Introduction

Compiler Construction

2024-10-15

Prof. Dr. Michael Kuhn

`michael.kuhn@ovgu.de`

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
`https://parcio.ovgu.de`

Outline

- Introduction
  - Organization
  - Lecture
  - Exercises
  - Outlook
  - Summary

Survey

- How familiar are you with C?
  1. Expert
  2. Advanced
  3. Beginner
  4. Not at all

Survey

- How familiar are you with Linux?
    1. Expert
    2. Advanced
    3. Beginner
    4. Not at all

Survey

- How familiar are you with Git?
    1. Expert
    2. Advanced
    3. Beginner
    4. Not at all

Lecture and Exercises

- Lecture: Tuesdays, 11:15–12:45
  - Foundation and background of compiler construction
  - We will use this time slot for group exercises and discussion
  - Based on [Thain, 2020]
- Exercises: Wednesdays, 9:15–10:45
  - Practical exercises about compiler construction
  - We will discuss solutions and take a look at the next exercise sheet
- Exam: Oral

Lecture and Exercises…

- Lecture will use the flipped classroom format
    - You will have to read (at most) one chapter per week
    - There will be *no* summary of the chapter's content
    - We will perform group exercises and discuss the content
- Lecture is supposed to be interactive
    - Please prepare questions if you do not understand something

Communication

- Please sign up for the Mattermost team
  - If there are questions about the lecture or exercises, please ask them there
  - Feel free to use it for discussion and communication with your fellow students
    - You can also use it to find people for your exercise group
  - You can of course also send us e-mails:
    - `michael.kuhn@ovgu.de` (lecture and general)
    - `michael.blesel@ovgu.de` (exercises)
- Slides, exercise sheets etc. will be available on the website

Literature

- Introduction to Compilers and Language Design (Douglas Thain) (`http://compilerbook.org/`)

Outline

Introduction

- Organization
- Lecture
- Exercises
- Outlook
- Summary

Topics

- Introduction (today ☺)
- Scanning
- Parsing (Parts 1 and 2)
- Abstract Syntax Trees
- Semantic Analysis
- Intermediate Representation

Topics...

- Memory Organization
- Assembly Language
- Code Generation
- Optimization
- Research Talks

Outline

Introduction

- Organization
- Lecture
- Exercises
- Outlook
- Summary

Overview

- Exercises will involve some programming in C
    - Trying out the concepts taught in the lecture
- You should have experience in a programming language
    - Experience in C is not necessary (but helps)
- We will also work on our cluster via SSH
    - Logging in and setting everything up will be part of the first exercise

Outline

Introduction

- Organization
- Lecture
- Exercises
- Outlook
- Summary

Motivation

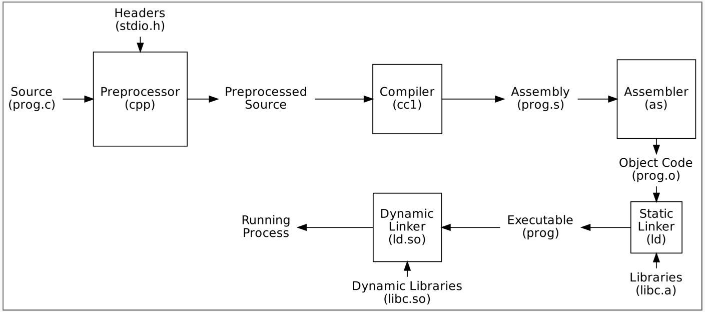
- Compilers translate programs from a source language to a target language
  - For example, C/C++ to machine code
  - Can also translate a high-level language into an intermediate representation
  - For example, Java source code to Java bytecode
- Compilers also help find errors at compile time
  - For example, uninitialized variables
  - Different languages have different strictness
- Compilers also improve performance using optimizations
  - Applying these optimizations in turn takes time and memory
  - Optimization potential is limited

Motivation…

- Understanding compilers makes you a better programmer
    - Helps understand how to write efficient and correct code
- Compiler knowledge allows you to create tooling
    - For example, debuggers, new languages or compilers

Toolchain

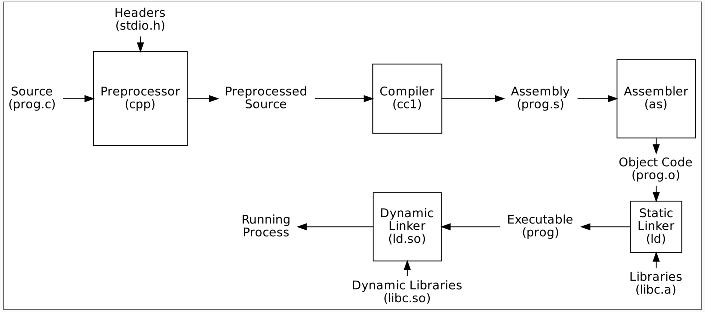
- Preprocessor performs relatively simple replacements
  - For example, `#include` or `#define` in C



[Thain, 2020]

Toolchain

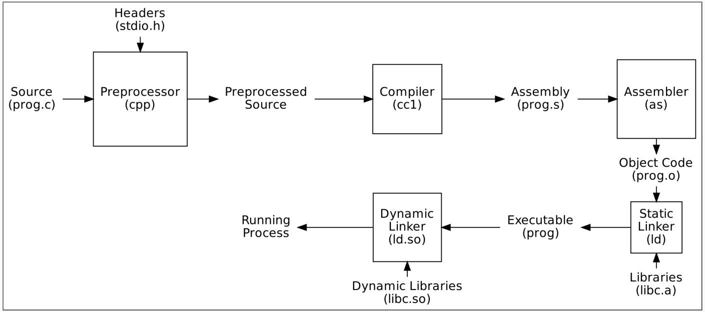
- Compiler translates individual translation units into assembly code
  - It scans and parses the code, performs checks and optimizations



[Thain, 2020]

Toolchain

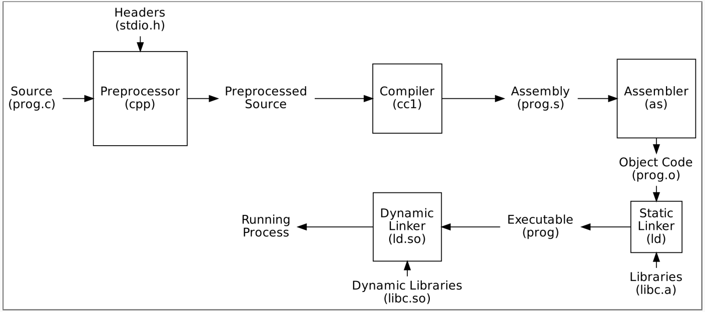
- Assembler translates assembly code into object/machine code
  - Object code does not contain concrete addresses



[Thain, 2020]

Toolchain

- Linker turns one or multiple object files into an executable program
  - Fills in everything left open by the assembler



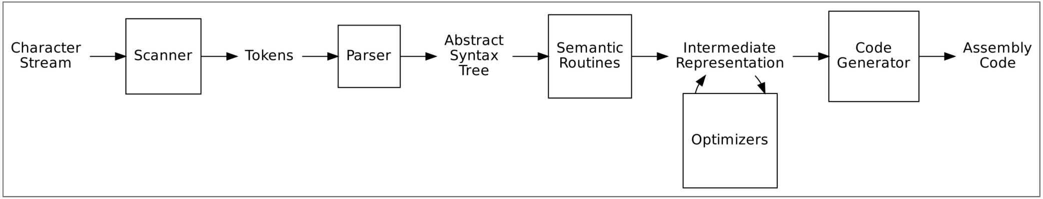
[Thain, 2020]

Live Demo

- `cat example.c`
- `gcc -E example.c`
- `gcc -S example.c`
- `gcc -c example.c`
- `gcc -v example.c`

Compiler

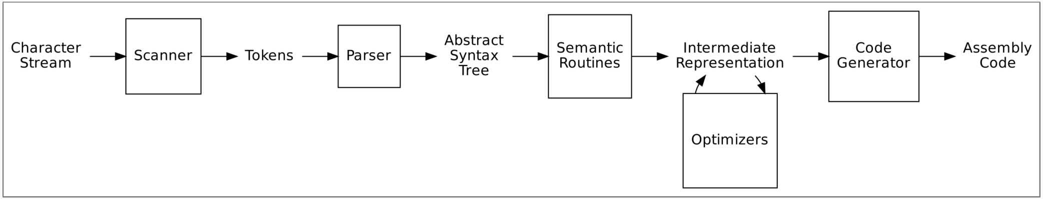
- Scanner turns the source code into tokens
  - For example, a token could be `int` or `42`



[Thain, 2020]

Compiler

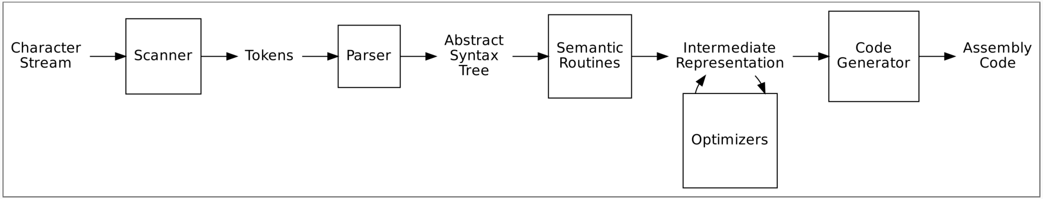
- Parser turns tokens into statements or expressions
    - Controlled by a grammar and outputs an abstract syntax tree (AST)



[Thain, 2020]

Compiler

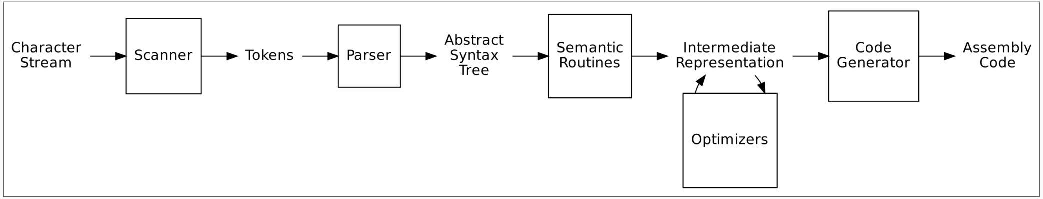
- Semantic routines derives meaning about the program
  - For example, resulting data types of calculations



[Thain, 2020]

Compiler

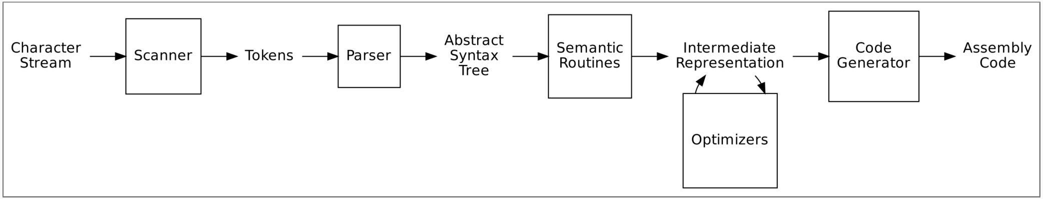
- Optimizers apply certain optimizations to the intermediate representation (IR)
  - For example, loop unrolling, inlining or vectorization



[Thain, 2020]

Compiler

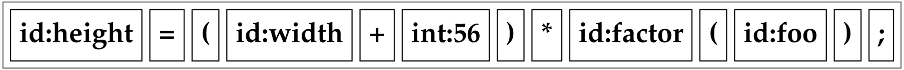
- Code generator turns IR into assembly code
  - Responsible for register allocation, instruction selection and sequencing



[Thain, 2020]

Example

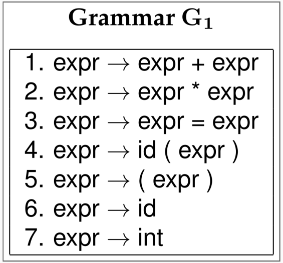
- We want to compile the following code into assembly code
  - `height = (width+56) * factor(foo);`
- The scanner will turn the source code into tokens
  - `height`, `width`, `factor` and `foo` are identifiers
  - Purpose of tokens is still unclear: some identifiers are variables, one is a function
  - Variable data types are also not clear yet



[Thain, 2020]

Example…

- The parser checks whether the sequence of tokens is valid
  - Validity is checked using a grammar of the language
  - Grammars consist of rules
  - Rules are applied and turned into an AST



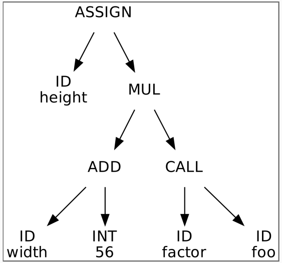
Grammar G_1

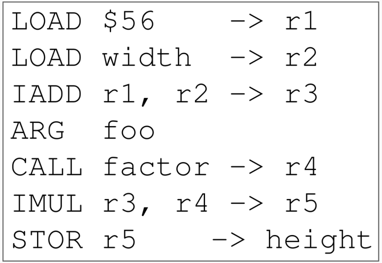
1. expr → expr + expr
2. expr → expr * expr
3. expr → expr = expr
4. expr → id (expr)
5. expr → (expr)
6. expr → id
7. expr → int

[Thain, 2020]

Example…

- Semantic routines derive additional meaning
  - For example, type checking for assignments and calculations
  - Post-order traversal turns AST into IR
  - IR typically looks like assembly and assumes infinite registers





```
LOAD $56    -> r1
LOAD width  -> r2
IADD r1, r2 -> r3
ARG  foo
CALL factor -> r4
IMUL r3, r4 -> r5
STOR r5    -> height
```

[Thain, 2020]

Example…

- Code generator turns IR into assembly code
  - Compilers are often highly modular: same IR for multiple languages, optimizer modules
  - Multiple code generators for different architectures

```
MOVQ    width, %rax      # load width into rax
ADDQ    $56, %rax        # add 56 to rax
MOVQ    %rax, -8(%rbp)   # save sum in temporary
MOVQ    foo, %edi        # load foo into arg 0 register
CALL    factor           # invoke factor, result in rax
MOVQ    -8(%rbp), %rbx   # load sum into rbx
IMULQ   %rbx             # multiply rbx by rax
MOVQ    %rax, height     # store result into height
```

[Thain, 2020]

Live Demo

- `https://godbolt.org/`

Outline

Introduction

- Organization
- Lecture
- Exercises
- Outlook
- Summary

Summary

- Compilers translate programs from a source language to a target language
- Compilers can also help find errors and optimize programs
- Toolchain consists of multiple components
    - Preprocessor, actual compiler, assembler and linker
- Compiler itself contains several modules
    - Scanner, parser, semantic analysis, optimizers and code generator

References

[Thain, 2020] Thain, D. (2020). ***Introduction to Compilers and Language Design: Second Edition.*** http://compilerbook.org/.