Notes on Compiler Construction

Emil Sekerinski, **McMaster University, January 2021; updated January 2023**

> This series of notebooks relies on following Jupyter extensions for classic Jupyter; these do not work and are not needed with JupyterLab:
>
> - `exercise` with `rubberband`: for revealing solution hints with the ⊞ symbol; these extensions can also be installed through the `Jupyter nbextensions configurator`
> - `jupyter-emil-extension`: for formatting of algorithms and layout of slides. Install locally by:

curl -LJO https://gitlab.cas.mcmaster.ca/parksj6/jupyter-emil-extension/-/jobs/artifacts/master/download?job=build unzip -a jupyter-emil-extension.zip cd dist/ python3 -m pip install -e . --upgrade jupyter nbextension install --py jupyter_emil_extension jupyter nbextension enable --py jupyter_emil_extension

```
- [`RISE`](https://github.com/damianavila/RISE): optionally, for viewing the notebooks as slides

Also needed for some of the svg images is the [`Google Noto Sans`](https://www.google.com/get/noto/) font
```

What This Series of Notebooks are About

Compilers convert computer programs in "high level" languages to executable code. These lecture notes introduce the concepts of compilation and illustrates those by a compiler for a small imperative programming language. However, this series of notebooks and the accompanying exercises are not just about writing compilers!

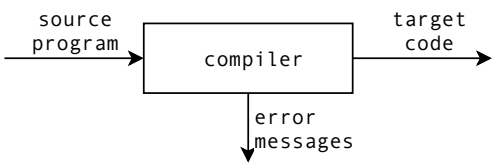
- You will learn to use parsing, analysis, and generation techniques for compilation and other syntax-directed processing (e.g. for query languages, mark-up languages, protocols).
- You will see other topics in computing connected, in particular computer architecture, programming languages, formal languages, and operating systems.
- You will appreciate efficiency issues in programming languages through the understanding of memory layout of data types (e.g. arrays, objects) and compilation of control structures (e.g. short circuit evaluation, recursion), which leads to a more resource-conscientious programming style.
- You will understand why historically programming languages are defined in the ways they are, and be better prepared for new languages.
- You will learn how compilation techniques (e.g. byte code, just-in-time, garbage collection) affect the suitability of programming languages for specific applications.
- You will learn about optimization techniques and the impact of processor architectures, which helps in using compilers more effectively.
- You will be made aware of compiler construction tools (e.g. scanner and parser generators) and will be able to judge if and when to use them.

For achieving above, besides studying these notes, it is imperative to engage in smaller and larger exercices:

- Self-review questions are embedded in these notes. You are urged to attempt them on your own while reading a chapter. You may reveal a hint by clicking on the ⊞ symbol to the left of the question.
- Further exercises are provided at the end of each chapter. These are best attempted after reading a chapter and before proceeding with the next one.

Task and Structure of Compilers

In general a compiler processes a structured source text and generates (simpler structured) target code or error messages.



Sources	Targets
programming languages: C, Java, Go, ...	*virtual machine languages:* LLVM, JVM, CIL, WebAssembly, ...
virtual machine languages: LLVM, JVM, CIL, WebAssembly, ...	*executables code:* RISC-V, ARMv8, x86-64, ...
scripting languages: bash, Python, JavaScript, ...	*assembly language:* as, nasm, ...
text formatting languages: TeX, html, markdown, ...	*coprocessor code:* FPU, GPU, FPGA, ...
interchange languages: JSON, XML, ...	*layout languages:* PDF, html, RTF, ...

Sources can also be *preference files, configuration files, database queries,* and *hardware description languages,* even if those are not compiled to executable code. The same languages, e.g. virtual machine languages, can be both target and source of different compilation processes.

Starting with Algol 60 (Backus et al. 1963), the first programming language with a formally defined *grammar,* the translation process of a compiler is guided by the *syntactical structure* of the source text. The method of *syntax-directed compilation* leads to the following decomposition:

- *Analysis:* recognising the structure of the source text according to its grammar.
- *Synthesis:* generating target code from the recognized structure.

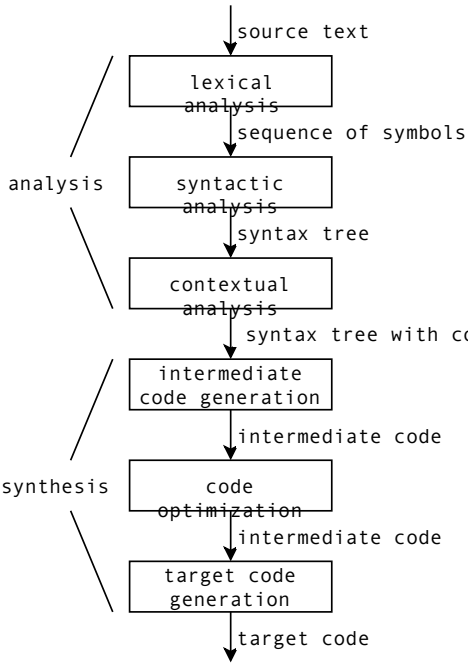
Analysis and synthesis are each split into a number of consecutive *phases* with different *intermediate representations*.

Symbols (or *tokens*) are sequences of characters like a number (a sequences of digits), an identifier (a sequence of letters and digits), a keyword (**if**, **while**), a separator (comma, semicolon).

The *lexical analysis* recognizes symbols; it is carried out by a *scanner*, hence also called *scanning*. The *syntactic analysis* recognizes the syntatic structure, a *syntax tree*; syntactic analysis is carried out by a parser, hence also called *parsing*.

The *contextual analysis* (or *type-checking*) augments the syntax tree with type information. The redundancy provided by type information allows common programming errors to be detected early and quickly; type information is also used by the compiler to memory management. Only syntactically correct and type-correct programs have a meaning and are subject to code generation.

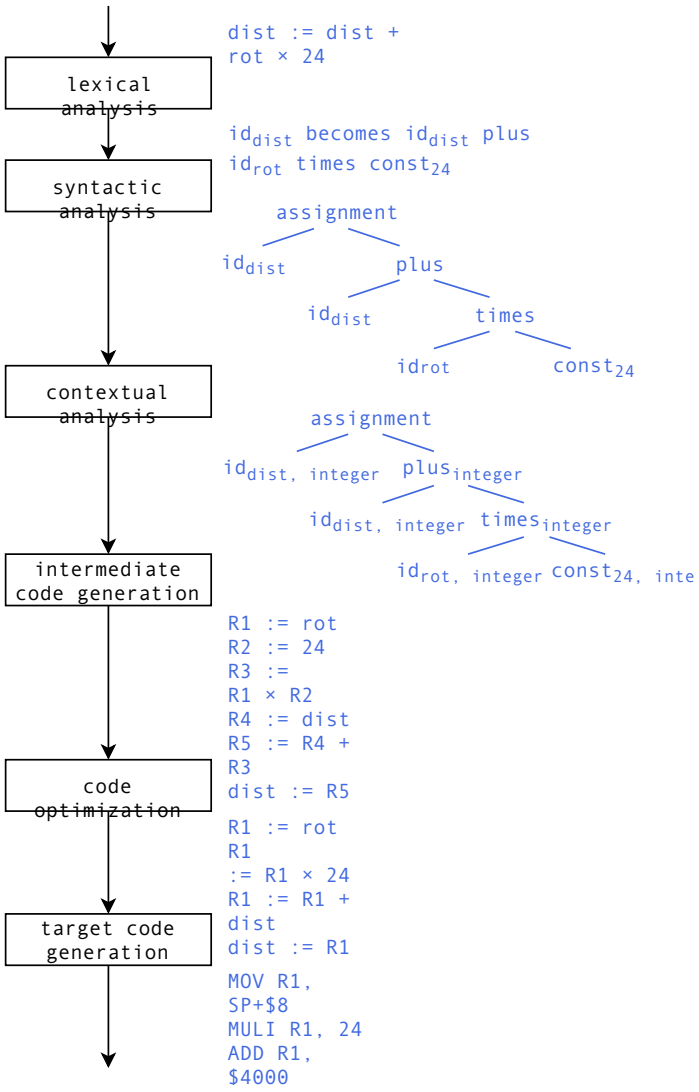
In the first phase of code generation, *intermediate code* is generated, a representation that is simpler than the target code, but close enough that it can be straighforwardly translated to the target code and that the gain of *code optimizations* can be judged.



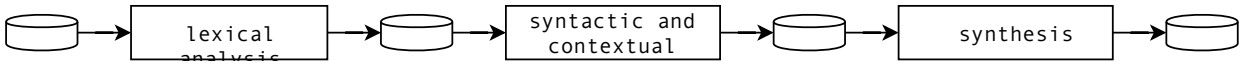
For example, suppose the context includes that `dist` is an integer variable and that `rot` is an integer parameter of the enclosed procedure `update`:

```
var dist: integer
procedure update(rot: integer)
```

The figure to the right illustrates the compilation of the assignment `dist := dist + rot × 24` in the body of `update`. The contextual analysis augments with syntax tree with type information; here all expressions are of `integer` type. The intermediate code is *register-based:* all arithmetic operations can only involve registers, here `R1`, `R2`, etc. The code optimization reduces the number of registers and the number of instructions. The generated code is specific as how to access variables; here, `rot` is SP-relative (stack pointer) with offset 8_{16} (base `16`) and `dist` is global at location 4000_{16}.

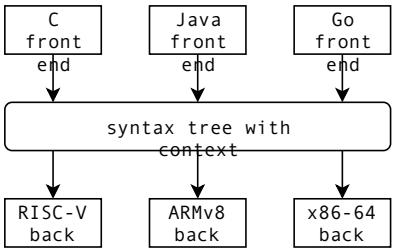


Phases conceptually decompose of the task of a compiler. In practice, several phases are merged into *passes* such that no intermediate data structure is necessary between the phases of a pass.



Historically, files were used for passing the data between the passes. Modern compilers use main memory.

A common and advantageous separation of the task is by dividing the compiler into a *front end* and a *back end*.

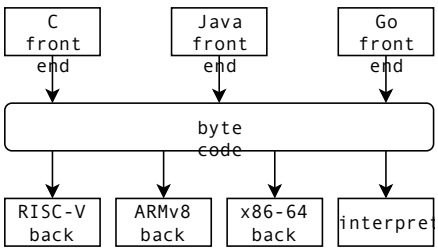


This division helps reducing the efforts for writing compilers for different targets for the same source language by sharing the front end, or for different source languages for the same target by sharing the back end.

In principle, given `m` source languages and `n` targets, this reduces the effort of writing `m × n` compilers to `m` front ends `+` `n` back ends.

In practice, this only works if the languages respectively the targets are sufficiently similar. It is nevertheless a good structuring principle for flexibility.

A variation of the front end / back end decomposition is when a virtual machine code rather than a syntax tree is used. In essence, the front end and back end become compilers of their own.



Some virtual machine codes represent each instruction by a single byte, hence are called *byte codes*. The compactness of byte code makes the memory footprint and the download times appealingly small.

Byte codes can either be *interpreted* without further compilation, i.e. executed instruction by instruction, or further compiled to excutable machine code *just-in-time* when loaded to main memory or while being executed. An advantage of just-int-time compilation compared to *ahead-of-time* compilation is that all characteristics of the processor are known when compiling and that the code execution itself can influence the compilation.

Structure of the Notebooks

The lecture notes are organized as one notebook for each chapter:

1. Language and Syntax
2. Regular Languages
3. Analysis of Context-free Languages
4. Syntax-Directed Translation
5. The Construction of a Parser
6. A Stack Architecture as Target
7. A RISC Architecture as Target
8. Further Data Types
9. Separate Compilation and Executable Files
10. Concurrency and Parallelism
11. Garbage Collection
12. Generalized Parsing

Historic Notes and Further Reading

A chapter in Wirth's book *Algorithms + Data Structures = Programs* (Wirth 1976) illustrates with a compiler for a subset of Pascal the principles that became the foundation for a whole line of work, including Turbo Pascal, an early interactive programming environment (which later became Delphi) and UCSD Pascal. Later versions of Wirth's compiler generate RISC code (Wirth 1996).

UCSD Pascal includes an editor and a whole operating system. The compiler generates p-code, a Pascal-influenced byte code, which is then interpreted. On the Apple II computer, UCSD Pascal, which itself is p-code, occupies 16KB of main memory, leaving the rest for programs and data. P-code influenced the design of JVM, the Java Virtual Machine, which itself influenced the design of CIL, the Common Intermediate Language (see Exercises below).

Some programming languages, notably C (Kernighan and Ritchie 1974) and Pascal (Jensen and Wirth 1974), were originally defined with *single pass compilation* in mind, due to the restricted size of main memory at that time: a single pass compiler generates target code while the source file is read. To allow static type checking and code generation with mutally recursive procedures, these languages have *forward declarations* (called *function prototypes* in C):

```
procedure p(x: integer); forward; (* forward declaration of p, without body *)
procedure q(); (* q with body *)
    begin ... p(3) ... end; (* call to p is allowed *)
procedure p(x: integer); (* parameters must be the same as in forward declaration *)
    begin ... q() ... end; (* call to q is allowed *)
```

Besides requiring forward declarations, single pass compilation also limits code optimizations. For modern compilers, main memory restrictions are not a concern: limits of human comprehension has kept the size of individual source files unchanged over decades, but hardware technology has provided compilers with an abundance of main memory.

Exercises

Exercise 1. The Common Interface Language (CIL) is a virtual machine that was originally developed by Microsoft and then standardized by ECMA. Front ends exist for C# (an object-oriented language), F# (a functional language), and "Managed C++", but not for full C++. What is the difference between Managed C++ and C++? Explain which properties of CIL prohibit full C++ to be supported!

Exercise 2. The Java Virtual Machine, JVM, is a byte code that was originally developed for Java but has been used as the target for other languages. Provide pointers to at least five other languages that target JVM! Is JVM code interpreted, just-in-time compiled, or ahead-of-time compiled?

Exercise 3. The functionality of processors can be augmented by *coprocessors* or *units* on a different chip or on the same chip as the processor. By consulting a textbook on computer architecture, discuss the functionality of SIMD units, FPUs, GPUs, FPGAs, how they are programmed, and what role compilers play, if any!

Bibliography

Backus, J. W., F.L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, et al. 1963. “Revised Report on the Algorithm Language ALGOL 60.” Edited by P. Naur. *Communications of the ACM* 6 (1): 1–17. https://doi.org/10.1145/366193.366201.

Jensen, K., and N. Wirth. 1974. *Pascal - User Mannual and Report*. Springer-Verlag.

Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*. 2nd ed. Prentice Hall Professional Technical Reference.

Wirth, Niklaus. 1976. *Algorithms + Data Structures = Programs*. Prentice-Hall.

Wirth, Niklaus. 1996. *Compiler Construction*. Revised Edition 2005. Addison-Wesley. http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf.