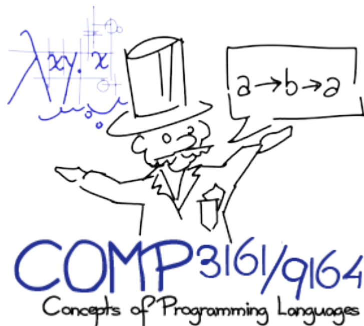


Introduction - 2025

Thomas Sewell Rob Sison
UNSW
Term 3 2025

Who are we?

Lecturers Thomas Sewell and Rob Sison.

Tutors Adam Stucci, Thomas Liang, Josh Lim, Kyle Zhang, Halogen Truong.

A lot of the material is inherited from previous convenors, including Johannes Åman Pohjola, Liam O'Connor, Christine Rizkallah and Gabriele Keller.

Contacting Us

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

Forum

Ask about course content on Discourse. A good forum helps everyone! We lecturers and tutors don't want to answer questions again and again.

Private or 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. No advanced Haskell is required, and we will do some demonstration exercises, but some self-study may be needed.

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. No advanced Haskell is required, and we will do some demonstration exercises, but some self-study may be needed.

Assessment

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

Tutorials

- Start next week.
- You may change tutorials, just seek approval first.
- Please attempt some of the questions beforehand.
- Tutes are 90 minutes! If the timetable says otherwise, ignore the timetable!

Assignment 0

- Focuses on the theory and proofs side of the course.
- It will be released in Week 3 and due in Week 4.

Assignments 1–2

- Build a compiler/interpreter component yourself.
- 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. Written lecture notes are made available throughout the trimester, 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..

Almost 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 Write?

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

However..

Almost 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 Write?

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

However..

Almost 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.

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.

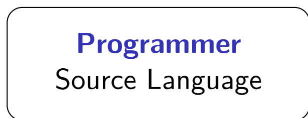
These tools are also very important for the pursuit of research in programming languages.

Why Haskell?

While are we foregrounding Haskell in this course?

- To ensure you have seen a diversity of programming languages; it's very unlike C.
- Reading & writing Haskell is close to reading & writing semantics.
- Functional languages are good for PL work.
- Haskell designers are enthusiastic PL adopters.
  - Adopts lots of new/interesting features before other languages.
- Much more of this in COMP3141.

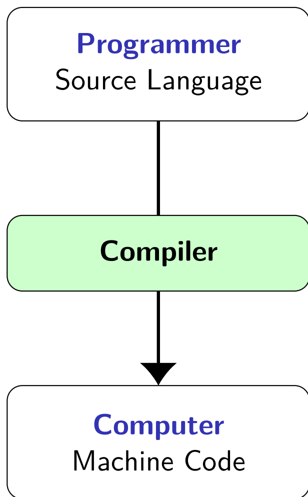
Bridging the Gap



Computers typically can't execute source code directly.

Computer
Machine Code

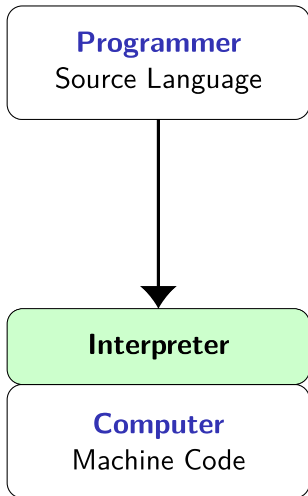
Bridging the Gap



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

Example: C, C++, Haskell, Rust

Bridging the Gap

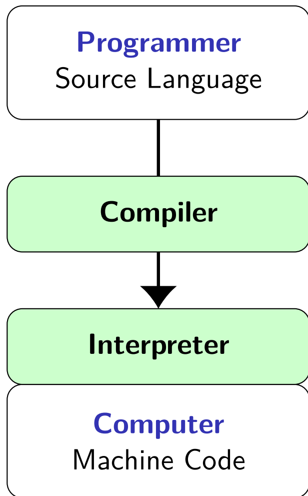


An interpreter executes a program as it reads the source code.

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

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)

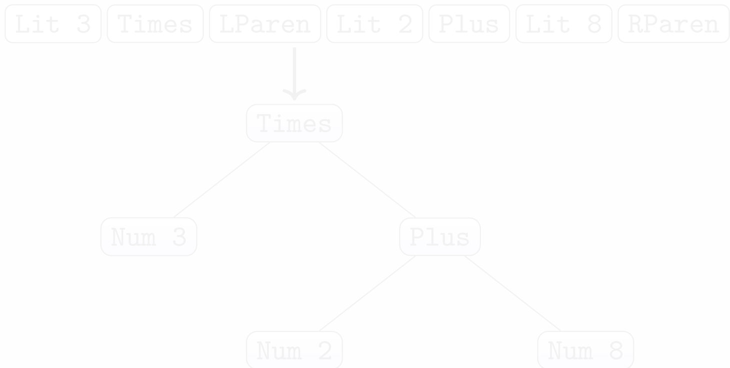
```
int foo () {
  int i;
  i = 11;
  if (i > 5) {
    i = i - 1;
  }
  return i;
}
```

lexer ⟹

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

Stages of a Compiler

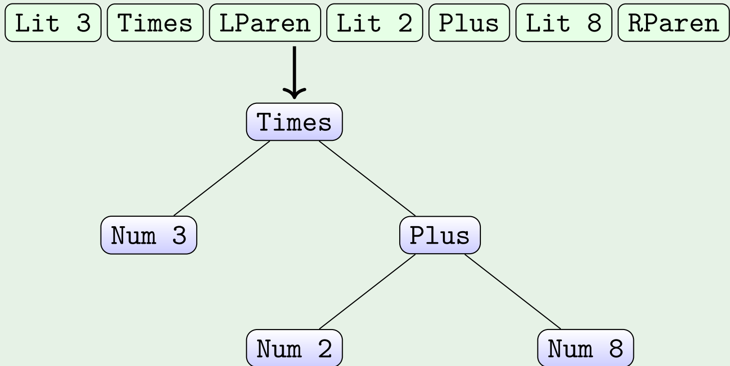
A *parser* converts the stream of tokens from the lexer into a *parse tree* or *abstract syntax tree*:



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.

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.

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.

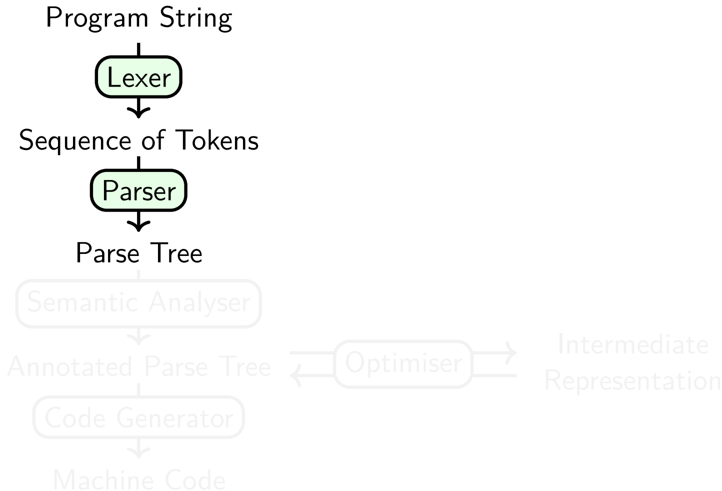
Backus-Naur Form

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

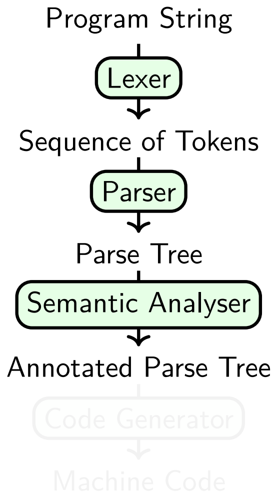
Example (C subset)

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

Stages of a Compiler



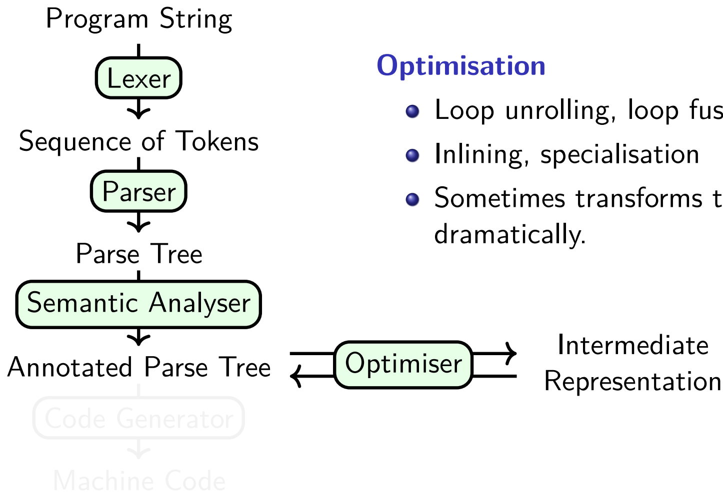
Stages of a Compiler



Semantic Analysis

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

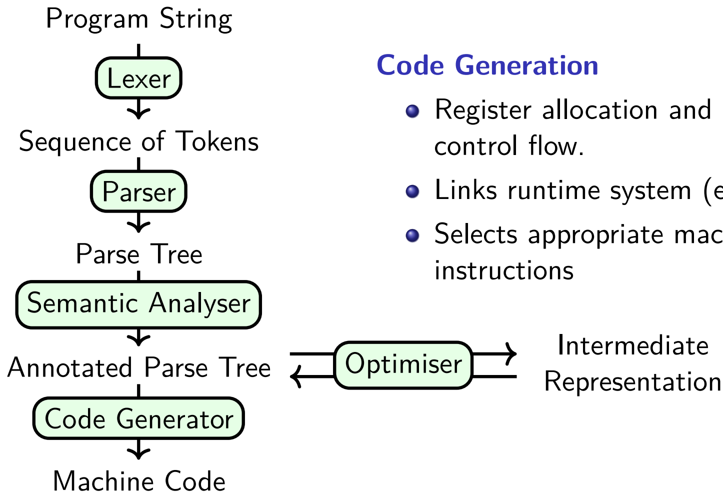
Stages of a Compiler



Optimisation

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

Stages of a Compiler



Code Generation

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

For the remainder of the lecture, we'll do a hands-on demo, to introduce Haskell to those who haven't seen it.

Let's try to implement a Haskell lexer for the C subset on the previous slides.

(If you're just reading the slides, you'll have to look elsewhere for this material.)