Spoofax Documentation

Release 0.5

MetaBorg

Jan 24, 2018

Contents

CHAPTER 1

Preface

These are lecture notes that go with the Compiler Construction course taught at Delft University of Technology.

The course takes the perspective of *declarative language definition*, using high-level declarative meta-language the define / specify the various aspects of programming languages. Given such definitions, the implementation of various compiler components can be generated automatically. In the course we study the use of meta-languages to define compilers. In the lab assignments for the course students build a compiler for MiniJava, a subset of the Java programming language using the Spoofax Language Workbench.

In the lectures and in these lecture notes, we study these meta-languages, and also their underlying theory and implementation.

CHAPTER 2

What is a Compiler?

This book is about compiler construction. Before we dive into the details of constructing compilers, we need to understand the motivation for doing so.

2.1 Slides

PDF

2.2 Etymology

What is that word? According to the *[WiktionaryCompile]*:

English

Verb

compile (***third-person singular simple present compiles, present participle compiling, simple past and past participle compiled***)

1. (transitive) To put together; to assemble; to make by gathering things from various sources. Samuel Johnson compiled one of the most influential dictionaries of the English language.
2. (obsolete) To construct, build.
3. (transitive, programming) To use a compiler to process source code and produce executable code. After I compile this program I'll run it and see if it works.
4. (intransitive, programming) To be successfully processed by a compiler into executable code. There must be an error in my source code because it won't compile.
5. (obsolete, transitive) To contain or comprise.
6. (obsolete) To write; to compose.

And where does it come from? Again according to the *[WiktionaryCompilo]*:

Latin

Etymology

From con- (“with, together”) + pīlō (“ram down”). Pronunciation

- (Classical) IPA(key): /kompi.lo/, [kmpi.o]

Verb

compīlō (present infinitive compīlāre, perfect active compīlāvī, supine compīlātum); first conjugation

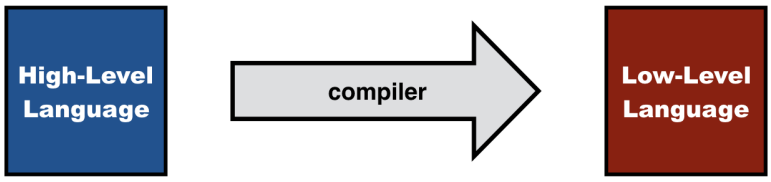
> I snatch together and carry off; plunder, pillage, rob, steal.

How does ‘putting together’ relate to what we think of as compilers? The Wikipedia page on The History of Compiler Construction sheds light on the issue:

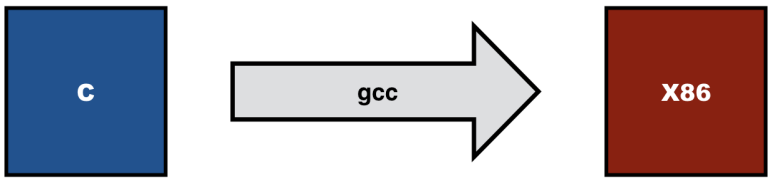
The first compiler was written by Grace Hopper, in 1952, for the A-0 System language. The term compiler was coined by Hopper.[1][2] The A-0 functioned more as a loader or linker than the modern notion of a compiler.

2.3 What is a Compiler?

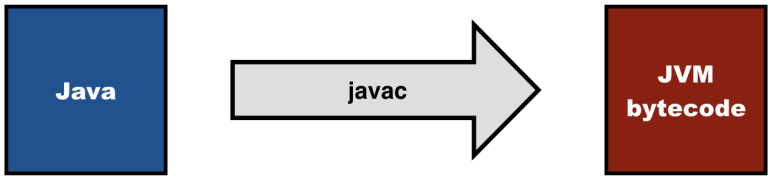
In our modern understanding, compilers are translaters. In particular, a compiler translates high-level programs to low-level programs



Let’s look at some typical instances. A C compiler translates C programs to object code, i.e. instructions for some computer architecture. Examples of C compilers are GCC and clang.



A Java compiler translates Java programs to bytecode instructions for the Java Virtual Machine. In the lab for this

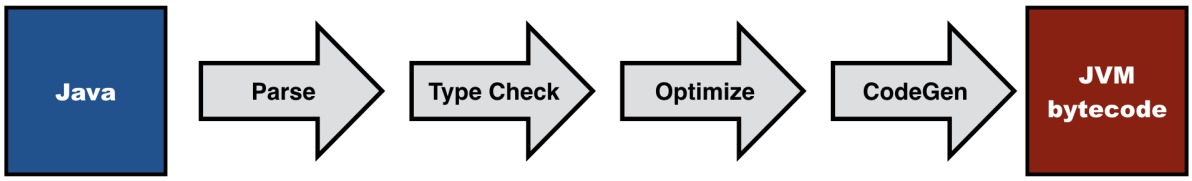


2.4 Compiler Architecture

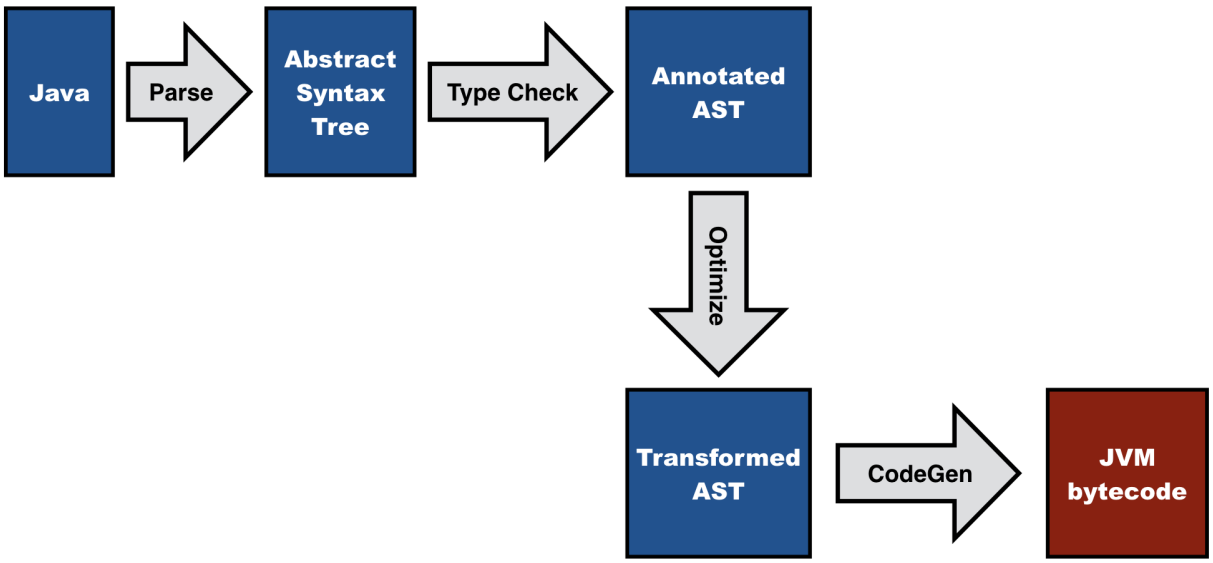
The central topic of a course on compiler construction is understanding what the black box of a compiler looks like inside.

Early compilers were *one-pass* compilers, which look at each line of code only once. This was important in order to fit in the limited memory of those days. This architecture posed limitions on language design, such as declaration before use.

Modern compilers do not suffer the harsh resource constraints of early compilers and are typically designed as a sequence of passes or stages each of which completely process a program or program unit. That is each, pass loads (the representation of) the entire program unit in memory.



Each pass typically changes the representation of the program being compiled. Thus, a compiler can be seen as the composition of a series of translaters, each consuming a program in some representation and producing a program in another representation.

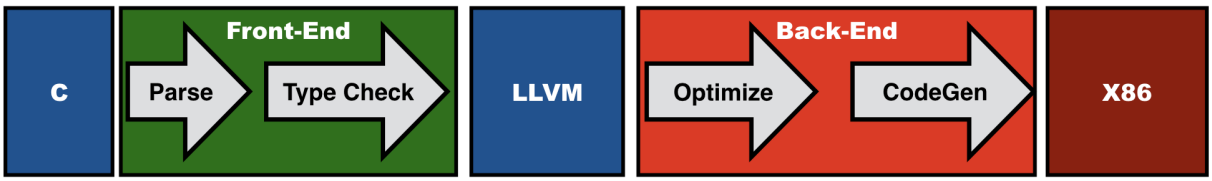


The typical components of a compiler pipeline are:

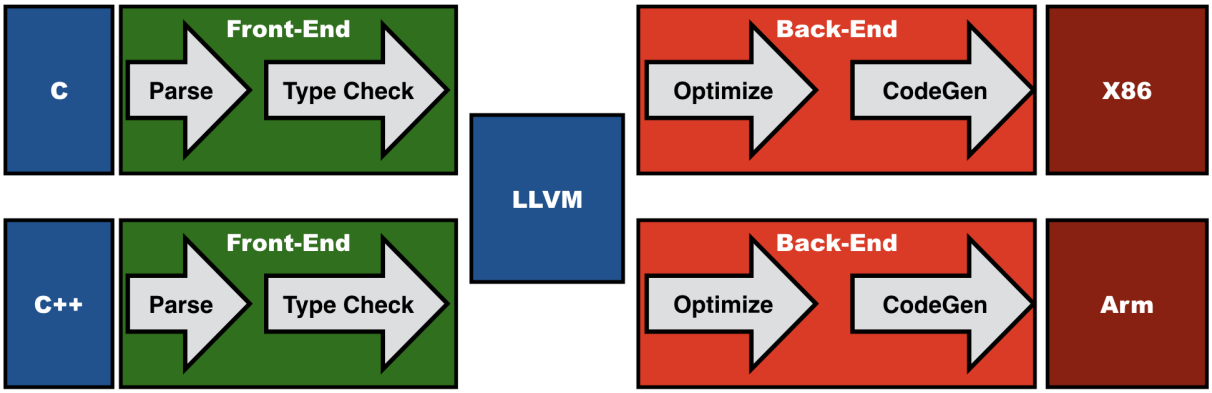
- *Parser*: Reads in program text, checks that it complies with the *syntactic* rules of the language, and produces an *abstract syntax tree*, which represents the underlying (syntactic) structure of the program.
- *Type checker*: Consumes an abstract syntax tree and checks that the program complies with the *static semantic* rules of the language. To do that it needs to perform name analysis, relating uses of names to declarations of names, and checks that the types of arguments of operations are consistent with their specification.
- *Optimizer*: Consumes a (typed) abstract syntax tree and applies transformations that improve the program in various dimensions such as execution time, memory consumption, and energy consumption.
- *Code generator*: Transforms the (typed, optimized) abstract syntax tree to instructions for a particular computer architecture. (aka instruction selection)
- *Register allocator*: Assigns physical registers to symbolic registers in the generated instructions.
- *Linker*: Most modern languages support some form of modularity in order to divide programs into units. When also supporting *separate compilation*, the compiler produces code for each program unit separately. The linker takes the generated code for the program units and combines it into an executable program.

2.5 Retargeting

The passes that make up a compiler are often divided in two clusters, front-end and back-end:



The focus of the front-end is on *analysis*, i.e. parsing (syntactic analysis) and type checking (static analysis). The focus of the back-end is on *synthesis*, i.e. optimization and code generation. The advantage of this division is that front-ends and back-ends can be used in multiple combinations, provided they share a common intermediate language:



2.6 Why do we need compilers?

So, studying compiler construction means studying these compiler components. And we will. However, compilers are not quite so stereotypical. The techniques that we study here have more applications than for constructing variants of C and Java. Let's take a step back and investigate why we need compilers in the first place.

Compilers are used to support programming. What is that? And how do they do that?

Programming is instructing a computer to perform computations. The Central Processing Units (CPUs) of computers process low-level operations

- fetch data from memory
- store data in register
- perform basic operation on data in register
- fetch instruction from memory
- update the program counter
- etc.

However, such operations are far removed from the problems we want to address with software.

2.7 Programming is Expressing Computational Intent

We use computers to get stuff done

- Buy shoes
- Book a trip
- Design a lecture

We program so that we can use computers to get stuff done. Programs are the intermediaries for getting stuff done

- Web browser
- Shoe webshop
- Text editor

When programming we would like to think about the thing the program is doing for us, i.e. computational thinking:

"Computational thinking is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out." *[CompThink]*

Writing instructions to fetch data from memory or incrementing the program counter does not contribute to effective computational thinking. *It does not allow us to express our intentions at the right level of abstraction.* The machine does not understand us!

2.8 Types of Compilers

So, a compiler is a translater. And often that is understood as a translater from high-level languages to machine languages. However, the techniques employed in the construction of such compilers are also useful in other types of translaters, of which there are many kinds:

- Compiler: translates high-level programs to machine code for a computer
- De-compiler: translates from low-level language to high-level language
- Cross-compiler: runs on different architecture than target architecture
- Source-to-source compiler (transpiler): translate between high-level languages
- Interpreter: directly executes a program (although prior to execution program is typically transformed)
- Bytecode compiler: generates code for a virtual machine
- Just-in-time compiler: defers (some aspects of) compilation to run time
- Hardware compiler: generate configuration for FPGA or integrated circuit

See *[CompilerWikipedia]* for a more extensive discussion and links.

Thus, the classical compiler that translates a high-level imperative language to machine code is just one instance of a large family of programs that operate on programs as data.

2.9 Levels of Understanding Compilers

There are many dimensions to the study of compilers and programming languages. The goal of this course is to get a general understanding of the domain of compilation and of the techniques employed in the construction of compilers. That means that you need to go through the following levels of understanding compilers.

At the *base level* you understand the construction of a specific compiler. In this course you will build a compiler that translates MiniJava programs to Java Bytecode. This requires:

- Understanding a programming language (MiniJava)
- Understanding a target machine (Java Virtual Machine)
- Understanding a compilation scheme (MiniJava to Byte Code)

However, you should *generalize* from this experience in order to understand the general principles and architecture of compiler construction. This requires

- Understanding architecture of compilers
- Understanding (concepts of) programming languages
- Understanding compilation techniques

This level of understanding provides you with *design patterns* for programming compilers, which you can employ in the construction of compilers for different languages. Perhaps even for a language of your own design.

However, we can do better than that. Instantiating design patterns can be repetitive and involve a lot of *boilerplate code*. We can abstract from such design patterns through *linguistic abstractions* for sub-domains of compilation. This requires

- Understanding (principles of) syntax definition and parsing
- Understanding (principles of) static semantics and type checking
- Understanding (principles of) dynamic semantics and interpretation/code generation
- Understanding design of meta-languages and their compilation

2.10 A First Taste of Compiler Construction

As a first taste of what we will do in this course, browse through the section 'Language Definition with Spoofax' (*[Calc]*), which runs through a complete definition of a little calculator language with the Spoofax Language Workbench.

2.11 Further Reading

CHAPTER 3

Declarative Language Definition

Notes on a general approach to declarative language definition. The goal is to separate the concerns of language definition from language implementation. A language definition states the specific rules for a language. Language implementations typically have much in common. By factoring out the language-specific rules into a declarative meta-language, the language-independent aspects of implementations can be automatically generated.

3.1 A Language Designer's Workbench

- Objective: A workbench supporting design and implementation of programming languages
- Approach: Declarative multi-purpose domain-specific meta-languages
- Meta-Languages: Languages for defining languages
- Domain-Specific: Linguistic abstractions for domain of language definition (syntax, names, types, . . .)
- Multi-Purpose: Derivation of interpreters, compilers, rich editors, documentation, and verification from single source
- Declarative: Focus on what not how; avoid bias to particular purpose in language definition

3.2 Meta-Language Design

Representation

- Standardized representation for <aspect> of programs
- Independent of specific object language

Specification Formalism

- Language-specific declarative rules
- Abstract from implementation concerns

Language-Independent Interpretation

- Formalism interpreted by language-independent algorithm
- Multiple interpretations for different purposes
- Reuse between implementations of different languages

CHAPTER 4

Declarative Syntax Definition

In this chapter we explore the definition of *syntax*, which governs the form and structure of programs.

4.1 Slides

PDF

PDF

4.2 Further Reading

https://en.wikipedia.org/wiki/Syntax

https://en.wikipedia.org/wiki/Syntax_(programming_languages)

https://en.wikipedia.org/wiki/Formal_language

https://en.wikipedia.org/wiki/Formal_grammar

https://en.wikipedia.org/wiki/Chomsky_hierarchy

https://en.wikipedia.org/wiki/Parsing

CHAPTER 5

Formatting

- Formatting
- Completion

CHAPTER 6

Parsing

- Parsing
  - Scanning / Tokenization / Lexical Analysis
  - Parsing
  - Parsing Algorithms
  - LL Parsing
  - LR Parsing

[ScannerlessWikipedia]

6.1 Further Reading

CHAPTER 7

Transformation

- Term Rewriting
- Rewriting Strategies

PDF

CHAPTER 8

Static Semantics

- Type Systems
- Constraint-Based Static Semantics
- Name Binding
- Type Analysis
- Completion
- Other Approaches

8.1 Name Resolution

Slides PDF

The following video is from a talk about name binding with scope graphs at the Curry On 2017 conference and covers some of the same material.

8.2 Type Checking

Slides PDF

8.3 Constraint Resolution I

Slides PDF

8.4 Constraint Resolution II

Slides PDF

Exercises PDF

Exercises + Solutions PDF

CHAPTER 9

Dynamic Semantics

- Running Programs
- Operational Semantics
- Partial Evaluation
- Frames

CHAPTER 10

Static Analysis

- Data-Flow Analysis

10.1 Data-Flow Analysis

Slides PDF

Exercises PDF

Exercises + Solutions PDF

CHAPTER 11

Code Generation

- Virtual Machines
- Run-Time Systems
- Garbage Collection

Bibliography

[WiktionaryCompile] https://en.wiktionary.org/wiki/compile

[WiktionaryCompilo] https://en.wiktionary.org/wiki/compilo#Latin

[CompThink] Jeanette M. Wing. Computational Thinking Benefits Society. In Social Issues in Computing. January 10, 2014. <http://socialissues.cs.toronto.edu/index.html>

[CompilerWikipedia] <https://en.wikipedia.org/wiki/Compiler>

[Calc] Language Definition with Spoofax. A complete example of a Spoofax language definition for a little calculator language. <http://www.metaborg.org/en/latest/source/langdev/meta/lang/tour/index.html>

[DeclSD] Lennart C. L. Kats, Eelco Visser, Guido Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In William R. Cook, Siobhán Clarke, Martin C. Rinard, editors, Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA. pages 918-932, ACM, Reno/Tahoe, Nevada, 2010. <http://doi.acm.org/10.1145/1869459.1869535>

[SPT] Lennart C. L. Kats, Rob Vermaas, Eelco Visser. Integrated language definition testing: enabling test-driven language development. In Cristina Videira Lopes, Kathleen Fisher, editors, Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011. pages 139-154, ACM, 2011. <http://doi.acm.org/10.1145/2048066.2048080>

[Completion] Luis Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, Eelco Visser. Principled syntactic code completion using placeholders. In Tijs van der Storm, Emilie Balland, Dániel Varró, editors, Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016. pages 163-175, ACM, 2016. <https://doi.org/10.1145/2997364.2997374>

[Templates] Tobi Vollebregt, Lennart C. L. Kats, Eelco Visser. Declarative specification of template-based textual editors. In Anthony Sloane, Suzana Andova, editors, International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012. pages 1-7, ACM, 2012. <http://doi.acm.org/10.1145/2427048.2427056>

[SDF3] Syntax Definition with SDF3. Documentation. <http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/index.html>

[ScannerlessWikipedia] https://en.wikipedia.org/wiki/Scannerless_parsing