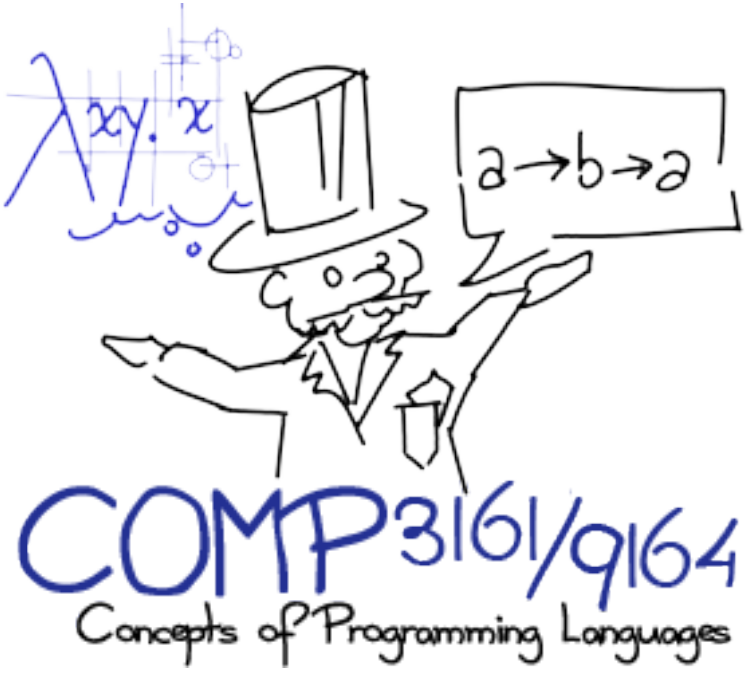


Introduction

Johannes Åman Pohjola
UNSW
Term 3 2022

Who are we?

I am Johannes Åman Pohjola, the course convenor. I'm a lecturer at UNSW. My research area is formal methods, particularly interactive theorem proving, compiler verification and concurrency theory.

Zoey Chen, James Davidson, Raphael Douglas Giles, Charran Kethees, and Tiana Tsang Ung. A lot of the material is inherited

from previous convenors, including Liam O'Connor, Christine Rizkallah and Gabriele Keller.

Contacting Us

`http://www.cse.unsw.edu.au/~cs3161`

Forum

There is an Ed forum available on the website. Questions about course content should typically be made there. You can ask us private questions to avoid spoiling solutions to other students.

Administrative questions can be sent to `cs3161@cse.unsw.edu.au`.

What do we expect?

Maths

This course uses a significant amount of *discrete mathematics*. You will need to be reasonably comfortable with *logic*, *set theory* and *induction*. MATH1081 is neither necessary nor sufficient for aptitude in these skills. We teach enough of it to keep the course reasonably self-contained, but some self-study may be needed.

Programming

We expect you to be familiar with C and at least one other programming language. Course assignments 1 and 2 are in Haskell. Only very simple Haskell is required, but some self-study may be needed.

Assessment

Assignment 0	15%
Assignment 1	17.5%
Assignment 2	17.5%
Final Exam	50%

Tutorials

- Start this week on.
- You may change tutorials, just seek approval first.
- Please attempt some of the questions beforehand.

Assignment 0

- Focuses on theory and proofs.
- It will be released in Week 3 and due in Week 4.
- Aim to have marks back by census date (not guaranteed).

Assignments 1–2

- Given a formal specification, implement in Haskell.
- Released around Week 5 and Week 8
- Approximately 2 weeks to complete each assignment.

Lectures

- Lectures will be delivered in-person and via Zoom, concurrently.
- Recordings will be made available on Echo360.
- Separate lecture notes will also be published on occasion.

Books

There is no textbook for this course. Regular written lecture notes are made available throughout the semester, along with challenge exercises.
Much of the course material is covered in these two excellent books, however their explanations may differ and the usual disclaimers apply — this course does not follow these books exactly:

- *Types and Programming Languages* by Benjamin Pierce, MIT Press. https://www.cis.upenn.edu/~bcpierce/tapl/
- *Practical Foundations for Programming Languages* by Bob Harper, Cambridge University Press. http://www.cs.cmu.edu/~rwh/pfpl.html

Course Content

This is a programming language *appreciation* course. This means we focus on the three R's of computer science, giving you the skills to:

Read and understand new programming languages;

Write your own programming languages; and

Reason about programming languages in a rigorous way.

Why Read?

The choice of programming language affects nearly every aspect of a system:

- Design
- Development Costs and Productivity
- Safety and Security
- Performance

The Obvious

Learning to read and understand new programming languages is a vital skill in any computing discipline.

Why Write?

You may not implement a general-purpose programming language like C or Haskell in your career.

However..

Every company has its own hand-rolled *domain-specific* language for accomplishing some task, often embedded in another language in a very ad-hoc and ugly way.

Example

XSLT, Perl scripts for processing text files, CSE's `give` system, etc.

Learn how to make a PL properly and save yourself and your colleagues from headaches.

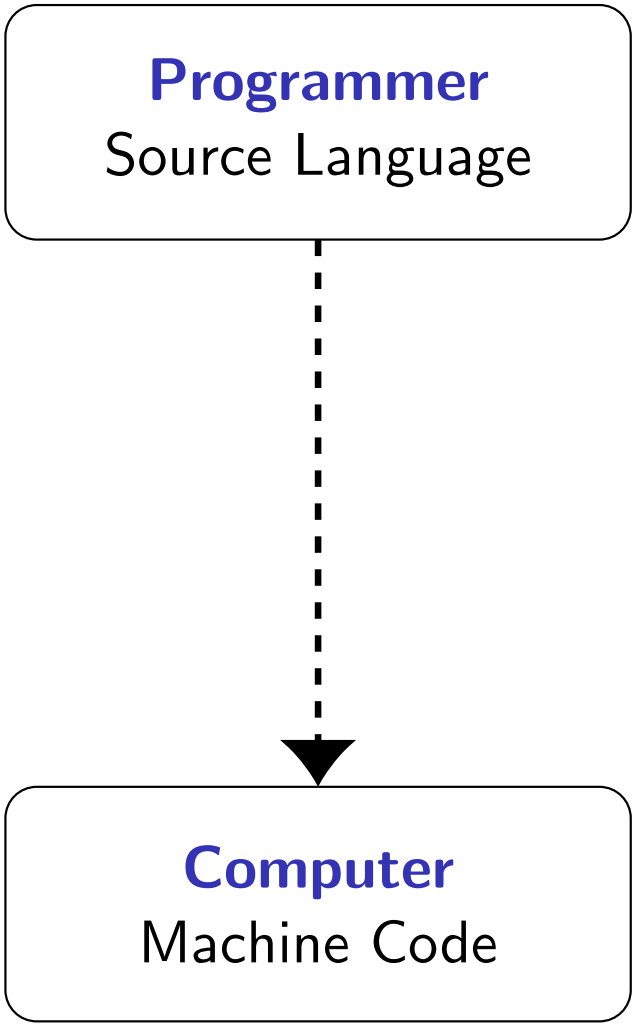
Why Reason?

Programming languages are formal languages. Formal specification and proof allows us to:

- Design languages *better*, avoiding *undefined behaviour* and other goblins.
- Make languages easier to process and parse. COMP3131
- Give a mathematical meaning to programs, allowing for *formal verification* of programs. COMP4161, COMP2111, COMP6721
- Develop algorithms to find bugs automatically. COMP3153
- Rigorously analyse optimisations and other program transformations.

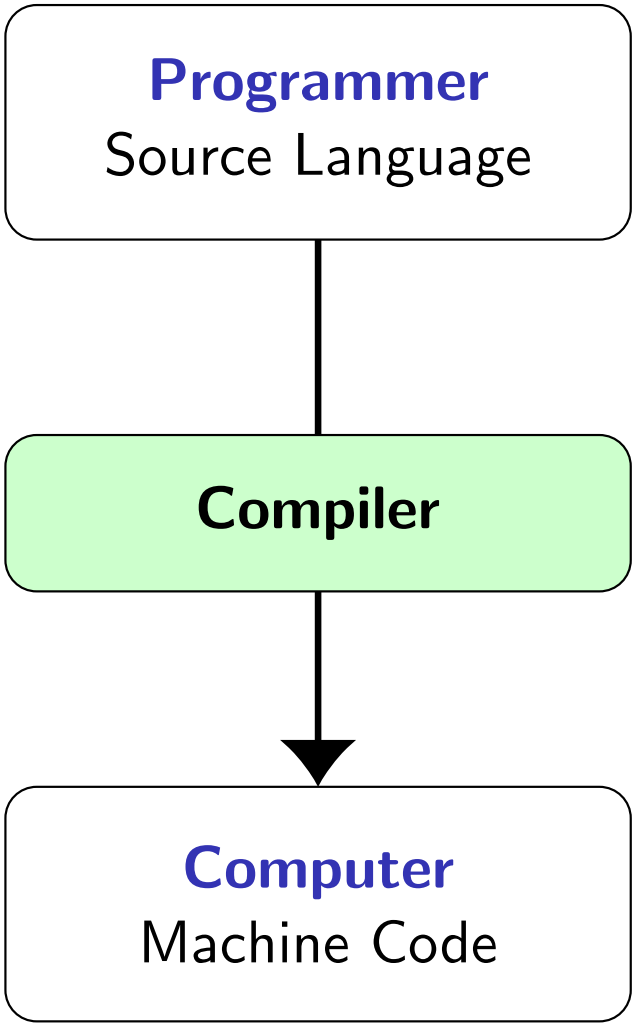
These tools are also very important for the pursuit of research in programming languages.

Bridging the Gap



Computers typically can't execute source code directly.

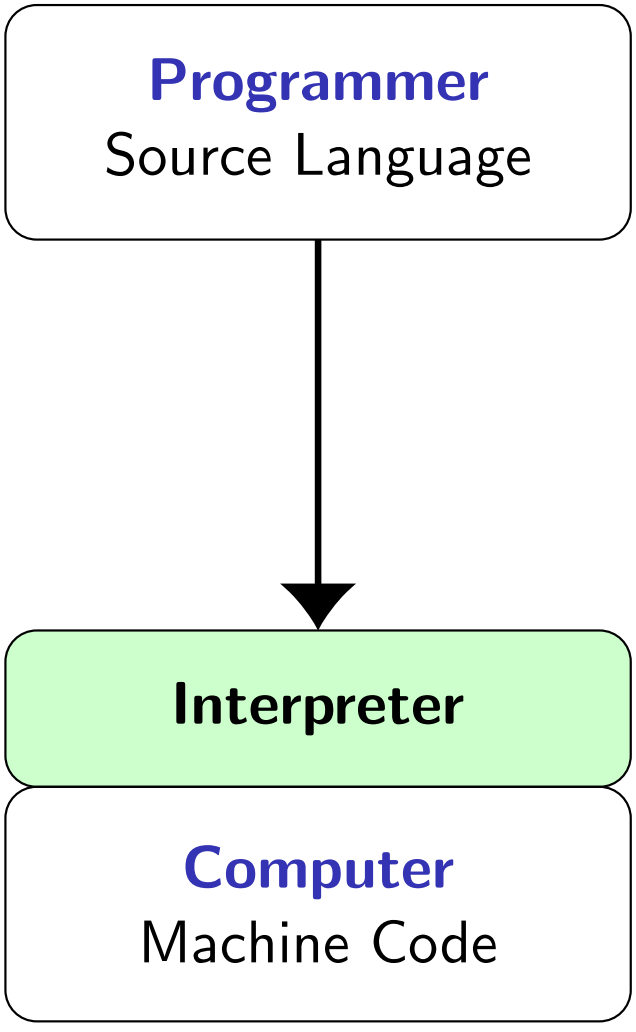
Bridging the Gap



A compiler translates from source code to a target language, typically machine code.

Example: C, C++, Haskell, Rust

Bridging the Gap

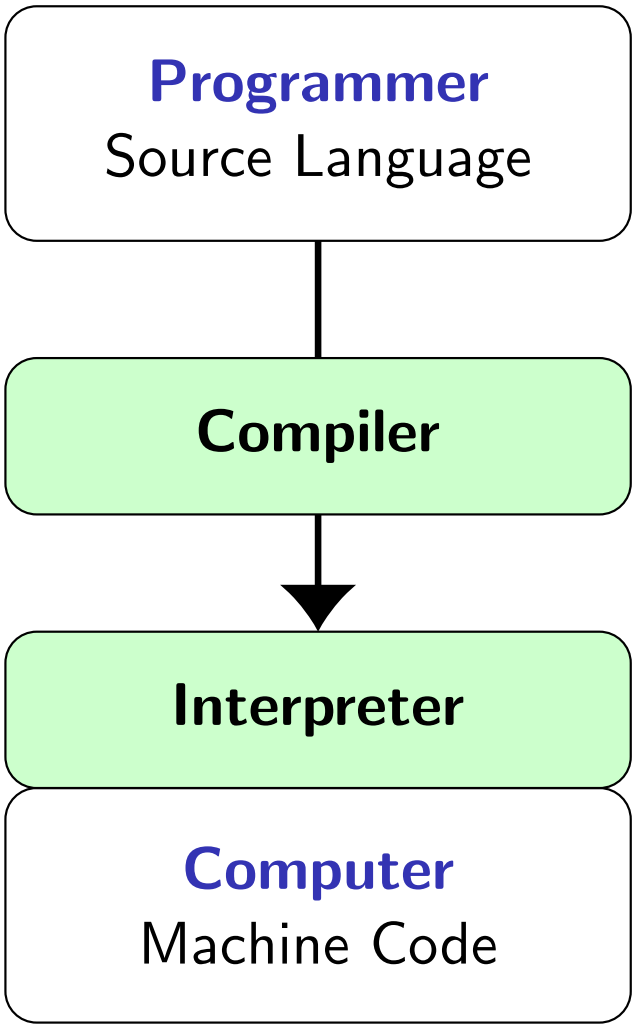


An interpreter executes a program as it reads the source code.

Examples: Perl, Python, JavaScript

JIT compilers complicate this picture somewhat.

Bridging the Gap



Some languages make use of a hybrid approach. First translating the source language to an intermediate language (abstract or virtual machine), then interpreting that.

Examples: Java, C#

Stages of a Compiler

The first stage of a compiler is called a *lexer*, which, given an input string of source code, produces a stream of *tokens* or *lexemes*, discarding irrelevant information like whitespace or comments.

Example (C)

```
int foo () {
  int i;
  i = 11;
  if (i > 5) {
    i = i - 1;
  }
  return i;
}
```

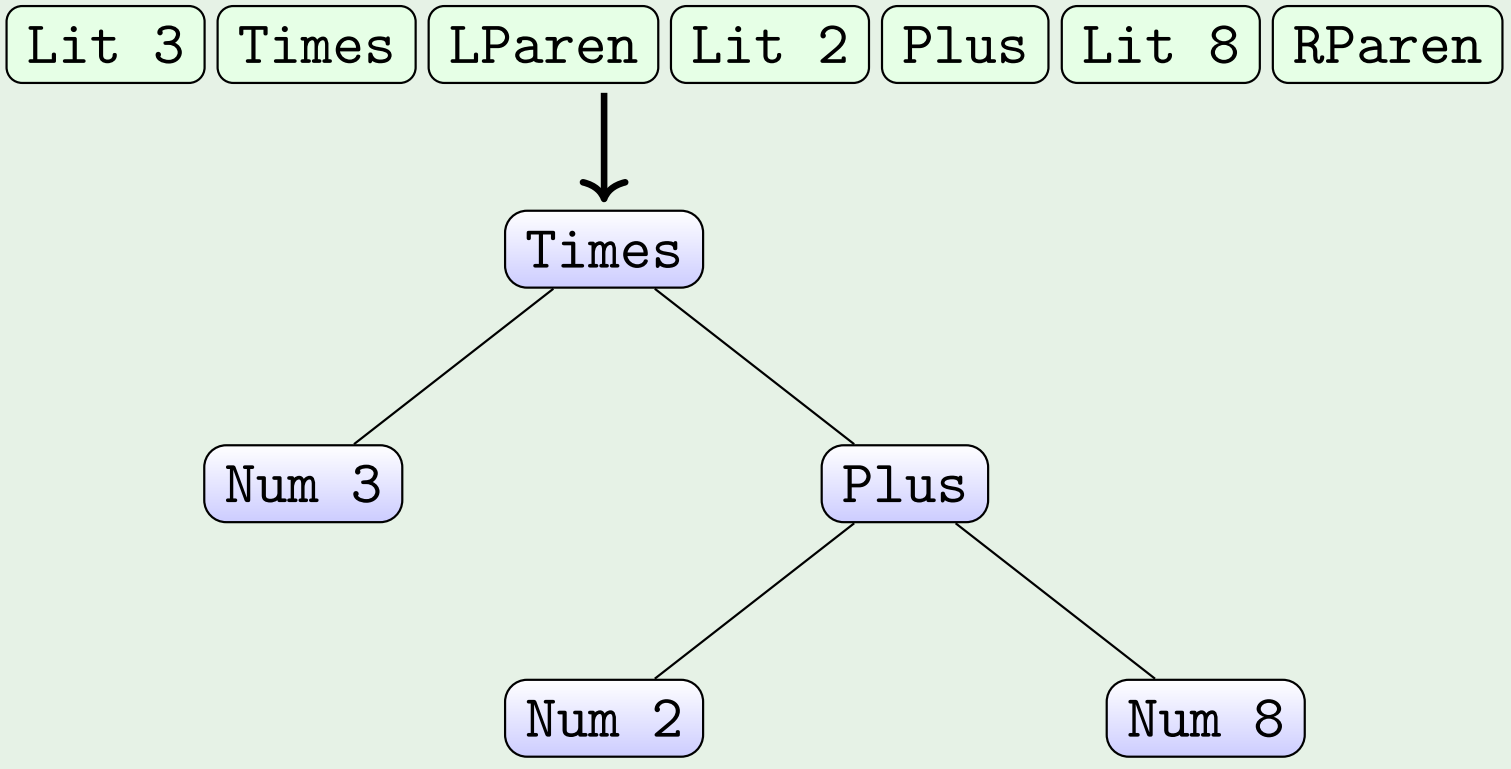
lexer ⟹

```
Ident "int"  Ident "foo"
LParen  RParen  LBrace
Ident "int"  Ident "i"  Semi
Ident "i"  . . .
```

Stages of a Compiler

A *parser* converts the stream of tokens from the lexer into a *parse tree* or *abstract syntax tree*:

Example (Arithmetic)



Grammars

The structure of lexemes expected to produce certain parse trees is called a *grammar*.

Example (Informal grammar for C)

C function definitions consist of:

- an identifier (return type), followed by
- an identifier (function name), followed by
- a possibly empty sequence of arguments, enclosed in parentheses, then
- a statement (function body)

Conclusions

This kind of definition is way too verbose and too imprecise to specify an implementation.

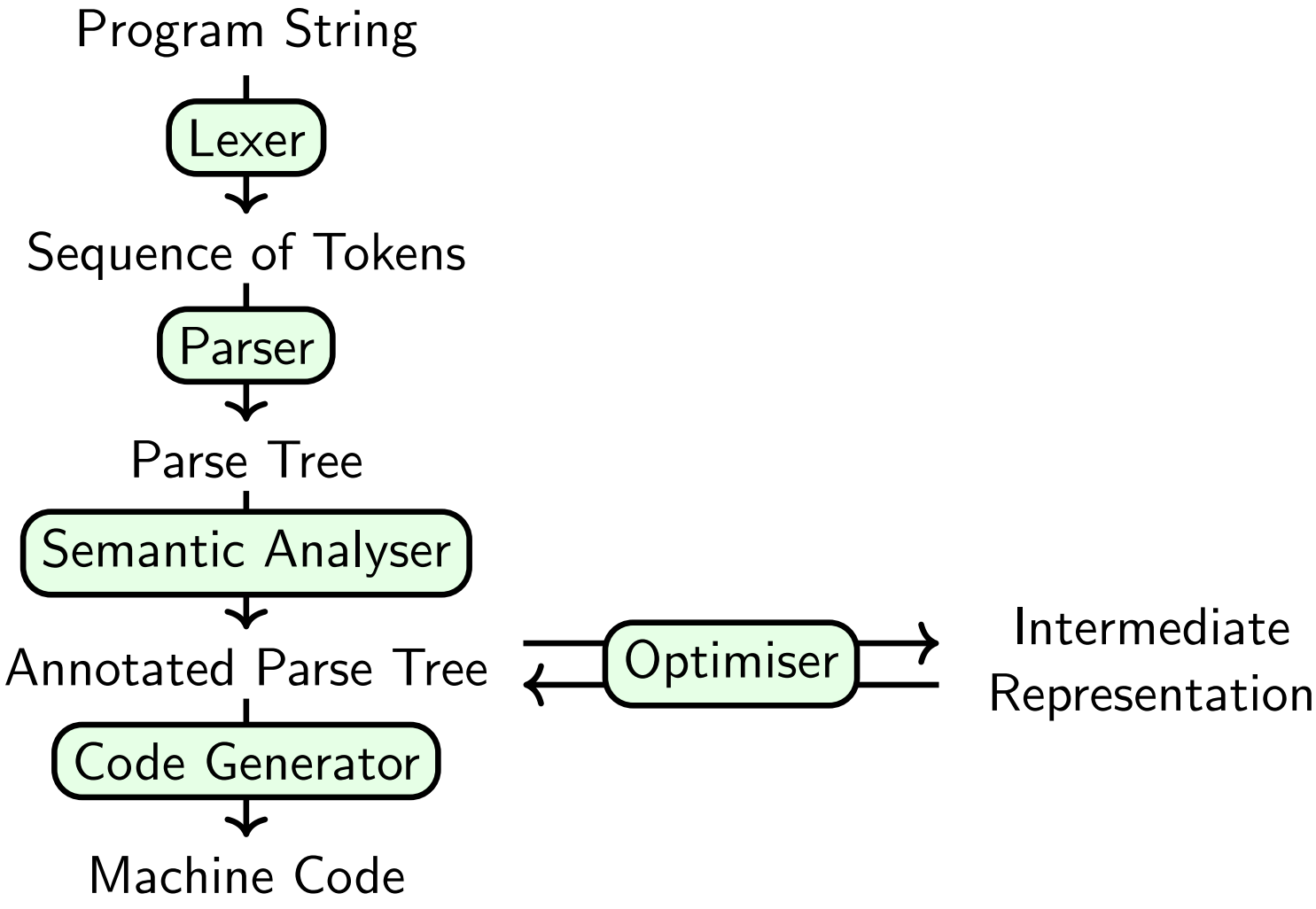
Backus-Naur Form

Specify grammatical productions by using a bare-bones recursive notation. *Non-terminals* are in *italics* whereas *terminals* are in **this typeface**.

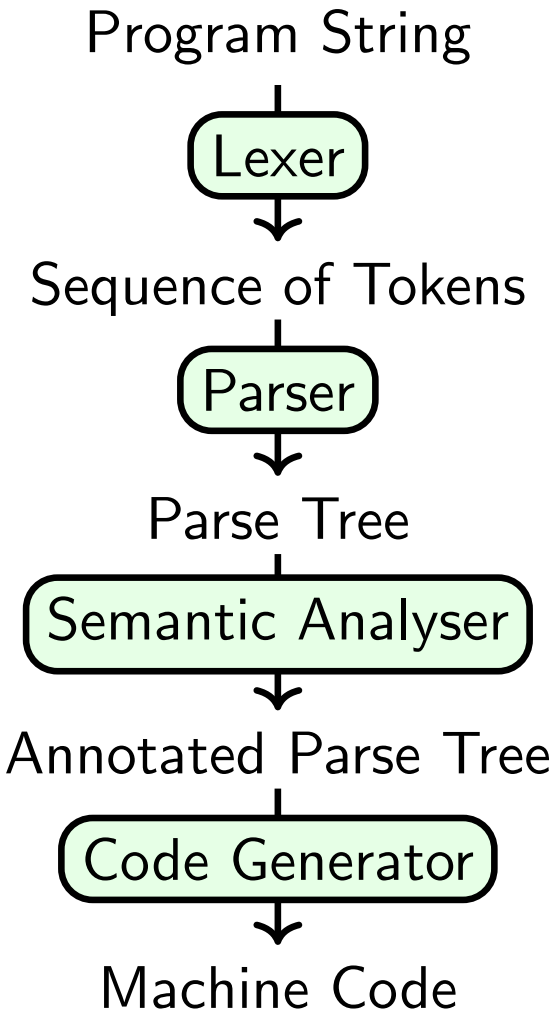
Example (C subset)

$$
\begin{array}{lcl}
\mathit{funDef} & ::= & \mathit{Ident}_1\ \mathit{Ident}_2\ \textbf{(}\ \mathit{args}\ \textbf{)}\ \mathit{stmt} \\
\mathit{stmt} & ::= & \mathit{expr}\ \textbf{;} \mid \textbf{if (}\ \mathit{expr}\ \textbf{)}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
& \mid & \textbf{return}\ \mathit{expr}\ \textbf{;} \mid \{\ \mathit{locDec}\ \mathit{stmts}\ \} \\
& \mid & \textbf{while (}\ \mathit{expr}\ \textbf{)}\ \mathit{stmt} \\
\mathit{stmts} & ::= & \varepsilon \mid \mathit{stmt}\ \mathit{stmts} \\
\mathit{expr} & ::= & \mathit{Number} \mid \mathit{Ident} \mid \mathit{expr}_1 + \mathit{expr}_2 \\
& \mid & \mathit{Ident} = \mathit{expr} \mid \mathit{Ident}\ \textbf{(}\ \mathit{expr}\ \textbf{)} \\
\mathit{locDec} & ::= & \mathit{Ident}_1\ \mathit{Ident}_2\ \textbf{;} \\
\mathit{args} & ::= & \varepsilon \mid \cdots
\end{array}
$$

Stages of a Compiler



Stages of a Compiler

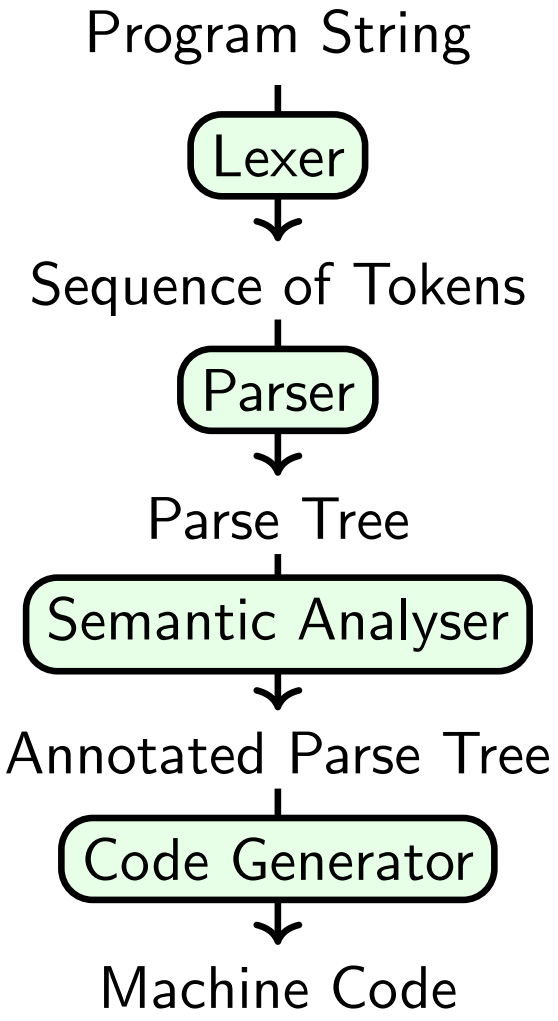


Intermediate Representation

Semantic Analysis

- Checks variable scoping
- Static semantics checks: most notably type checking.
- Adds extra information to the tree.

Stages of a Compiler

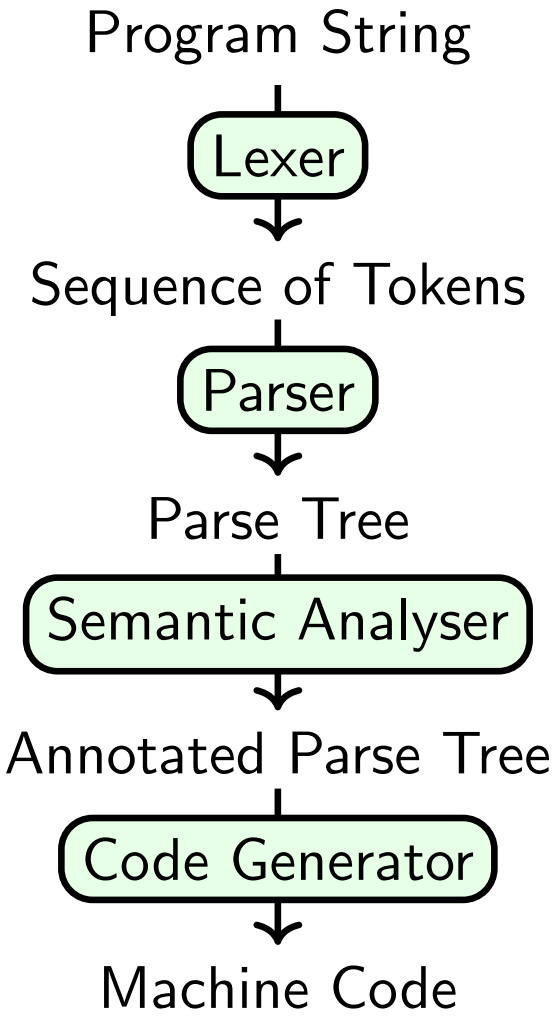


Intermediate Representation

Optimisation

- Loop unrolling, loop fusion
- Inlining, specialisation
- Sometimes transforms the tree dramatically.

Stages of a Compiler



Intermediate Representation

Code Generation

- Register allocation and explicit control flow.
- Links runtime system (e.g. GC)
- Selects appropriate machine instructions