Bachelor Thesis

Design of a Flexible Compiler

Benjamin Weber

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

October 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
Laboratory for Software Technology

Contents

1 Introduction

Compiler design is well researched at this point. A compiler can be divided into multiple stages (i.e. parsing, semantic checking, transformations and code generation). Such an approach is robust and works well in practice. Nevertheless, writing a compiler is still a tedious task. After all, a compiler has to cover all corner cases of a programming language. Furthermore, without sophisticated optimization transformations the produced executables can barely compete with executables produced by existing compilers such as clang, GCC's compilers or similar. Therefore, there is a strong interest in being able to reuse code of existing compilers.

This is also done in practice in various forms. For example one approach is to compile a programming language into an existing programming language and then use another compiler to produce the executable from the existing programming language. This approach is often used in practice where C is mostly chosen as the target language because C can express low level code well.

Another approach is taken by GCC (GNU Compiler Collection)[2] which can compile several languages into several assembly instruction sets. To achieve this one can create a front-end for GCC that compiles the program into an intermediate representation (IR) which GCC understands. Then, GCC can use all of its optimization transformations on this IR and finally GCC will produce the executable from the optimized IR. The front-end then usually consists of a parser, the semantic checks and a code generator that produces the IR of GCC. A small compiler itself.

LLVM (Low Level Virtual Machine)[3] is a project that provides a compiler infrastructure that allows others to reuse its components to easily write new compilers. LLVM is built around a well defined IR (the LLVM IR) so that existing transformations have a well defined set of instructions to work with. Here one would also write the front-end to benefit from LLVM's optimizations and code generators.

Both the Java Virtual Machine (JVM)[4] and Microsoft's Common Language Runtime (CLR) [1] are virtual machines that execute a well defined IR. Many languages target these IRs so that they benefit from the sophisticated runtime optimizations done by these virtual machines. As with GCC and LLVM one would have to write a small compiler to make use of those virtual machines.

2 Motivation & Problem Statement

While the approaches discussed in the introduction allow for code reuse in compilers they also suffer from some drawbacks on which I want to elaborate in the following:

Customization is done through different front-ends which means that a developer has little control over what happens afterwards.

Next, the IR is static. Consequentially, information could get lost while expressing the original program in the IR. Moreover, the IR may not be very suitable to express programs of the original programming language.

Furthermore, concepts shared by many languages still have to be implemented in each front-end separately. Such concepts would be variables, types, functions, basic instructions, control-flow elements and many more.

If we look at a **while** loop for example, then its syntax must be defined at the parsing stage and during semantic checking some checks need to be carried out (depending on the language). However, each front-end usually implements this separately, even though a **while** loop is very similar in each language.

In this bachelor thesis I want to explore an approach that does not suffer from these drawbacks. I want to design a compiler such that each concept of a potential programming language is separate from all other concepts. E.g. the code responsible to make variables work should be separate from the code responsible to make types work. This way it will hopefully be possible to combine these concepts flexibly to create new programming languages.

More specifically, programming language concepts should be implemented in separate modules such that each offers reasonable configuration possibilities. Then one should be able to combine these modules flexibly to create compilers for new programming languages.

3 Implementation

In this section, I will give an overview of the software and highlight and discuss some interesting aspects of the implementation.

3.1 Compiler Generation

For this prototype I have identified two different approaches. The first is to create a prototype that generates a complete compiler (including its source code) which then can be compiled into an executable. This would lead to a workflow where one would first use the prototype to create a compiler and then use the generated compiler to compile programs. This compiler would be completely independent of the prototype.

Secondly, one could create a prototype that reads the configuration of each module and how they are composed, loads those modules and configures them appropriately such that they can then be used to compile programs. In that case, the generated compiler would depend on the prototype and would be created each time someone wants to compile a program. In a way the generated compiler exists at runtime of the prototype, but not as standalone software.

I have chosen the first approach, because the generated compiler is independent, its source can be further adjusted if needed and it should have less overhead as the configuration is implicitly embedded into the generated compiler (this is in contrast to the second approach where a small overhead might be introduced, because the generated compiler has to take the configurations into account). Furthermore, this works well when using a parser generator which often yields source code for the parser.

3.2 Overview

A design that has proven itself robust is to divide the compiler into several stages i.e. parsing, semantic checking, transformations and finally code generation. While I explore a different design that separates concepts, I would like to also benefit from this design. Therefor, the generated compiler also follows this multi-stage approach.

The prototype is divided into two distinct parts i.e. the modules that contain language features (from here on simply features) and a framework that makes the features work together.

A feature is a module that describes a programming language concept. For example the concepts of variables or functions could be implemented as features and that is indeed exactly what I have

Listing 3.1: The interface of a feature

```
public class Feature {
        public String getName() {
                return this.getClass().getSimpleName();
        }
        public void setup(FeatureHelper helper) {}
        public void setupFeatureArrangements(
                ServiceProvider provider) {}
        public void setupCompilerGenerator(CompilerGenerator cg) {}

}
```

done.

The framework glues together the features in a sense. It loads a configuration that defines which features should be included and how they should be configured to generate a new compiler. It also defines how features can interact with each other and provides some basic elements required by all generated compilers.

3.3 Compiler Setup

Language features can have complicated relationships amongst each other. For example a **while** loop requires a boolean expression and a loop body that can be executed or accessing an object's field (**some_object.some_field**) involves features like expressions, classes and variables.

There are many options to model these relationships. I chose a procedure that consists of three steps to keep it simple and within the scope of a bachelor thesis. The three steps are built around a model of services i.e. a feature can provide a service that other features can use. A service is simply an instance of a class that implements the **Service** interface.

These three steps are reflected in the Feature class (see listing 3.1).

Feature setup: The first step consist of simply setting up the feature. The framework provides the feature with a configuration given by the user. The feature can check if the configuration is valid and take actions accordingly. The feature can declare dependencies on other features. Finally, the feature can register services with the framework that other features can then use (see the next step).

A feature must be able to do this step completely autonomously i.e. without the need of another feature.

Feature arrangements: In this step the features can communicate through services provided in the first step. Such services can look very differently depending on what they need to achieve.

Compiler generation: In the last step the compiler is set up i.e. the source code of every file of the generated compiler has to be provided, the grammar for the parser generator will be determined, the order of the semantic checks is given and a code generator may be provided.

This procedure is simple and worked well for the implemented prototype. Nevertheless, I would like to mention one major drawback to this simple system.

Communication through services enforces a consumer-provider relationship in the sense that the features using the service are consumers and the feature providing the service is the provider. However, this model does not work well when several features need to be involved and none of them is a clear provider and the others are consumers.

This is the case when it comes to variables as class members which involves the expressions, variables and classes features. This relationship does not fit into the consumer-provider relationship well.

3.4 IR Extensions

Because this bachelor thesis focuses on the feasibility of the introduced approach, an abstract syntax tree (AST) is used throughout the whole compilation (this is in contrast to more sophisticated compilers that may choose to use other intermediate representations that lend themselves better for optimizations e.g. single static assignment form).

Moreover, the AST used offers a system of extensions in the sense that every node can be flexibly extended with objects (see listing 3.2 for the implementation). This is often used in various forms by the features.

E.g. the variables feature can extend nodes with a **VariableScope** extension that contains information about variables that are declared in the given scope. So, if a **for** loop introduces new variables into the scope, its AST node can be extended to contain that information. However, if no variables are introduced by that loop, its node does not need to be extended.

Or the x86 assembly back-end feature can extend function nodes to mark them as the entry-point. This is useful, because their names should not be changed so that the assembler can generate the executable correctly. Whereas other function names can be changed.

3.5 Code Generator

One drawback of a flexible IR is that code generation becomes harder. With a static, well defined IR there will be a fixed set of IR elements. A correct code generator has to be able to handle all IR elements. This has even more relevance when one wants to implement several code generators for different targets (such as x86 assembly, arm assembly, mips assembly) which is often done in practice.

That is, for every new IR element one would have to adjust each code generator which could quickly become very tedious.

I have identified two useful strategies to deal with this problem. One could define a reasonable set of IR elements which every code generator has to support. With such a basis, one can either rewrite

Listing 3.2: Code of the extensions system

```
public interface ASTNode {
        public List<ASTNode> getChildren();
        public void extend(Object extension);
        public <T> T extension(Class<T> type);
}

public abstract class ExtendableNode implements ASTNode {

        private Map<Class<?>, Object> extensions =
            new HashMap<Class<?>, Object>();

        @Override
        public void extend(Object extension) {
                assert !extensions.containsKey(
                              extension.getClass());

                extensions.put(extension.getClass(), extension);
        }

        @SuppressWarnings("unchecked")
        @Override
        public <T> T extension(Class<T> type) {
                return (T) extensions.get(type);
        }

}
```

new IR elements into the set of supported IR elements or provide extensions for code generators for unsupported IR elements.

I have kept the code generation within one feature i.e. the X86 feature which generates x86 assembly code. To gain further insights into the previously mentioned problem, I have designed the code generation such that the code to handle each IR element could be easily refactored out of the X86 feature into the features that introduce those IR elements.

To achieve this there is a static function (i.e. in the sense of Java static methods) for each IR element. Then for each configuration there is a core class generated that calls the appropriate static functions for each IR element.

Due to this design most required flexibility is contained in this core class and handling of each IR element can simply be done in static functions (which could easily be provided by other features).

Furthermore, the generated core class allows flexible preparations before code generation. This is useful to e.g. create vtables for classes or determine the stack frame layout for local variables (which works particularly well with extensions).

This design worked very well and suggests that there are good solutions to the mentioned drawback of a flexible IR.

The code for generating the core class lies in the **features/lfocc/features/x86/X86.java** file. The preparations and static code generation functions are in the **features/lfocc/features/x86/backend/preparation/** folder and **features/lfocc/features/x86/backend/generators/** folder resp.

4 Evaluation

In this section, I will first present the results achieved by this simplified implemented prototype and then focus on whether such an approach is feasible especially on a larger scale.

4.1 Results

In this bachelor thesis, I have implemented a total of 12 language features which I introduce in the following:

Base: The Base feature contains a lot of functionality used by many other features. E.g. it defines commonly used tokens for parsing such as identifiers, different brackets, comments, whitespaces etc. Furthermore it takes care of some special cases of feature combinations.

GlobalScope: Its purpose is mainly limited to parsing. It defines the grammar rules to be applied at the root of the file (e.g. global variable declarations, global function declarations or class declarations).

Types: This is again a very simple feature which provides a grammar rule to use for types which can be extended by other features (via a service). Furthermore, it defines a **TypeSymbol** class which can be used as the base for other types. Lastly, it provides a **TypeDB** class where each type can be registered.

This **TypeDB** class is not suitable for structural types, but this did not pose a problem in the limited scope of this bachelor thesis (only nominal typing is used). Support for structural types would probably require a different solution.

Expressions: This feature takes care of expressions. It defines some fundamental arithmetic, comparisons and boolean logic operators as well as their types. It also allows other features to add their own expressions (this is relevant for function calls, class casts, class members etc).

Additionally, it distinguishes between assignable expressions and normal expressions at the parsing stage (i.e. there is a grammar rule for each). Thus, checking whether the target of an assignment is valid is done at the syntactical level.

Most of the operators defined as part of this feature could be moved into separate features. But these operators are mainly only relevant for expressions, that is why it worked well to keep them in this feature and simplified the implementation.

Listing 4.1: Basic input-output functions that are introduced by the functions feature

```
// the read functions read from stdin
int read() { /* ... */ }
float read() { /* ... */ }

// the write functions write to stdout
void write(int x) { /* ... */ }
void writef(float x) { /* ... */ }
// writes a new line
void writeln() { /* ... */ }
```

Variables: The variables feature introduces variables to languages. It can be configured to support global variables, function parameters, function local variables and variables as class members.

In the x86 code generator feature separate functions to get the address of a variable are provided so that variables' values can be overwritten.

Functions: With this feature functions can be added to languages. It supports global functions and functions as class members. Both can be flexibly configured. Additionally, it can be configured whether functions can have return values or not.

To add support for basic input-output functionality this feature also introduces **read**, **write** etc functions (see listing 4.1).

Statements: This is another feature which is only relevant for the parsing stage. It defines what a statement is. The distinction between a block of code and a statement is mostly relevant for **for** loops where a statement can occur to be executed at the beginning of the loop. Examples of a statement are variable declarations, function calls and assignments.

Assignments: The assignment feature defines assignments in programming languages. It makes use of the assignable expression grammar rule from the expressions feature.

CodeBlock: This feature is very light in the sense that it is only relevant for the parsing stage. It defines what a block of code is which is used by function declarations and control flow elements such as **if** clauses and various loops.

ControlFlow: With this feature control flow elements can be added to programming languages through its configuration. The supported control flow elements are **if**, **else** and **else if** clauses as well as **for**, **while** and **do while** loops.

Classes: The classes feature allows a user to add classes to a programming language. It can be configured to support single inheritance or no inheritance at all (similar to C's **struct**).

It is a fairly big feature as it introduces many elements such as a **null** reference, a **this** reference, a **new** operator, class casts, class types and class declarations.

On its own it is not a very useful feature. Only in combination with variables or functions as class members can it heavily influence a programming language.

X86: Finally, this feature is a back-end which generates x86 assembly code from which an executable is then assembled. Its design is described in section 3.5.

With these features a plethora of programming languages can be created through different combinations. However, not all combinations are possible due to dependencies (e.g. a **while** loop requires a boolean expression). Moreover, many combinations are not reasonable. A programming language that does not have expressions would be barely useful. Thus, during implementation reasonable assumptions (such as that the expressions feature is always enabled) were made.

Nevertheless, there are still many combinations possible. In the following I will introduce six sample languages which are quite simple, but should show that the prototype can already create quite different programming languages.

Furthermore, they are accompanied by a sample program to compute the nth Fibonacci number in linear time.

Listing 4.2: Fibonacci implementation in C

```c
int fib(int n) {

   if (n <= 0) {
      return -1;
   } else if (n <= 1) {
      return 0;
   } else if (n == 2) {
      return 1;
   }

   n = n - 2;

   int temp, f1 = 1, f2 = 0;

   for (; n > 0; n = n - 1) {
      temp = f1;
      f1 = f1 + f2;
      f2 = temp;
   }

   return f1;
}

void main() {
   write(fib(read()));
   writeln();
}
```

C This language mimics C as much as possible. Classes are enabled, but do not support methods or any inheritance and should model C's **struct**.

Listing 4.3: Cpp version of a Fibonacci implementation

```
class Fib {
   int fib(int n) {

      if (n <= 0) {
         return -1;
      } else if (n <= 1) {
         return 0;
      } else if (n == 2) {
         return 1;
      }

      n = n - 2;

      int temp, f1 = 1, f2 = 0;

      for (; n > 0; n = n - 1) {
         temp = f1;
         f1 = f1 + f2;
         f2 = temp;
      }

      return f1;
   }
}

void main() {
   write(new Fib().fib(read()));
   writeln();
}
```

Cpp This language is a superset of the previous C-like language as it is as close to C++ as possible. It is special as all implemented features are activated and their configurations are the least restrictive. Thus, all other languages are a subset of Cpp.

Listing 4.4: Fibonacci computation in Java

```
class Main {

   int fib(int n) {

      if (n <= 0) {
         return -1;
      } else if (n <= 1) {
         return 0;
      } else if (n == 2) {
         return 1;
      }

      n = n - 2;

      int temp, f1 = 1, f2 = 0;

      for (; n > 0; n = n - 1) {
         temp = f1;
         f1 = f1 + f2;
         f2 = temp;
      }

      return f1;
   }

   void main() {
      write(fib(read()));
      writeln();
   }
}
```

Java As the name suggests, this language is a Java clone. It does not support global functions or global variables.

Listing 4.5: Fibonacci implementation using the Functional language

```
int fib(int n) {
   if (n <= 0) {
      return -1;
   } else if (n == 1) {
      return 0;
   } else if (n == 2) {
      return 1;
   } else {
      return _fib(0, 1, n);
   }
}

int _fib(int fib1, int fib2, int n) {
   if (n == 2) {
      return fib2;
   } else {
      return _fib(fib2, fib1 + fib2, n - 1);
   }
}

void main() {
   write(fib(read()));
   writeln();
}
```

Functional A language that follows the functional paradigm by disabling assignments, classes, loops and only allowing variables as function parameters.

Listing 4.6: PureState's Fibonacci version

```
int ref, n, f1, f2, temp, ret;

void fib() {
   if (n <= 0) {
      ret = -1;
   } else if (n <= 1) {
      ret = 0;
   } else if (n == 2) {
      ret = 1;
   }
   n = n - 2;
   f1 = 1; f2 = 0;
   for (; n > 0; n = n - 1) {
      temp = f1;
      f1 = f1 + f2;
      f2 = temp;
   }
   ret = f1;
}

void main() {
   n = read();
   ref = read();
   fib();

   if (ret != ref) {
      // this will crash the program
      // denoting failure to compute the correct value
      ret = 1/0;
   }
}
```

PureState This language is quite the opposite of the Functional language as it only allows global state. It has no classes and allows only global variables. Functions have no parameters, local variables or return values (Because functions cannot have parameters the output functions **write** and **writef** are unfortunately not usable).

Listing 4.7: Typeless' Fibonacci computation

```
void fib(n) {

   if (n <= 0) {
      return -1;
   } else if (n <= 1) {
      return 0;
   } else if (n == 2) {
      return 1;
   }

   n = n - 2;

   temp, f1, f2;

   f1 = 1;
   f2 = 0;

   for (; n > 0; n = n - 1) {
      temp = f1;
      f1 = f1 + f2;
      f2 = temp;
   }

   return f1;
}

void main() {
   write(fib(read()));
   writeln();
}
```

Typeless A fairly simple language with dynamic typing and without classes. No code generator supports this language. However, the parsing and semantic checks are implemented.

4.2 Feasibility

In this section, I want to discuss how feasible such an approach is on a bigger scale than a simple prototype.

One can easily notice that many stages of a compiler follow a very recursive structure. The grammar defining the syntax of a language reuses grammar rules heavily.

For example, a function declarations has a block of code which in turn can contain assignments and those can again contain expressions. Expressions are even more recursive in the sense that all operators work on expressions and simultaneously introduce operator expressions. Recursion then occurs naturally in concrete parse trees.

Abstract syntax trees are often very similar to parse trees and also heavily contain recursion. Their structure is determined by the class hierarchy of the AST nodes and their fields. Particularly the fields of AST nodes are another example of recursion. E.g. a binary operator expression node contains a left and right field which are both expressions again. Trivially, there is a correspondence between what the grammar is to a parse tree and what the definitions of the AST nodes are to a concrete abstract syntax tree, as the later is always an instance of the former. Furthermore, many AST nodes directly correspond to grammar rules.

The implemented code generator transforming the AST to x86 assembly works recursively as well (as explained in section 3.5). In fact, it follows the same recursive structure as the AST nodes. This is again quite obvious, as the code generator directly works with the AST and thus the code generator's functions correspond to the different types of the AST nodes.

This also holds for AST visitors in general, because the code generator is an AST visitor itself. So the correspondence also extends to the semantic stage which is often done with AST visitors in practice (this bachelor thesis uses AST visitors for the semantic stage too).

Therefore, one can see that there is a direct correspondence between the same language concepts at different stages of the compilation.

This observation suggests the viability of the proposed approach and was also a driving factor to pursue this particular bachelor thesis.

A more pessimistic picture is painted by considering the potential coupling between language features as modules. Coupling between language features can occur if one language feature needs to be aware of the presence of another language feature. An example would be that in the current implementation the variables feature needs to be aware of the presence of the assignments feature to allow variable declarations of variables which are also initialized with an assignment (e.g. `int a = 42;`).

Intuitively, it seems that language features are often highly connected with each other which suggests that there would be a high coupling between language features as modules. However, this is seen as problematic in software engineering and it is desirable to keep the coupling between modules low. Which suggests that this approach is impractical.

However, it is noteworthy that I have made the experience that adding a feature can also ease the implementation of other features which is counter to the previous point made. This occurred when adding the types feature. Originally, I intended not to implement a types feature and put the necessary functionality in the variables feature which caused problems as types were connected to many other features such as expressions, classes, functions, control flow and others.

I also considered adding a scopes feature. Scopes are relevant for all kinds of symbols such as variables, functions or types. In the current implementation the concept of symbol scopes is contained separately in both the variables and functions feature.

It appears that choosing the right abstractions for language concepts leads to a better design and makes implementing other features easier for this approach.

5 Conclusion

In this bachelor thesis I explored a different approach to compiler design than the traditional separation into stages. This was motivated by improving code reuse, simplifying compiler construction and the observation that traditional compiler design already suggests the viability of such an approach. I implemented a prototype which can create compilers for many simplified yet different languages.

The created prototype strongly suggests that this approach is practical as no major obstacles occurred during its implementation. However, this result has its limitations when looking at the bigger question of whether the approach is viable on a larger scale due to the simplifications made. Yet, it is still a very positive indicator.

Thus, I believe the proposed approach is attractive for compiler design.

Bibliography

[1] CLR, microsoft's commong language runtime. https://msdn.microsoft.com/en-us/library/8bs2ecf4.

[2] GCC, gnu compiler collection. https://gcc.gnu.org/.

[3] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO04)*, 2004.

[4] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition.* Addison-Wesley Professional, 1st edition, 2013.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Design of a Flexible Compiler

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
WEBER	BENJAMIN

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Ebikon, 30. October 2015

Signature(s)

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.