

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Who are we?

I am **Liam O'Connor**, a casual academic at UNSW. I do research work on formal methods and programming languages with various companies and the people at data61.

Michael Sproul, who will be running many of the tutorials, is a proof engineer at data61 working on formally-verified operating systems and compilers.

Gabriele Keller, who now works at Utrecht University, is the former lecturer of this course. Her research interests revolve around programming languages for formal methods and high performance computing. Hopefully we can maintain the high standard she set.

Who are we?

I am **Liam O'Connor**, a casual academic at UNSW. I do research work on formal methods and programming languages with various companies and the people at data61.

Michael Sproul, who will be running many of the tutorials, is a proof engineer at data61 working on formally-verified operating systems and compilers.

Gabriele Keller, who now works at Utrecht University, is the former lecturer of this course. Her research interests revolve around programming languages for formal methods and high performance computing. Hopefully we can maintain the high standard she set.

Who are we?

I am **Liam O'Connor**, a casual academic at UNSW. I do research work on formal methods and programming languages with various companies and the people at data61.

Michael Sproul, who will be running many of the tutorials, is a proof engineer at data61 working on formally-verified operating systems and compilers.

Gabriele Keller, who now works at Utrecht University, is the former lecturer of this course. Her research interests revolve around programming languages for formal methods and high performance computing. Hopefully we can maintain the high standard she set.

Contacting Us

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

Forum

There is a **Piazza** 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. I highly recommend disabling the Piazza Careers rubbish.

Administrative questions should be sent to `liamoc@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.

Programming

We expect you to be familiar with C and at least one other programming language. Course assignments are in Haskell, but Haskell knowledge is not assumed. Only very simple Haskell is required, 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.

Programming

We expect you to be familiar with **C** and at least one other programming language. Course assignments are in **Haskell**, but Haskell knowledge is not assumed. Only very simple Haskell is required, but some self-study may be needed.

Assessment

class

Assignment 2 35%	
Assignment 1 30%	
Midsem 35%	Midsem 25%
	Tutes 10%

9161

3161

$$mark = \frac{2 \times class \times exam}{class + exam}$$

Assessment

class

Assignment 2 35%	
Assignment 1 30%	
Midsem 35%	Midsem 25%
	Tutes 10%

9161

3161

Mid-semester Exam

- We will try to run in Week 5.
- Lasts for 50 minutes.
- Runs in lecture time slot.
- Web stream students **must** attend!
- Still organising venues.

Assessment

class

Assignment 2 35%	
Assignment 1 30%	
Midsem 35%	Midsem 25%
	Tutes 10%

9161

3161

Assignments

- Given a formal specification, implement in Haskell.
- Released around Week 5 and Week 9
- Approximately 3-4 weeks to complete each assignment.
- 10% penalty for one day late, 25% for two, 50% for three and 100% for four+.

Lectures

- There is one stream of lectures, and a (hurk!) web stream.
- Web stream students **must** watch recordings as they come out.
- Recordings are available through echo 360. I will to my best to make these usable.
- All board-work will be done digitally and made available to you.
- Separate lecture notes are also published.
- The second hour on Thursday is reserved for **extension** and **revision** topics. Please make suggestions!

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. These 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 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 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](#)
- 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.

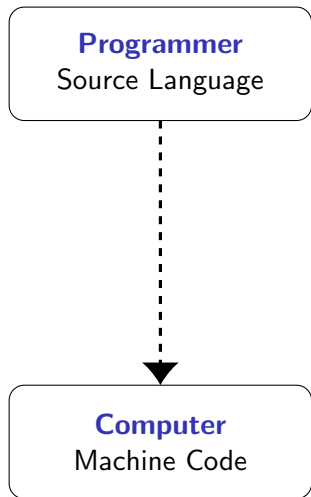
Why Reason?

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](#)
- Develop algorithms to find bugs automatically. [COMP3153](#)
- Rigorously analyse optimisations and other program transformations.

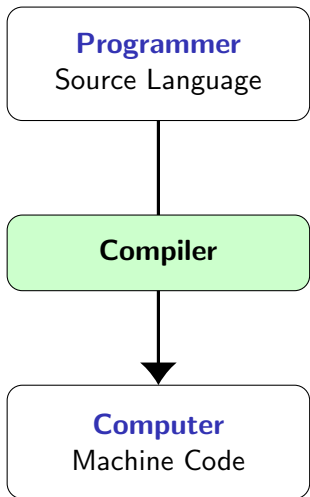
These tools are also very important for the pursuit of research in programming languages.

Bridging the Gap



Computers can't typically execute source code directly.

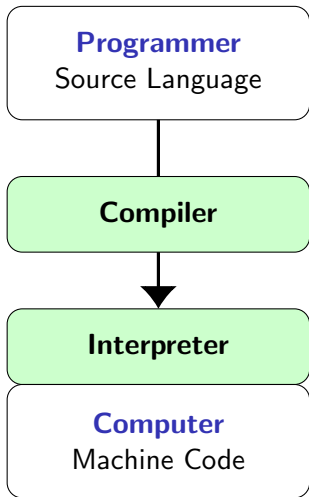
Bridging the Gap



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

Example: C, C++, Haskell, Rust

Bridging the Gap



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

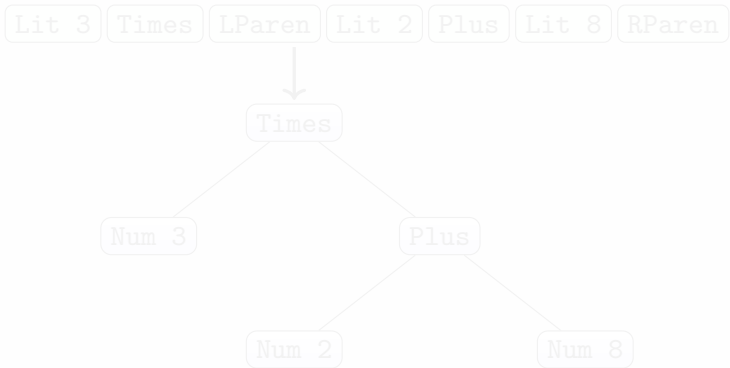
Ident "i"

...

Stages of a Compiler

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

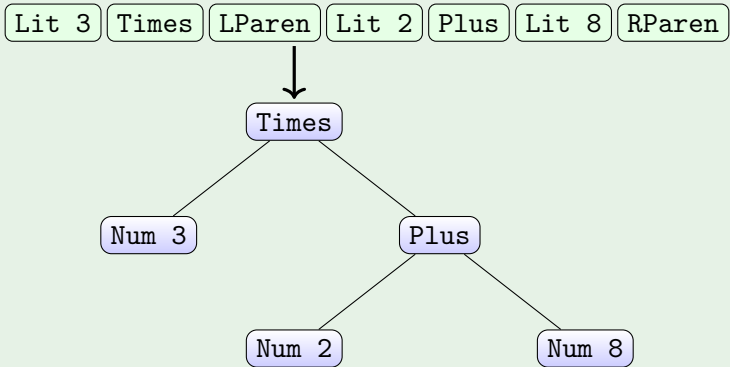
Example (Arithmetic)



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

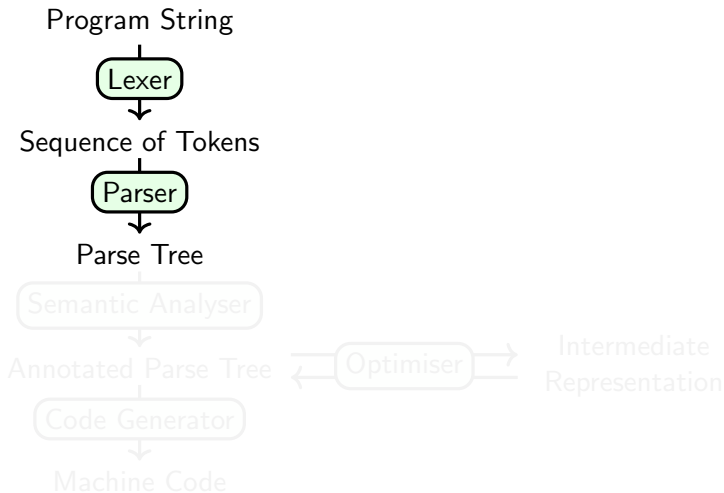
Specify grammatical productions by using a bare-bones recursive notation. *Non-terminals* are in *italics* whereas *terminals* are in **this typeface**.

Example (C subset)

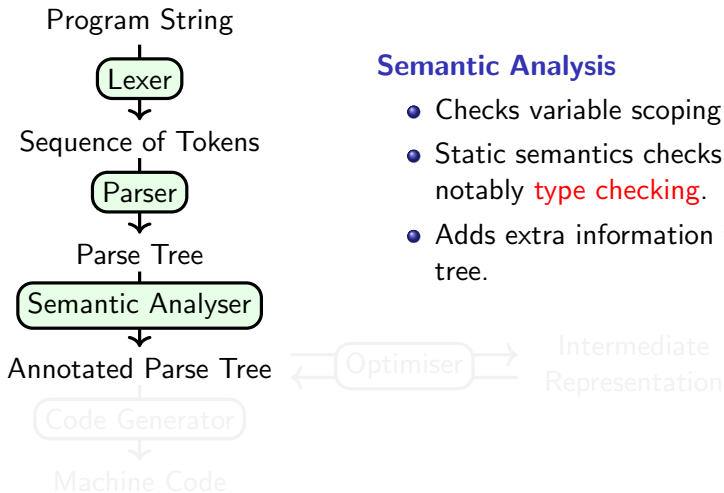
```

funDef ::= Ident1 Ident2 ( args ) stmt
stmt   ::= expr ; | if ( expr ) stmt else stmt
          | return expr ; | { locDec stmts }
          | while ( expr ) stmt
stmts ::= ε | stmt stmts
expr  ::= Number | Ident | expr1 + expr2
          | Ident = expr | Ident ( expr )
locDec ::= Ident1 Ident2 ;
args   ::= ε | ⋯
  
```


Stages of a Compiler



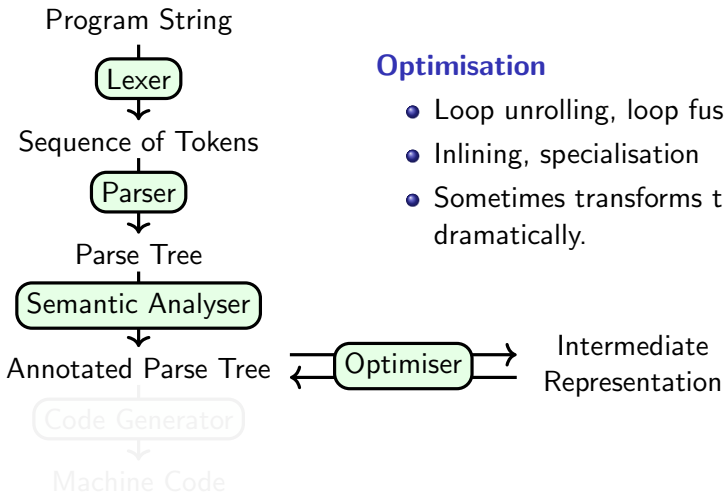
Stages of a Compiler



Semantic Analysis

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

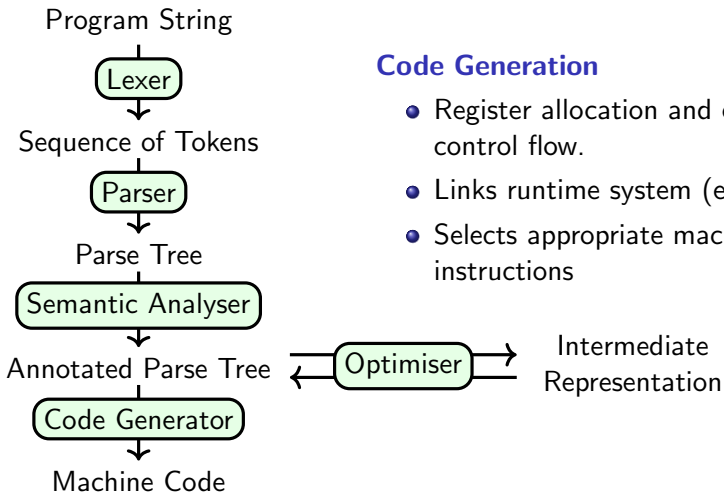
Stages of a Compiler



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

Homework

- There are no tutes in Week 1.
- As a substitute, we have set some discrete math exercises to help you revise your logic and proof skills, heading in to the first tute in Week 2. If you get stuck, feel free to ask your tutor for help.