.e. singly-linked lists with associated predicate that is guaranteed to hold for its elements. For example, in ALGOL 60, each assignment statement is made up from a vector of left hand sides, which have to be lvalues, and one right hand side. To express this property, code such as the one at figure 5.2 can be written.

A function private_check_expr_lvalue is a predicate, that, given a void* pointer, can decide whether the given pointer points to lvalue expression. It does so by first checking that void* is expression at all, and then asks expression module (because it's safe by now) on lvalue status. This predicate is attached as a type to the passed-in list of left hand sides (lhss), thus providing additional checking in case anything slipped between fingers of yacc parser. Slist module would abort if one of the elements didn't pass the test, or an element that doesn't pass was added.

As is the case for other dynamic features, in release mode this one goes completely away, so there is neither a performance const, nor a memory overhead.

5.3 The Tour of ALGOL 60

Perhaps whole ALGOL 60 can be explained in terms of the following three fundamental abstractions: statements, expressions and declarations. In this section, I will give a glimpse

of Algol 60. This serves a purpose of laying foundations for next section, where an implementation of various Algol 60 constructs is explained. Also in this section are described bits and pieces of the AST design: how is each Algol element modeled.

5.3.1 Expressions

The Algol 60 spec groups together two notions: numerical expressions (which includes literals, arithmetic and Boolean expressions, variable references including array access, function calls), and designational expressions, which more naturally belong to flow control. In the compiler, these two kinds are disjoint: computed, numerical expressions are described by a type expression_t, and designational expressions by desig_expr_t.

Expressions use natural polymorphic organization: literals and symbols form leaf expressions, and more complex expressions, such as binary, are recursively composed of other expressions.

Despite their specification together with other expressions, designational expressions are completely disjoint polymorphic hierarchy, used in other context, namely as targets of the **goto** statement. There are three subtypes: a label reference, a switch expression, and conditional designational expression. The following example gives an overview of what's possible to express using the notion of designational expressions[5]:

```
begin
   switch b := q, r, 5;
   switch a := q, if k > 005 then r else 5, b[k];
   integer array ar[1:100]; integer ix, k;
   initialize(ar, ix, k);
   goto a[ar[ix]];
q: puts('q!');
r: puts('r!');
05:puts('5!');
end;
```

As you can see, certain designational expressions use ordinary expressions as leaves (that's the case of switch expression, which can be seen at the **goto** statement and as part of declaration of **switch** a).

The example also shows that it's possible to use both numerical and symbolic labels. Fortunately, expressions such as **goto** 1+1 are disallowed, but still the labels 5, 05 and 005 are considered equal. This is handled in parser, who translates the label into canonical representation right away.

5.3.2 Statements

The repertoire of Algol 60 is on par with most C-like languages up to these days. Supported statements are assignment, goto, conditional, and for. Most of them work as expected. Assignment statement supports assignment of one value to several variables (without reevaluation). Goto has unusually rich syntax for computation of target, see the description of designational expressions in previous section. For has also rich syntax for iteration, see this example:

[5] It doesn't compute anything interesting, but uses all computed **goto** features Algol 60 has, except for non-local jump.

```
begin
   integer i;
   for i := 1, i + 1 while i < 10, 100, 105 step 2 until 181 do
      perform_interesting_stuff(i);
end;
```

The compiler uses auxiliary polymorphic type for_elmt_t to keep the list of iteration branches. It has exactly three subtypes: an expression, a while-kind of branch, and an until-kind of branch.

Statements can be grouped to blocks or compound statements[6]. Both abstractions are described by the same type: a container with associated symbol table, a60_symtab_t[7].

5.3.3 Declarations

The third fundamental element of ALGOL 60 programs are declarations. ALGOL distinguishes switch declarations (those were seen in an example in section 5.3.1), variable declarations, and procedure declarations.

The simplest case is switch declaration: it consists of series of designational expressions, which was described earlier. When used in a goto statement, switch declaration is used this way:

```
goto sw[expr];
```

Where sw is a switch, and expr is arbitrary integer expression. The flow of control will end up in the branch with the same number, as is the result of expr, counting from one.

Slightly more complicated case is with variables, mostly because of types. ALGOL 60 supports basic types such as integers, real numbers and Boolean values, and one compound type: an array of basic types. Number of dimensions of array is static property of the declaration, but the dimensions themselves are computed. Each variable can be marked "own", which is basically the same as static in C. The last supported type is string, which is only allowed as procedure argument, and cannot be otherwise manipulated or assigned by ALGOL program. Inevitably, each string in ALGOL program ends up passed to native code procedure. Few examples of variable declarations and types:

```
begin
   integer a;
   Boolean b;
   own real c
   integer array d[1:5];
   own integer array c[2:6,3:9,4:10];
end;
```

The declaration on its own right is a procedure. Procedures are *the* reason why ALGOL 60 is kinky. In particular, formal argument of procedure can be declared as *pass by name*,

[6]The difference being that block can contain declarations, while compound statement cannot. When the grammar allows for compound command, it also allows for block, so it would seem that the distinction is nil. But labels are defined at the most enclosing *block*, so the compiler has to keep track of which container is actually a block, and which is mere compound statement.

[7]Non-clashing naming convention was introduced quite late in development cycle. Symbol table is as of this writing the only type that uses the a60_ prefix.

which basically means that the actual argument of this formal parameter is reevaluated each time it's hit in control flow—like if the function was actually a macro, and replaced each occurrence of the formal with passed-in actual.

Also, by-name parameters don't have to be typed, an unusual feature in compiled language. Coupled with the way ALGOL 60 handles parameter-less functions, it's possible to pass in e.g. either an integer, or integer function without a parameter. Depending on call site in effect, and the way each identifier ends up resolved, compiler has to emit either variable access, or function call.

Yet another funny thing about the spec is that the resolution of non-local variable references is done on a call site—another macro-like feature of ALGOL 60. These three features combined are presented in a famous Man-or-Boy test of Donald Knuth's:

```
begin
  real procedure A (k, x1, x2, x3, x4, x5);
  value k; integer k;
  begin
    real procedure B;
    begin k := k - 1;
          B := A := A (k, B, x1, x2, x3, x4);
    end;
    if k <= 0 then A := x4 + x5 else B;
  end;
  outreal (A (10, 1, -1, -1, 1, 0));
end;
```

My humble ALGOL 60 compiler doesn't pass this test: not by a great distance. It's on the way there, but ended up being half-finished in this respect. This is perhaps the worst possible scenario, because functions, as of this writing, simply don't work: not even semantic analysis is done, let alone code generation.

5.4 Implementation

5.4.1 Polymorphism and Visitors

Visitors and dynamic typing go hand in hand, and are both implemented by the same module. Each module, that wants to participate in this scheme, has to list the field visitable_t (from visitor-impl.h) as the first member of its data structure:

```
struct struct_expression_t {
  visitable_t base;
  // rest of structure as usual
};
```

The base structure contains a signature (only in debug mode), and a kind distinguisher.

Kind is simple int, and its values would typically be determined by some internal enum. Signature is char const*, and is used as follows:

```
static char const * const private_expression_signature = "expression";
static expression_t *
```

```
private_new_expr (cursor_t * location, expr_kind_t kind) {
  expression_t * ret = calloc (1, sizeof (expression_t));
#ifndef NDEBUG
  ret->base.signature = private_expression_signature;
#endif
// ...
}
```

First, using address of string gives us a unique pointer, usable for distinguishing between structure types. Besides this, using strings is extremely useful for debugging and diagnostic messages. All you have to do to find out the type of mysterious object is to dereference the pointer and interpret what it points to as a string.

When creating a visitor, address of signature variable is used instead of signature itself. Before dispatch, the visitor performs a simple check:

```
#ifndef NDEBUG
  // check signature
  char * obj_sig = *(char const* const*)object;
  char * vis_sig = **(char const* const* *)visitor;
  if (obj_sig != vis_sig) {
    fprintf (stderr,
             "error: visitor %p for `%s' dispatches over `%s'.\n",
             (void*)visitor, vis_sig, obj_sig);
    abort ();
  }
#endif
```

As you can see, the very typing mechanism used gives us ability to deduce what's wrong, and inform user in a meaningful way:

```
error: visitor 0x0804d200 for `expression' dispatches over `statement'.
```

5.4.2 Messages and Locus Support

Almost every part of Algol 60 needs to inform users about problems it found. This includes warnings about constructs that are legal, but not moral or wholesome, or errors for constructs that are disallowed by the spec.

Messages in the compiler are formatted in usual GNU format:

```
./fail-fun2.a60:5: warning: function `foo1' has an implicit parameter `a'
./fail-fun2.a60:5: error: type mismatch in expression `(a + 1)':
  'Boolean' + 'integer'
./fail-fun2.a60:9: error: at this point in file.
```

Proper error reporting, as well as proper debuginfo generation, requires an information about location of various program constructs. This support is introduced via cursor_t structure. It is defined the way all other structures are, namely it uses signature for dynamic type checking, is heap-allocated, and opaque.

Each message is output to log with given *severity*, which can be one of *debug*, *info*, *warning*, *error* or *fatal error*. Each log has associated a filter threshold: messages with severity below the threshold are thrown away, the rest is written.

Log and cursor are two structures at the edge between GCC and Algol specific code. By default, they use their own structures and methods, but when compiled with GCC (as is checked by IN_GCC preprocessor variable), they use GCC services: logger uses GCC "native" error reporting tools, and cursor is capable of conversion to GCC *locus*. More will be written about these topics in 6.5.1 and 7.3.

5.4.3 Semantic Checks

The front end does the compilation in several passes. First pass consists of lexical and syntax analysis, processed in parallel. The output of this phase is raw AST which represents merely a syntactic structure of source text. Certain semantic checks are already performed in this stage, e.g. identifiers are checked whether they are unique.

Second pass is true semantic analysis. The compiler recursively descends through whole syntax tree, performs context checks, computes result types, etc. Each module handles its own semantic checks in a function named something_resolve_symbols, e.g. stmt_resolve_symbols. During semantic analysis, three special types (i.e. subtypes of type_t) come into play: implicit, unknown and any.

The latter one, any, is used for querying. It is possible to ask e.g. for "a symbol that is an array of any type". This is used mainly to look up functions by their signatures.

The type implicit is used to track implicit arguments of functions. When the compiler parses the body of function declaration (which is a good thing to do, because it will discover certain errors even if the function is not called), and it fails to resolve variable reference in the body itself, it marks that variable as an implicit parameter of the function. Such variable is then assigned a placeholder type of implicit.

Third mentioned type was unknown. This type is assigned to expression whose type is not known, most often because there was an error found in the expression. When the resolver hits this type, it will not do further type checking, and will not produce any error messages. This is done at the leaf expression that caused the type to be unknown in the first place.

Third pass is compilation into GENERIC, so called Binder. This step is performed only when no previous pass resulted in an error. The compilation is, too, done by a recursive descend through AST. Exactly how is the compilation done will be described in next chapter.

Chapter 6

Targeting GCC

This chapter describes how to generate GCC intermediate language called GENERIC, and uses a lot of examples from the Algol Binder.

In comparison with GCC internals [3], the description in this thesis is kept in more abstract level. I consider GCC to be a target of compilation process, and as such I'm more interested in how the nodes are created and how to express various imperative constructs. GCC internals documentation is directed also towards the analysis of GENERIC and internal structure of nodes.

6.1 GENERIC in General

GENERIC is a tree-like language, similar in nature to Algol AST. It's representation in GCC is via polymorphic nodes of type **tree** :

```
tree ret = NULL_TREE;
```

tree is itself a pointer type, so **tree** * denotes a *pointer to* **tree** . The value NULL_TREE is thus analogy to NULL pointer of the **void*** world. Pointer to **tree** is occasionally also used, e.g. as an output parameter of a function.

Each node has an arity, i.e. a number of child nodes that it needs; and *signature*[1], which describes what type each of children nodes has to have.

GCC defines nodes in a catalogue, stored in file gcc/tree.def. This is basically ordinary header file, included in compiler proper .c file (e.g. algol601.c). Refer back to example 4.3.1 to see how it's used. Properties of each node (symbolic and string name, class and arity) are described via series of calls to yet-undefined DEFTREECODE. The macro is defined on include site (in the compiler proper), and depending on its definition, it's possible to extract desired information about nodes. Thorough explanation is provided for each node type in a C comment.

The nodes are built via suite of build functions: build0 to build7, depending on a number of node children, or arity of the node being built. Each build function has the following signature:

```
tree buildX (enum tree_code code, tree type, ...);
```

[1] Which is an informal term, not used by GCC people as far as I know.

Here the code is a symbolic name of the node type, e.g. a MULT_EXPR for multiplication; type is a tree representing a result type of the node; the ellipsis represents children to attach to the node.

When building GENERIC tree, you have to be very careful with what you attach to which node. GCC is unfortunately not typed statically, and all checks are performed at runtime. Moreover few things are checked explicitly: when you screw up, GCC is likely to end up with ICE as some internal assertion fails. (Which is actually a good thing, because should the assertion not be there, GCC would probably end up emitting wrong code, or would fail with SIGSEGV.) There is little I can advise, generally just read tree.def, read [3], and look to other front ends, in this order. Often it is helpful to inspect the failing site. Surprisingly often it's possible to deduce what went wrong by looking at the assertion or comments nearby.

For debugging purposes, a function debug_tree comes in handy. It recursively dumps the tree passed in as argument, to the depth of 6, giving you a chance to inspect whether the node in hand makes any sense.

6.2 Variables, Types, Symbols

Symbols in general appear in two contexts: in a *declaration*, their existence is announced. In a *definition*, the symbol comes to an existence. In expressions, the symbol is *referenced*.

6.2.1 Declarations

The declaration vs. definition distinction is mostly a matter of programming language, GCC doesn't care about that. All that matters from GCC viewpoint is where is the symbol defined, i.e. which program block owns it, or whether it is global, and where should the memory be allocated—whether in ELF section (assuming ELF output) or on stack. This is described by what's called a declaration node.

Declaration is created with function called build_decl. The two arguments of the function are *identifier* node, which denotes what's the declaration *called*; and *type* node, with obvious meaning. An example will show how is the declaration handled in Algol front end:

```
tree symbol_var_ordinary_build_generic (symbol_t * sym, binder_context_t * ctx) {
    label_t const * lbl = symbol_label (sym);
    tree id = get_identifier (estr_cstr (label_id (lbl)));
    tree tt = type_build_generic (symbol_type (sym), ctx);
    return build_decl (VAR_DECL, id, tt);
}
```

The function get_identifier takes a name represented as ordinary char* string, and creates an identifier node with the same name. Identifier nodes with the same name are shared: there is only one identifier node ever made for any particular name.

The function type_build_generic is a function of the front end, and it does recursive dispatch. In the end it returns a node that represents the type of given symbol.

The two nodes thus obtained are then combined into the VAR_DECL node.

There are more types of declaration nodes than just VAR_DECL. Labels are represented with LABEL_DECL nodes, fields of structures with FIELD_DECL, types with TYPE_DECL, etc.

So you have the declaration in hand. Next you have to store it somewhere, so that a) it's possible to look up declaration when symbol references it in an expression; and b) GCC knows that it should allocate the memory for the declaration.

The solution to a) will depend on architecture of your front end. The approach used for Algol 60 is that each symbol allows an "extra" information to be stored. When the GENERIC declaration is built, it's immediately stored to the symbol. In expressions, variable references are stored in idref AST node. One attribute of this node, resolved during semantic analysis, is the symbol to which the identifier refers. So when the expression is translated, the compiler knows which symbol the variable references, and from that symbol it extracts the declaration:

```
tree expr_idref_build_generic (expression_t * self, binder_context_t * ctx) {
  symbol_t * sym = expr_symbol (self);
  tree decl = symbol_extra (sym);
  return decl;
}
```

For handling of b) part, GCC prescribes the mechanism: when creating a block, the chain of declarations at this block is one of the attributes. Furthermore, each declaration has a DECL_CONTEXT attribute, which points to the containing block of the variable, or is NULL_TREE for file-global variables.

Declarations with static storage (a.k.a static variables) are defined by having set the TREE_STATIC attribute.

6.2.2 Types

Most types that Algol 60 supports are primitive types, such as integers, real numbers, void, etc. GCC has these types predefined, so most of the type building procedures look like this:

```
  tree type_real_build_generic (type_t * self, binder_context_t * ctx) {
    return double_type_node;
  }
```

You have to take care of type building for non-trivial types, such as are structures and arrays. Algol 60 doesn't support structures, but it does support arrays. This is how the front end builds GENERIC for array type:

```
  tree type_array_build_generic (type_t * self, binder_context_t * ctx) {
    tree emtt = type_build_generic (type_host (self), data);
    boundspair_t * bp = t_array_bounds (self);
    tree highb = expr_build_generic (boundspair_hi (bp), ctx);
    tree lowb = expr_build_generic (boundspair_lo (bp), ctx);
    tree arrbdst = build_range_type (integer_type_node, lowb, highb);
    tree ret = build_array_type (emtt, arrbdst);
    return ret;
  }
```

The boundspair_t type holds a pair of expressions with dimensions the user has provided. The function build_range_type is GENRIC representation of the same. The build_array_type then builds whole array type. Note that this function works recursively: if the element type (emtt) happens to be an array, this function is invoked through the visitor dispatch in type_build_generic. Multidimensional arrays are thus composed as arrays of arrays, just like you would expect from well behaved tree language.

6.3 Expressions

There's not that much to say about expression nodes. They use directly the node-creation mechanism that was described earlier: each expression has a type, and zero or more operands.

6.3.1 Literals

Starting from the leaves, that is nodes with arity zero, ALGOL recognizes the following literals: integer numbers, real numbers, string literals, and variable references. I will leave variable references for next section, and only cover literals here.

Starting with integers, there are two basic function for creation of these, namely:

```
// in gcc/gcc/tree.h
tree build_int_cst (tree type, HOST_WIDE_INT);
tree build_int_cstu (tree type, unsigned HOST_WIDE_INT);
```

The type of integer node doesn't have to be int. For example character constants are also represented with integer node. The type has to match the constructor function: if you use _cstu variant, the type has to be unsigned. The size of type HOST_WIDE_INT depends on a *host* platform, but is guaranteed to be at least 32 bits.

GCC allows you to create an identity of given type, with function build_one_cst. The type of the "one constant" is the sole argument of this function. The indentity nodes are shared, i.e. several calls to this function with the same type may yield the same **tree** pointer.

Floating point numbers are a bit more complex to create, because each platform has its own idea of how many bits are used for exponent and mantissa. IEEE 754 has three categories of floating point numbers: single-precission 32-bit, double-precission 64-bit and double-extended-precission 80-bit. In addition some architectures support 128-bit floating point, e.g. ia64 [2], and some platform don't have to support IEEE floating points at all[2]. GCC has ways to deal with this diversity. It allows construction of floating point literals from string as well as from various representations, has dedicated functions for comparison and formatting them, and can also convert internal floating point number back to some target representation. Look up the gcc/gcc/real.h header if you are interested.

The ALGOL 60 front end uses the way of building the real number from string:

```
tree expr_real_build_generic (expression_t * self, binder_context_t * ctx) {
  REAL_VALUE_TYPE real;
  tree t = type_build_generic (expr_type (self), data);
  real_from_string3 (&real, estr_cstr (expr_real_value (self)), TYPE_MODE (t));
  if (REAL_VALUE_ISINF (real))
```

[2]Although I don't know if GCC is ported to any such platform.

```
        pedwarn ("floating constant exceeds range of %qT", t);
    tree ret = build_real (t, real);
    return ret;
}
```

GCC also supports complex numbers, via a node COMPLEX_CST, created with function build_complex. The arguments are simply a type, a real node, and an imaginary node.

String literals are created by the function build_string. The two arguments of this functions are the length of string literal, and pointer to the character array. This is how ALGOL front end does it:

```
tree expr_string_build_generic (expression_t * self, binder_context_t * ctx) {
    estring_t const * s = expr_string_value (self);
    int len = estr_length (s) + 1; // +1 for trailing zero
    tree ret = build_string_literal (len, estr_cstr (s));
    return ret;
}
```

(The "+1" business actually represents how deeply C-ish the ALGOL front end is. In particular: estr_cstr has to return NUL-terminated character array, and all functions in language runtime library have to assume NUL-terminated, C-like strings. E.g. Borland®-compatible Pascal front end would probably want to encode strings so that the length is stored in first byte, and the string would NOT be NUL-terminated. For GCC, string literal is just a bunch of bytes with given length.)

6.3.2 Other Expressions

Let's look at expressions themselves a bit. First to cover are symbol references. I already mentioned handling of symbols in a section 6.2.1. When used in expressions, the **tree** node that represents variable reference is a declaration itself—look at the function expr_idref_build_generic from example in section 6.2.1.

Simple expressions (binary and unary) are build in a straightforward manner. For example this is how ALGOL 60 front end does it:

```
tree private_expr_build_binary_generic (expression_t * self, binder_context_t * ctx, int op) {
    type_t * t = expr_type (self);
    tree ttt = type_build_generic (t, ctx);
    tree op1 = expr_build_generic (expr_binary_left (self), ctx);
    tree op2 = expr_build_generic (expr_binary_right (self), ctx);
    tree ret = build2 (op, ttt, op1, op2);
    return ret;
}
```

Unary expressions are built in similar fashion, and also conditional (ternary) expression. The argument op is set by the callee to the exact type of expression, e.g. MINUS_EXPR or EQ_EXPR. The only thing to keep in mind is typing of arguments. E.g. COND_EXPR breaks when other value than 0 or 1 is used as condition, and both branches have to have the same type as whole expression. This all is written in a comprehensive manner in gcc/gcc/tree.def, and also in [3].

Array access expression, or ARRAY_REF in GCC lingo, are similarly straightforward, with one glitch, which is multidimensional arrays. This is nothing you wouldn't expect: multidimensional array is simply array of arrays, and so the result of array access is another array, with fewer dimensions. Single multidimensional access is then tree of array accesses, each yielding result of the type one dimension "shorter" than the other. The code is a little bit too verbose, and if you are interested, look up function expr_subscript_build_generic in al60l-bind.c.

The last expression to cover is a function call. Just like at the variable reference, you need to look up symbolic declaration first. Then you have to build expressions for all arguments, and connect them via tree_cons operation to a list of expressions, in a backwards fashion, i.e. last argument first. The GCC function build_function_call_expr is then used to turn the decl and argument list into function call node. The example follows, edited to pseudo code to take up less space, and assuming that expr_call_args are stored in backwards manner:

```
tree expr_call_build_generic (expression_t * self, binder_context_t * ctx) {
   symbol_t * sym = expr_symbol (self);
   tree proc_decl = symbol_extra (sym);
   tree arg_list = NULL_TREE;
   for (expression_t * expr in expr_call_args (self))
      arg_list = tree_cons (NULL_TREE, expr_build_generic (expr, ctx), arg_list);
   return build_function_call_expr (proc_decl, arg_list);
}
```

6.3.3 Expressions Evaluated Once

In certain contexts, it is necessary to evaluate an expression, and use its value on several places. In a C-like language, you would use a temporary variable for that. GENERIC supports the notion of reused expression directly with SAVE_EXPR.

SAVE_EXPR node has a single operand, which is the actual expression to be computed. The resulting SAVE_EXPR can be used on several places. The expression it wraps around is ever computed only once, in further invocations it's replaced by a reference to temporary variable.

The Algol 60 front end uses SAVE_EXPR for assignment of one right hand side to several left hand sides, such as in this example:

```
V1 := V2 := EXPR;
```

6.3.4 Addresses and Dereferences

One more thing that Algol front end does is that it handles code such as this:

```
A[V] := V := EXPR;
```

The Algol standard specifies that in such a case, the array accesses are resolved before any assignment takes place. In other words, the newly-obtained value of V has no effect on how the A[V] is computed.

The front end does this by first taking address of each left hand side, and then assigning the left hand side expression to dereference of each of taken addresses. Two expression nodes

are in effect here: ADDR_EXPR for taking an address, and INDIRED_REF for dereference. Each of these has single operand, and they map, respectively, to & and * operators of C language.

The type of ADDR_EXPR node has to be a pointer. Given a type t, pointer to t is built with GCC function build_pointer_type(t).

6.4 Statements

From the viewpoint of GENERIC, statements are expressions, too. Their type is void_type, but otherwise they obey the same rules as nodes of "ordinary" expressions.

6.4.1 Blocks

One of major features of structured programming is the ability to group single statements into compound statements, in a recursive manner (statement inside block can itself be a block). GCC naturally supports the notion of blocks.

In context of GENERIC, compound statement is expressed with an expression node BIND_EXPR. Block expression has three operands: a chain of variables local to the block, a list of statements to process, and associated block.

First, the two sequences: the *chain* of variables, and *list* of statements. Each of them is created differently. Chaining is natural operation of GENERIC nodes, built in for exactly the reason to be able to form linked lists easily. To link two nodes together, use macro TREE_CHAIN:

```
void bind_add_decl (binder_context_t * ctx, tree decl) {
   TREE_CHAIN (decl) = ctx->cur_block_vars;
   ctx->cur_block_vars = decl;
}
```

Statement lists are different beasts. New statement lists is created with function alloc_stmt_list, and new statements are added via append_to_statement_list:

```
void bind_add_stmt (binder_context_t * ctx, tree stmt) {
   append_to_statement_list (stmt, &ctx->cur_block_stmts);
}
```

Now for the "block" part. BIND_EXPR is node that binds block together with variables and statements. Block is what expresses nesting structure of compound statements, and also has a reference to its variables.

Each block has its associated superblock, set with a macro BLOCK_SUPERCONTEXT, and a list of subblocks, assigned in creation time. You have to keep everything in sync: bind expressions, supercontexts and subblocks. If a bind expression A with block A_B appears as a statement inside a statement list B_S of bind expression B, then BLOCK_SUPERCONTEXT of A_B has to be B_B, and list of subblocks of B_B has to contain A_B.

Algol 60 front end has two functions for block handling: bind_state_push_block and bind_state_build_block. The former one opens new block context, and actually does very little work: only sets up initial (i.e. empty) values for list of statements, chain of variables, and list of subblocks. The latter one does all the interesting things: builds BIND_EXPR from collected variables, statements, and subblocks; sets supercontext of each of collected

subblocks; and adds itself to the list of subblocks of next open block. Both functions are available with minor edits in figure 6.1.

6.4.2 Loops

All loops in GENERIC are expressed with the same LOOP_EXPR node. Sole argument of this node is the body, which the LOOP_EXPR endlessly processes. Computations of loop control variables have to be done explicitly, as well as loop termination.

To end the loop, GENERIC uses a EXIT_EXPR node. It has one argument, which is a condition that has to be true for the break to be taken. For example, following ALGOL 60 abstract loop:

for i := E while F do BODY;

Would be translated to the following GENERIC:

```
LOOP_EXPR (void, BIND_EXPR({
   1: i := E;
   2: EXIT_EXPR (void, !F);
   3: BODY;
}))
```

The translation itself is lengthy, and not suitable for inclusion as an example.

6.4.3 Flow Control

Other flow control primitives include conditional statement, and jumps, labels and switches.

Conditional statement is done exactly like conditional expressions, except that the node type is void.

While ALGOL 60 front end handles labels as symbols with attached statement (where the label is defined), in GENERIC they have dual nature: a symbol part is described by LABEL_DECL; the location by LABEL_EXPR, which is unary node whose parameter is LABEL_DECL. Similarly **goto** is unary node, whose parameter is either a LABEL_DECL, or arbitrary ADDR_EXPR that denotes the target site. This latter is used for implementation of computed goto, and switch designators in ALGOL 60 front end.

Besides this, GENERIC has support for "true" C-like switches. The two interesting nodes in this respect are SWITCH_EXPR and CASE_LABEL_EXPR.

6.5 Debugging Information

6.5.1 Loci

Locus is a simple stack-allocated structure that denotes a location of certain program construct in source code. It is used for diagnostic message reporting, and for tracking locations for purposes of writing debugging information. The contents of locus are a *name of the file* and a *number of the line* inside that file, where given construct appeared. Thus its creation is straightforward, one just has to include the right headers. E.g. conversion between ALGOL 60 front end's *cursor* type and locus is done this way:

```
void bind_state_push_block (binder_context_t * ctx) {
  slist_pushfront (ctx->subblocks, NULL_TREE);
  slist_pushfront (ctx->vars, NULL_TREE);
  slist_pushfront (ctx->stmts, alloc_stmt_list ());
}

tree bind_state_build_block (binder_context_t * ctx) {
  // Collect the subblocks that might have been added during
  // translation of container statements.
  tree subblocks = slist_popfront (ctx->subblocks);
  if (subblocks)
    subblocks = nreverse (subblocks);
  tree vars = slist_popfront (ctx->vars);
  tree stmts = slist_popfront (ctx->stmts);

  // Create new block and make it a superblock of all subblocks.
  tree block = build_block (vars, subblocks, NULL_TREE, NULL_TREE);
  if (subblocks) {
      tree subblock_node;
      for (subblock_node = subblocks; subblock_node != NULL_TREE;
           subblock_node = TREE_CHAIN (subblock_node)) {
          tree subblock = TREE_VALUE (subblock_node);
          BLOCK_SUPERCONTEXT (subblock) = block;
      }
  }

  // Then add the block to super's subblocks.
  slist_it_t * it = slist_begin (ctx->subblocks);
  tree super_subs = slist_it_get (it);
  super_subs = tree_cons (NULL_TREE, block, super_subs);
  slist_it_put (it, super_subs);

  // Finally return the binding expression that represents our block.
  return build3 (BIND_EXPR, void_type_node,
                 BLOCK_VARS (block), stmts, block);
}
```

Figure 6.1: Block handling in ALGOL 60 front end.

```
#ifdef IN_GCC
# include "system.h"
# include "coretypes.h"
# include "limits.h"
# include "input.h"

void cursor_to_loc (cursor_t const * cursor, location_t * loc) {
    memset (loc, 0, sizeof (location_t));
    loc->file = cursor->filename;
    loc->line = cursor->line;
}
#endif
```

6.5.2 Emitting Debugging Information

"Debuginfo" is additional information stored in compiled binary (be it object or resulting executable), which allows debuggers map binary code to original source. GCC has a support for this kind of additional information: it is possible to annotate expressions and statements with locus. The interesting macro (for both cases) is SET_EXPR_LOCATION. This is how ALGOL 60 front end does it:

```
void private_set_location (tree node, cursor_t * cursor) {
    location_t loc;
    cursor_to_loc (cursor, &loc);
    SET_EXPR_LOCATION (node, loc);
}
```

Be warned: it doesn't make a sense for all node types to bear line information. In particular, constants are usually shared between expressions, and assigned locus mangles them.

Chapter 7

GCC Services

7.1 Preprocessing

GCC makes it possible to preprocess the source files with a traditional C preprocessor. The magic takes place in a language spec, which is defined in lang-specs.h file of the front end. I will describe the hack in a little unusual way: start by spoiling the result, and will explain how to get there and what it means. Have a look at a figure 7.1.

The spec prescribes that ordinary, pure ALGOL files have extension .a60 and .alg (i.e. lower case), while files to be preprocessed are .A60 and .ALG (upper case). This convention is used also in other cases, e.g. for assembly (.s vs. .S) and Fortran (.f90 vs. .F90), and because nobody really uses ALGOL 60, it was an arbitrary decision.

Spec is line-oriented language: is has one statement per line. The statements are just like normal shell command lines, with program at the beginning, followed by a list of options separated by white space, except that redirection of input or output is not supported. The special thing about spec lines is that %-prefixed parts are subject to additional processing.

Let's have a look at the non-preprocessing spec string. There is only one interesting %-rule used: %{!S:X}, which "substitutes X, if the -S switch was NOT given to CC". Obviously, if user entered -E, he wants to stop compilation after the preprocessing, which for .a60 file means do nothing. Similarly for -M and -MM. So the whole string reads something like "if neither of the options E, M and MM is given, launch compiler. After that, unless the option -fsyntax-only was given, invoke assembler."

Wait, how is it that the assembler is invoked on the same spec line as compiler? Actually, %(invoke_as) is a macro (defined in gcc.c) and expands to new line with invocation of assembler, unless there was a -S given.

The trick very similar to the one with assembler is also used for preprocessor. The file is first preprocessed—this takes place always (given a right file name), because preprocessing is the first step of compilation. *If* neither of the options E, M and MM is given, we redirect the output of preprocessor (-o %|.a60) and emit pipe and newline. ("The pipe symbol at the beginning of the predicate text is used to indicate that a command should be piped to the following command, but only if -pipe is specified.") This way we form either only one preprocessing command, or two commands: preprocessed output is then pushed into compiler. (And actually, if -S is not given, that is then pushed to assembler.)

The last important bit is how the stages communicate with each other, and how they know where to read the files from. That's where the %|.a60 bit comes in play: it's replaced either by "-" (which is traditionally treated as a standard input or output), if -pipe is in effect, or by a random name, replaced consistently in whole spec string.

```
/* This is the contribution to the 'default_compilers' array in gcc.c
   for the Algol 60 language.  */

{".A60",   "@a60-cpp-input", 0, 0, 0},
{".ALG",   "@a60-cpp-input", 0, 0, 0},
{"@a60-cpp-input",
  "cc1 -E -traditional-cpp -D_ALGOL60_ %(cpp_options) \
      %{E|M|MM:%(cpp_debug_options)}\
      %{!E:%{!MM:%{!M: -o %|.a60 |\n\
    algol601 -fpreprocessed %|.a60 %(cc1_options)
     %{!fsyntax-only:%(invoke_as)}}}}", 0, 0, 0},

{".a60",   "@a60", 0, 0, 0},
{".alg",   "@a60", 0, 0, 0},
{"@a60",   "%{!E:%{!MM:%{!M:algol601 %i %(cc1_options) %{I*}\
             %{!fsyntax-only:%(invoke_as)}}}}\n", 0, 0, 0},
```

Figure 7.1: lang-specs.h of Algol 60 front end

So you see, spec is one strange language (its definition is in gcc/gcc/gcc.c), but the magic that gives you the preprocessor launched is not that esoteric.

That's obviously not all, because preprocessed file is littered with line markers, such as this:

```
# 1 "x.A60"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "x.A60"
```

There are two options: either you disallow emitting of line markers by passing -P to cpp; or you will cope with them, which means will be able to parse them, and will keep line markers in mind when emitting error messages and debug info. Parsing is best done in lexical analyser. Given that the marker can appear between arbitrary two tokens it's the only place where it makes a sense.

You may want to disallow line markers in programs that didn't come from preprocessor. This is the case of .a60 files of Algol 60 front end, where the issue is resolved by introduction of command line option -fpreprocessed. Unless this option is given, the "#" character is considered invalid. Only when this option is given, further parsing of line marker is done. Note that -fpreprocessed is passed to the compiler by spec line.

7.2 Command Line Options

Each GCC compiler has the capability of processing command line options. Moreover it inherits all the options from the main part of GCC, so e.g. -O3, -o file and others are available for all front ends with no work. The only work is necessary for definition of options peculiar to given front end, and even there is the tedium of command line parsing left off your shoulders.

7.2.1 Processing Options

GCC understands both positive and negative variants of -f and -W options. E.g. when your front end supports -fdump-ast, GCC will understand also -fno-dump-ast. Furthermore, each option can be parametrized. Thus you can have e.g. --output-pch= option for output of precompiled headers, and the part after "=" is delivered to option handler as an argument.

Of course, you have to write the handler for frontend-specific options yourself. All the work takes place in LANG_HOOKS_HANDLE_OPTION hook, GCC calls this function each time it hits an option that the frontend understands. The communication isn't done through option strings, though. Instead, GCC associates each option a symbolic identifier with unique integer value. When option is handled, simple switch statement can be used to decide what should be done. Option strings are transformed to identifiers in a straightforward manner: each non-alphanumeric character in a string is replaced with underscore, and OPT is prepended before the resulting string. Thus e.g. --output-pch= is referred-to by identifier OPT__output_pch_.

Other parameters in option-handling hook are argument and value. Variable argument is either NULL, or it holds a string with the extra option argument (as was described few paragraphs up). The variable value is 1 if an option is used in its positive variant, and 0 for no- variant.

Option handling hook can return three values: 0 if the option was invalid, 1 is if was valid, and -1 if it was valid and no further processing of the option should be done.

7.2.2 Defining New Options

All front-end-specific options are defined in lang.opt. This file gives, through build magic, rise to options.h. Format and features of lang.opt are to be found in GCC internals documentation [3, chapter Options].

A warning is due when changing the lang.opt file. Almost whole GCC depends on options.h, directly or indirectly. Changing lang.opt will lead to almost whole bootstrap being processed again. Have a cup of tea ready before doing so.

7.2.3 Interesting Langhooks

Following langhooks are of interest when you are doing option processing:

1. LANG_HOOKS_INIT_OPTIONS is called before any option processing is done. You will probably want to initialize your flags to their default values here.

2. LANG_HOOKS_HANDLE_OPTION is called to handle a single command-line option.

3. LANG_HOOKS_MISSING_ARGUMENT is called when the argument is missing to the option that requires one. The hook is handed over an offending argument string and its unique identifier. If it answers false, default missing-argument complaint should be used. true means that you took care of the error message yourself. This hook doesn't have to be defined, it's bound to false-returning empty function by default.

4. LANG_HOOKS_POST_OPTIONS is called after all command-line processing has been done. This lang hook also is a convenient place to determine the name of the input file to parse.

7.3 Diagnostics

GCC uses its own functions for reporting problems. You may or may not use these in your compiler. My advise is do. First, if errors are not reported via GCC's mechanisms, GCC thinks compilation was successful, and proceeds with compilation. Second, a lot of development tools rely on the way GCC reports errors. Prime example is GCC's own test suite, you will have to provide means of converting from your format to what's expected by the test suite scripts. Third, GCC itself will use its own reporting tools nonetheless, e.g. for purposes of reporting internal errors.

That said, using GCC reporting is simple. For example figure 7.2 has the program excerpt that illustrates how ALGOL front end does error reporting.

7.3.1 Configuring Reports Via Command Line

In GCC front ends, warnings make up significant part of reported messages, and can be richly configured. Perhaps because C is very loose language, with almost everything allowed, even though lots of it are just historical crud that you use only inadvertently. Anyway, GCC allows a very fine-grained tuning of what to report, by way of various -W and -Wno- options.

Further, GCC error reporting primitive (report_diagnostic) allows an explicit setting of dependency on particular command line option. For example, look at how it's done in GCC's own warning function:

```
void
warning (int opt, const char *fmt, ...) {
   diagnostic_info diagnostic;
   va_list ap;

   va_start (ap, gmsgid);
   diagnostic_set_info (&diagnostic, fmt, &ap, input_location, DK_WARNING);
   diagnostic.option_index = opt;

   report_diagnostic (&diagnostic);
   va_end (ap);
}
```

The interesting line is rendered in bold, the rest of function should be familiar from figure 7.2. By setting the option_index, one explicitly states that given message should be reported only if the associated option is in effect.

7.3.2 Pedantic Settings

On GCC command line, one can use an option -pedantic to express the desire to process the program as closely to language standard, as possible. The use of -pedantic on command line has the effect of setting the global variable pedantic to non-zero value, thus allowing for conditions on this variable. Another option -pedantic-errors will turn these warnings into full errors.

To honor GCC in this respect, you should report pedantic warnings with GCC function pedwarn. This function will report a warning, unless -pedantic-errors was given on the command line, in which case it issues an error. An example from C front end:

```
#ifdef IN_GCC
// Use GCC-native error reporting when possible.  This is at the end
// of definitions, to keep us away from GCC memory management poisoning.
# include "config.h"
# include "system.h"
# include "coretypes.h"
# include "limits.h"
# include "toplev.h"
# include "diagnostic.h"

static int
private_log_printfc (logger_t * logger, debug_level_t level,
     cursor_t * cursor, char const * fmt, va_list * ap)
{
  // GCC diagnostics are not intended for debug messages and notes.
  // Fall back to our own reporting in such a case.
  if (level < ll_warning)
    return private_log_printfc_nogcc (logger, level, cursor, fmt, ap);

  // Table to convert from our own severity encoding to GCC's.
  static diagnostic_t gcc_diagnostic_kind_map[] = {
    [ll_warning] = DK_WARNING,
    [ll_error] = DK_ERROR,
    [ll_fatal_error] = DK_FATAL
  };

  diagnostic_info diagnostic;
  location_t loc;
  diagnostic_t gcc_level = gcc_diagnostic_kind_map[level];
  if (cursor != NULL)
    cursor_to_loc (cursor, &loc);
  else
    memset (&loc, 0, sizeof (loc));
  diagnostic_set_info (&diagnostic, fmt, ap, loc, gcc_level);
  report_diagnostic (&diagnostic);
  return 0;
}
#else
// ...
#endif
```

Figure 7.2: Error reporting via GCC tools

```
if (pedantic)
  pedwarn ("ISO C does not support %<++%> and %<--%>"
           " on complex types");
```

7.3.3 Other Reporting Functions

Apart from general reporting mechanism through report_diagnostic, GCC supports a lot of quick reporting function for various levels of severity. These include inform, warning, error, sorry, fatal_error and internal_error (in the order of increasing severity). Most of them have obvious meaning, sorry is used for reporting of unimplemented language features.

GCC assumes that you use its own global variable input_location (which is a locus of currently processed part of source code), and if you don't, these functions won't produce the output you expect.

You will probably want to use these functions only if your front end is written directly for GCC.

7.3.4 Formatting the Report Strings

GCC uses its own formatting functions, presumably for independence on the host libc environment. In the format string, it's possible to use traditional numerical tags (%d et al.), modified optionally by l or ll prefixes. In addition, prefix of w is available, expanding to either l or ll depending on width of type HOST_WIDE_INT. String formatting via various %s variants is also allowed, etc.

In addition to these more or less standard tags, GCC also supports native language quotes by way of %< and %> (as was seen in a pedwarn example above), and %H for formatting of location_t. This may come in very handy, although is seldom necessary, because report_diagnostic takes care of this automatically.

Have a look at a comment before function pp_base_format in pretty_print.c for full reference.

7.4 Runtime Libraries

Languages usually need a support beyond mere syntactic and semantic actions. This includes input and output routines, some cunning numerical algorithms for scientific languages, or simply language standard library, such as libstdc++ or java platform. Most probably you will have to roll one of these for your language.

Runtime support also includes system's libc and libm. To honor GCC interfaces, you will have to pay some attention to these, too.

GCC doesn't support integration of the runtime library with the same ease it supports new language front ends. There are files to be patched, an operation inherently unsafe in a volatile environment of GCC trunk. Apart from that, however, the automation works nicely, and with some rules in mind, you can build runtime library that is linked to your binaries by default. Let's get dirty!

7.4.1 Agenda

You need the following.

1. To write the library itself. This includes source files and build system.

2. To include the library into the build chain. This includes patching toplevel configure and Makefile.def.

3. To inject the library into compilation command line, so that it's linked in produced binaries.

7.4.2 The Library Subdirectory

Given a toplevel directory gcc, runtime libraries typically reside in subdirectory gcc/libsomething. This directory is basically ordinary library build directory: it is almost possible to simply copy preexisting library files there.

GCC expects that the library provides a configure script, which, when launched, creates Makefile. (Note that the created Makefile has to reside in a *build* directory, but the script is launched from a *source* directory.) I assume that you obey autotools, as it will save you a lot of work.

The best approach is probably simply to take other frontend's configure.ac and Makefile.am, look how is it done, and bend it to suit your needs. The files are mostly classical autotools source files, but there are certain hacks here and there necessary for integration into GCC build system. In particular:

- configure.ac has to call GCC_TOPLEV_SUBDIRS macro after AC_INIT. This relates to the dreaded GCC's build/host/target trichotomy. Each of builds is separated in directory of its own: build_subdir, host_subdir, and target_subdir. This macro determines these values.

- In Makefile.am, gcc_version has to be calculated for use in expansion of toolexeclibdir variable. This can be done by reading the contents of file gcc/gcc/BASE-VER.

- libtool version info (i.e. library soname) is not hardcoded into the Makefile, but rather is stored in a file called gcc/libsomething/libtool-version. This is not compulsory step, it just eases things a bit in that you don't have to rebuild autotools-generated files when you update library version.

Besides the concrete cases just mentioned, other front ends usually contain workarounds that were necessary to resolve problems in past. It's certainly not wise to copy blindly every hack, as many of them may be outdated and superfluous. But other implementers likely fallen into the same traps you did, and reinventing the wheel rarely pays off.

7.4.3 Patching Toplevel Build System

One last step is necessary before your library gets built as part of build process. GCC has to *know* about it. As it is, you have to touch GCC's privates to do it: you need to patch toplevel configure and Makefile.def. Fortunately, both patches are trivial.

There is a variable target_libraries in toplevel configure. As the name implies, what libraries get built is driven by the contents of this variable. You need to add the name of your library among the others.

Patching Makefile.def is a more creative task. This file is used in concert with Makefile.tpl by a tool called autogen to produce Makefile.in. Inside Makefile.def, it is possible to express

```
This patch is artificially created to have the smallest possible
dependence on patched source.
Index: configure
===================================================================
--- configure (revision 118066)
+++ configure (working copy)
@@ -916,2 +916,3 @@
 target_libraries="target-libiberty \
+ target-libga60 \
  target-libgloss \
Index: Makefile.def
===================================================================
--- Makefile.def (revision 118066)
+++ Makefile.def (working copy)
@@ -100,0 +100,1 @@
+target_modules = { module= libga60; };
```

Figure 7.3: Toplevel GCC Patch for Algol 60 Front End

various inter-library dependencies to define exact conditions when your library should be built, as well as whether the library should be installed, bootstrapped, or checked.

An example toplevel patch, taken from Algol 60 front end, is at figure 7.3. If you want to be extra independent on the exact layout of libraries in configure, you can craft e.g. a sed script instead of patch:

```
sed '/^target_libraries="/s/"/"target-libga60 \\\n\t\t/' configure
```

Personally I'd rather stick with the patch approach, because a bit of dependence is good: it helps make sure that the expectations hold. E.g. the sed script doesn't do any error detection, duplicate detection or reporting, all of which patch does.

Release Mode

When doing a patch for release, i.e. not for development in GCC trunk, it's better to rebuild the file Makefile.in after patching, and generate new patch from this. The user then does not have to install development tools, autogen in particular. Build system in release version of GCC will not change very much.

7.4.4 Linking the Binaries With Runtime Library

The method that GCC uses to decide which runtime libraries to link in is pretty much a hack: you will inject necessary libraries into the command line, before it's processed.

The work is done in lang_specific_driver. Very general command line tweaking can take place here. The function is given pointer to argc and argv variables, and is free to reorder, add or remove any of the arguments as seems fit. To support proper linking, you will want to detect the situation when your runtime library, libc, libm, or any other required library isn't at the command line yet (or is and shouldn't be), in which situation you will add or remove it from the vector and adjust argc accordingly. It's necessary to support native

GCC switches, such as -nostdlib and -nodefaultlibs, as well as your own switches that imply that no linking is to be done (e.g. -fsyntax-only or similar).

Unfortunately that implies processing command line arguments one at a time, skipping their arguments as necessary (think -o, -Xlinker). It's really not nice solution, but that's the way it works.

7.4.5 Depending on System Libraries

You may or may not want to depend on system libraries, such as libc or libm. Their handling is mostly the same as handling your own runtime library, except that you don't have to build them.

Since version 4.3.0, GCC uses libgmp to do its constant folding, which means that the library will be available at compiler runtime. Very often that means the library will also be available at binary run site. That doesn't hold universally, but at least libgmp was ported to all platforms where GCC has supported back ends. What this means for you is that you can dispatch to this library many compile-time numerical algorithms that would otherwise have to be hand-rolled or cut'n'pasted. As far as I know, this is the approach Fortran fronted is about to take.

7.5 Regression Test Suite

During the development of a tool as complex as compiler is, thorough test suite is an absolute necessity. The GCC project does contain such test suite. It's built on popular dejagnu test harness, and as such suits best traditional regression testing approach.

Unit testing technically could be done through dejagnu, but natural fit here is processing of output from given compiled binary. This works fine for compilers: the test suite contains source files to be processed by your compiler, annotated with commands that describe expected output of the compiler. Unit testing will be best done directly as part of build of each module.

7.5.1 Build System Adjustments

If you want your test suite to be run as part of generic make check performance, you have to append checking target for your language to the the variable in Make-lang.in as follows:

```
# List of targets that can use the generic check- rule.
lang_checks += check-algol60
```

check-algol60 is handled automatically by a pattern rule in build/gcc/Makefile, you don't have to provide that yourself.

If you want to include other means of testing, e.g. the test suite that is already written as part of your front end, do so by explicitly stating dependency of check. E.g look at this example:

```
internal-algol60-checks:
	$(MAKE) -C $(srcdir)/algol60/tests check
.PHONY check: internal-algol60-checks
```

7.5.2 Organization of GCC Test Suite

Test suite is located in the subdirectory gcc/gcc/testsuite (see 3.5 for explanation of directory notation) (this particular subdirectory will be referred to simply as testsuite/ in context of this section). Under this directory is located testsuite/lib/, which contains support files for test suite; and number of front end specific directories, such as testsuite/g++.dg/ or testsuite/algol60.dg/.

To run a testsuite for your front end, issue a line like this in build/ directory:

```
$ gmake -C gcc check-algol60
```

(You would run internal-algol60-checks target similarly, if it is present. Alternatively, simple make check will run all GCC tests, including yours.)

Going through each of the files that make up test suite really makes no sense. Setup of test suite scripts for your language is a convoluted process, and I recommend simply copying over test suite scripts from other front end (e.g. ALGOL 60, where I took a time to clean up files a bit to get rid of what is not used or looks like historical crud). I'll just explain what's the intent of each file.

First, testsuite/lib/algol60.exp support file. This is the more entangled of the two support files. It contains various initialization and tool-specific procedures, and also contains a very important procedure algol60_target_compile, which binds GCC-specific side of test suite to dejagnu back end.

Second, testsuite/lib/algol60-dg.exp support file. This file is included as a library by various test suites that you can have. ALGOL 60 only has one test suite, but that doesn't have to be the case. The important thing is, this file is *tool*-specific. This file abstracts away from GCC part of the suite, so that you only have to sync implementation on one place when porting test suites.

Third, the directory testsuite/algol60.dg/ itself. The name of this directory has to match the name of your checking target. I named my check-algol60, and directory is named accordingly. The directory contains a script testsuite/algol60.dg/dg.exp, which is test suite driver. Basically what it does is that it iterates through all the test cases matching glob *.a60, and runtest each of them.

7.5.3 Test Directives

By now we have developed a suite of scripts that checks that each test case passes, but what's inside the test cases themselves?

Each test case is just a source file written in a language of tested front end, which is ALGOL 60 in my case. The file contains dejagnu directives, a meta-information that describes what to do with the test (i.e. compile, run, ...), and what output to expect (particular error messages, output of test run, ...). The exact nature of the directives will be best seen on an example:

```
comment { dg-do compile } ;
begin
   integer a;
   real b;
   a := 4.5; comment { dg-error "type mismatch" } ;
   b := 'ahoy'; comment { dg-error "type mismatch" } ;
end;
```

The first directive specifies that the test should be compiled. The following two directives specify that the "type mismatch" error should appear on the line, on which the directive appears. The test fails should the error not appear, or should it be different error, and passes if the error message is matched.

There are many other directives, e.g. for capturing output of the test that was compiled and run, for marking the test case as expected to fail, to limit the test to certain architectures, etc. The full list is available in [3] [1].

The directive processing tool expects the diagnostic messages to be in certain format, namely the traditional file:line[:column]: severity: description. If your front end provides e.g. more verbose, or differently formatted messages, you have to re-wrap them to suit expectations. Have a look at testsuite/lib/gfortran-dg.exp if you are interested in that.

[1] http://gcc.gnu.org/onlinedocs/gccint/Test-Directives.html

Chapter 8

Conclusions

1. A minimal front end and its integration into GCC build system is described in this work. All files that the front end is comprised of are thoroughly examined, and pitfalls mentioned. Several tips are provided on connecting such minimal front end with preexisting language parser.

2. An implementation of Algol 60 compiler is described, including the way various Algol constructs are translated to GCC's GENERIC intermediate language.

3. Interfaces and services of GCC itself, such as parsing command line arguments, compilation of runtime library, etc., are also described, with real world examples taken from the Algol 60 front end.

4. The conclusion: GCC should be considered as a viable alternative to other back end solutions, especially as an alternative to emitting a C code. In general, GCC is very attractive target for a lot of compiler-related research. It's open development cycle makes up for great test bed: be it a new radical systems language, be it new optimization technique, new hardware platform in development, GCC should be in the loop.

8.1 The Summary of Contributions

This thesis is synthetic in nature. Information were taken from various sources, be it other work of documentation, spoken word, a question over an email, or investigation of source code. This very synthesis is sole contribution of the author: no other work exists, that provides continuous explanation of this detail on the topic.

8.2 Future Research

This thesis describes odds and ends of the integration of the parser into GCC. There was a lot of ground to cover, GCC is big software project with a lot of facets. Some topics were left insufficiently explained, and certainly not everything was at least touched. For example, GCC has native support for OpenMP, a parallel language, that is about to become *very* interesting in years to come; it can handle classes, and object oriented programming in general, an area that already *is* very interesting; it allows for direct inclusion of inline assembly. This thesis describes none of these topics.

Bibliography

[1] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language algol 60. *Commun. ACM*, 6(1):1–17, 1963.

[2] Intel Corporation. Intel®itanium®architecture software developer's manual.

[3] Free Software Foundation. Gcc internals manual. This electronic document is available online at `http://gcc.gnu.org/onlinedocs/gccint/`.

[4] Free Software Foundation. Gcc manual. This electronic document is available online at `http://gcc.gnu.org/onlinedocs/gcc/`.

[5] Jason Merill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–180, May 2003.

[6] Diego Novillo. Gcc–an architectural overview, current status, and future directions. In *Proceedings of the 2006 Linux Symposium, Volume Two*, pages 185–200, 2006.

[7] Eric S. Raymond. *The Art of UNIX Programming.* Pearson Education, 2003.

[8] Tom Tromey. Writing a gcc front end. *Linux J.*, 2005(133):5, 2005.

Appendix: The Algol 60 Compiler

Requirements

The Algol 60 GCC front end has following build requirements:

- GCC 4.2.x with resolved dependencies. GCC 4.1.x or earlier will NOT work[1]. You will need either complete GCC package (e.g. gcc-4.2.0.tar.bz2), or core package and test suite (e.g. gcc-core-4.2.0.tar.bz2 and gcc-testsuite-4.2.0.tar.bz2).
- Flex 2.5.31 or newer. Warning: this is a newer version than what GCC requires.
- Bison 2.3 or newer.
- libga60 wraps around target-native libc and libm, which have to be available on target site.

Supported Arches

As of this writing, gcc-algol was successfully compiled and test suite passed on systems with following triplets:

- i686-pc-linux-gnu
- ia64-unknwon-linux-gnu
- mips-sgi-irix6.5

This is quite diverse set of arches: little and big endians, 32 and 64 bits, two unrelated operating systems, though no really obscure system. This illustrates that the front end is probably written in enough hardware and OS independent way to be readily portable.

Build Process

Only a straight build (i.e. no Canadian Crosses) is supported at the moment. (Or at least tested. Chances are the compiler will work also for non-native setups.) First, prepare a source tree:

[1] gcc-algol uses few functions that were only introduced in GCC 4.2.x. Their number is not high, and the dependence is certainly for no fundamental reason. My guess is gcc-algol could be ported to GCC 4.1.x with a bit of effort.

```
$ ls
gcc-4.2.0.tar.bz2 gcc-algol60-0.2.tar.bz2
$ tar xjf gcc-4.2.0.tar.bz2
$ pushd gcc-4.2.0
$ tar xjf ../gcc-algol60-0.3.tar.bz2
$ patch -p1 < gcc-4.2.0-toplevel.patch
patching file configure
patching file Makefile.in
patching file Makefile.def
$ popd
```

Then prepare for the build:

```
$ mkdir build-4.2.0
$ cd build-4.2.0
$ ../gcc-4.2.0/configure --prefix=`pwd`/../inst-4.2.0/ \
                         --enable-version-specific-runtime-libs \
                         --enable-languages=c,algol60 \
                         --disable-werror
```

The --disable-werror part is unfortunate, but necessary: the parser and lexer .c files, generated from yacc and flex respectively, currently generate a lot of warnings.

Finally, do a build itself:

```
$ gmake -j 4
...lots of output...
gmake[1]: Leaving directory '/tmp/build-4.2.0'
```

(The -j constant determines how many parallel processes make should launch. You will want to adjust this value depending on number of CPUs or cores of your computer.)

You can test GCC as a whole with gmake check, or just ALGOL 60 front end with gmake -C gcc check-algol60. If you so wish, you can install the front end with gmake install, it will end up in configured --prefix directory.

Using gcc-algol

From now on, you can use it as an ordinary GCC command[2]:

```
$ which ga60-4.2.0
/tmp/inst-4.2.0/bin/ga60-4.2.0
$ ga60-4.2.0 –version | head -n 1
ga60-4.2.0 (GCC) 4.2.0
$ cat foo.a60
'begin' puts('Yay, it works!'); 'end';
$ ga60-4.2.0 foo.a60 -o x
$ ./x
Yay, it works!
```

[2]Actually, in the case as illustrated, the compiler would likely end up in a directory outside PATH and LD_LIBRARY_PATH. You may need to adjust these variables.

```
#if !defined(_ALGOL60_) || !defined(USK)
# error That does not compute.
#else
'begin'
        /* funky C-like comment */
#       include "y.ALG"
'end';
#endif
```

Figure 8.1: Preprocessing: Example file x.ALG

```
'string' a;
```

Figure 8.2: Preprocessing: Example file y.ALG

Let's have a look at the compiled binary:

```
$ readelf -a x | grep NEEDED
0x0000000000000001 (NEEDED) Shared library: [libga60.so.1]
0x0000000000000001 (NEEDED) Shared library: [libm.so.6.1]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6.1]
```

The binary itself depends on libm and libc. These dependencies are injected into the binary by the linking hook lang_specific_driver. If we link the binary with gcc driver, our hook will not be called, and only libc will be linked in:

```
$ gcc-4.2.0 x.a60 -o x -lga60
$ readelf -a x | grep NEEDED
0x0000000000000001 (NEEDED) Shared library: [libga60.so.1]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6.1]
```

Under current setup, where all ALGOL 60 services are provided by *shared* libga60, explicit dependence on libm is unnecessary, and even wrong—libga60 itself should have these dependencies (and has!). However, libga60 doesn't have to be shared, and in that case libm has to be brought in explicitly.

gcc-algol uses a preprocessor, and understands traditional preprocessing directives. The preprocessor is not run by default, only when the file extension is .A60 or .ALG (i.e. upper case). For example, let's have the two files as shown on figures 8.1 and 8.2.

```
$ ga60-4.2.0 ./x.ALG
./x.ALG:2: error: #error That does not compute.
$ ga60-4.2.0 -DUSK ./x.ALG
./y.ALG:1: error: type 'string' is invalid in this context.
$ ga60-4.2.0 -DUSK -M ./x.ALG
x.o: x.ALG y.ALG
$ ga60-4.2.0 -DUSK -E ./x.ALG
```

```
# 1 "./x.ALG"
# 1 "<built-in>"
... more preprocessed output ...
```

The errors at the first and second commands are intentional. The first is here to display that it's possible to pass in your own defines. The second shows that gcc-algol correctly tracks files and line numbers: even though the erroneous line appears as twelfth line of preprocessed output, error message is located properly.

Finally, let's look that the binary supports debugging:

```
$ cat foo.a60
'begin'
   puts ('ahoy');
   exit (9);
'end'
$ ga60-4.2.0 foo.a60 -o x -ggdb3
$ gdb -q ./x
(gdb) break main
Breakpoint 1 at 0x804846a: file foo.a60, line 2.
(gdb) run
Starting program: /tmp/x
Breakpoint 1, main () at foo.a60:2
2    puts ('ahoy');
(gdb) cont
Continuing.
ahoy
Program exited with code 010.
```