COMP3048: Lecture 1

Administrative Details 2018
and
Introduction to Compiler Construction

Matthew Pike

University of Nottingham, Ningbo

Finding People and Information (1)

- Matthew Pike
  - Office: PMB-435
  - e-mail: matthew.pike@nottingham.edu.cn

Finding People and Information (2)

- Moodle:
  - `moodle.nottingham.ac.uk/course/view.php?id=61556`
- Direct questions concerning lectures and coursework to the ***Moodle COMP3048 Forum***.

  Anyone can ask and answer questions, but you must not post exact solutions to the coursework.

Aims and Motivation (1)

- ***Aims:*** Deepened understanding of:
  - how compilers (and interpreters) work and are constructed
  - programming language design and semantics
- The former is a great, “hands on”, “learning-by-doing” way to learn the latter.

Aims and Motivation (2)

- ***Why?*** Because programming languages and tools are at the very core of Computer Science and Software Engineering.
  - There are 100s of programming languages . . .
  - you cannot learn them all.
  - But thoroughly understanding how languages work in general is a shortcut to:
    - learning new languages as needed
    - or even using unfamiliar ones
    - using the languages you know much more effectively.

Aims and Motivation (3)

- ***Moreover:*** Compilers: “a microcosm of computer science” [CT04]
  - Formal Languages and Automata Theory
  - Datastructures and algorithms
  - Computer architecture
  - Programming language semantics
  - Formal reasoning about programs
  - Software engineering aspects

Aims and Motivation (4)

- Or, in terms of modules, COMP3048 directly draws from/informs:
  - ***COMP2049***: formal language theory, grammars, (D)FAs
  - ***COMP1034***: formal reasoning, structural induction
  - ***COMP1039***, ***COMP1038***: programming, understanding programming languages
  - ***COMP1036***: how computers work

Aims and Motivation (5)

- ***Jobs?*** There are plenty of companies out there with in-house languages or that critically rely on compiler/interpreter expertise for other reasons. Some possibly surprising examples:
  - Facebook - HipHop
  - Google - Go
  - Microsoft - .Net, Visual Studio

Aims and Motivation (6)

- ***Learning Outcomes***:
  - Knowledge of language and compiler design, semantics, key ideas and techniques.
  - Experience of compiler construction tools.
  - Experience of working with a medium-sized program.
  - Programming in various paradigms
  - Capturing design through formal specifications and deriving implementations from those.

Literature (1)

David A. Watt and Deryck F. Brown.
Programming Language Processors in Java,
Prentice-Hall, 1999.

- Used to be the main book. The lectures partly follow the structure of this book.
- The coursework was originally based on it.
- Hands-on approach to compiler construction. Particularly good if you like Java.
- Considers software engineering aspects.
- A bit weak on linking theory with practice.

Literature (2)

An alternative: Keith D. Cooper and Linda Torczon. *Engineering a Compiler*, Elsevier, 2004.

- Covers more ground in greater depth than this module.
- Gradually becoming the new main reference for the module.

For each lecture, there are references to the relevant chapter(s) of both books (see the COMP3048 Moodle page).

Literature (3)

Great supplement: Alfred V Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*, Addison-Wesley, 1986. (The “Dragon Book”.)

- Classic reference in the field.
- Covers much more ground in greater depth than this module.
- A book that will last for years.
- ***There is a New(-ish) 2007 edition!***

Literature (4)

Other useful references:

- Benjamin C. Pierce. *Types and Programming Languages*.
- Graham Hutton. *Programming in Haskell*.

Lectures and Handouts

- Come prepared to take notes. There will be some handouts, but for the most part not.
- All ***electronic*** slides, program code, and other supporting material in ***electronic*** form used during the lectures, will be made available on the course web page.
- ***However!*** The electronic record of the lectures is neither guaranteed to be complete nor self-contained!

Assessment (1)

First sit:

- The exam counts for 75 % of the total mark.
- The coursework counts for the remaining 25 %.
- 2 h exam, 3 questions, each worth 25 %. (No optional questions: be aware of this if you look at past exam papers.)

Bonus! There will be (sub)question(s) on the exam closely related to the coursework!

Effectively, the weight of the coursework is thus more like 50 %, except partly examined later.

Assessment (2)

Resit:

- The exam counts for 100 % of the total mark.
- 2 h exam, 4 questions, each worth 25 %
- ***The coursework does* not *count***.

Assessment (3)

Why this structure?

- Compiler construction is best learnt by doing.
- Thus, if you do and understand the coursework, you will be handsomely rewarded.
- Past experience shows that students who don't engage with the coursework struggle to pass.

Coursework (1)

Learning goals:

- Getting a practical understanding of key concepts and techniques in the fields of compiler construction and language theory.
- Getting hands-on experience of working with language processors.
- Getting some experience of using compiler construction tools.
- Getting experience of the issues involved in working with medium-sized programs.

Coursework (2)

You will be given partial implementations of a compiler for a small language called ***MiniTrinagle***.

You will be asked to:

- answer theoretical questions related to the code
- extend the code with new features.

Detailed instructions for the coursework available (soon) from the course web page.
Study these instructions very carefully!

Haskell and Coursework Support (1)

The functional language Haskell is used throughout the module as:

- An ideal language for illustrating and discussing all aspects of compiler construction (and similar applications).
- Functional language notation is closely aligned with mathematical notation and formalisms (such as attribute grammars) commonly used in text books on compilers.
- A really good choice for implementing compilers in practice (and much else beside).

Haskell and Coursework Support (2)

To help you get up to speed using Haskell and provide some extra help with the coursework:

- Some of the lectures closely aligned with the coursework.
- Haskell revision coursework (unassessed).
- Slides from a Haskell revision lecture via module web page.

Laboratory Sessions

Laboratory sessions:

- Tuesdays, 9–11, PB-413

Coursework Assessment (1)

- Two parts to the coursework: I and II
- Each part to be solved ***individually***
- Submission for each part:
  - Brief written report (hard copy & PDF)
  - All source code (electronically)
- For part II, ***compulsory*** 10 minute oral examination in assigned slot during one of the lab sessions after the submission deadline.
- Catch-up slots ***only*** if missed slot with good cause; personal tutor to request on your behalf.

Coursework Assessment (2)

- A number of weighted questions for each part
- Written answer to each question assessed on
  - Correctness (0, 1, or 2 marks)
  - Style (0, 1, or 2 marks)
- In the oral examination (part II only), you ***explain*** your answers.
- Your explanations are assessed as follows:
  - ***2***: 100 % of mark for written answer
  - ***1***: 65 % of mark for written answer
  - ***0***: 0 % of mark for written answer

Coursework Deadlines - Tentative

Preliminary deadlines (to be confirmed shortly - on Moodle). Moodle is the only authorative (and therefore correct) reference for coursework deadlines. The below is to give you a time scale.

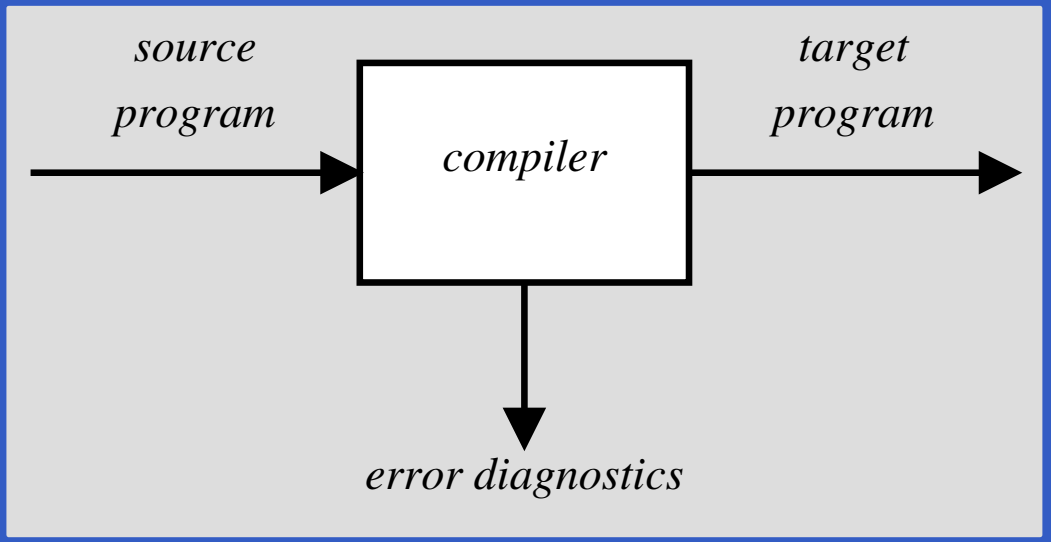
- Part I: Wednesday ***29 October***.
- Part II: Wednesday ***26 November***.

Oral examinations during the lab sessions the following two Tuesdays; i.e. ***2*** and ***9 December***.

Start early! It is ***not*** possible to do this coursework at the last minute.

What is a Compiler? (1)

Compilers are ***program translators***:



Typical example:

- Source language: C
- Target language: x86 assembler

Why? To make it easier to program computers!

What is a Compiler? (2)

GCC translates this C program

```
int main(int argc, char *argv) {
    printf("%d\n", argc - 1);
}
```

into this x86 assembly code (excerpt):

```
movl    8(%ebp), %eax
decl    %eax
subl    $8, %esp
pushl   %eax
pushl   $.LC0
call    printf
addl    $16, %esp
```

Source and Target Languages

Large spectrum of possibilities, for example:

- Source languages:
  - (High-level) programming languages
  - Modelling languages
  - Document description languages
  - Database query languages
- Target languages:
  - High-level programming language
  - Low-level programming language (assembler or machine code, byte code)

Where are Compilers Used? (1)

- Implementation of programming languages: C, C++, Java, C#, Haskell, Lisp, Prolog, Ada, Fortran, Cobol, ...
- Document processing: LaTeX $\rightarrow$ DVI, DVI $\rightarrow$ PostScript/PDF/...
- Databases: optimization of database queries expressed in query languages like SQL.

Where are Compilers Used? (2)

- Hardware design: modelling and simulation/verification, compilation to silicon. E.g. Spice, VHDL.
- Modelling and simulation of physical systems (cars, trains, aircraft, nuclear power plants, . . .): Simulink, Modelica.
- In Web browsers to speed up execution of code embedded in web pages, applets, Rich Internet Applications (RIA), etc.
- . . .

Compilers vs. Interpreters

Interpreters are another class of translators:

- Compiler: translates a program once and for all into target language.
- Interpreter: effectively translates (the used parts of) a source program every time it is run.
- Techniques like ***Just-In-Time Compilation*** (JIT) blurs this distinction.
- Compilers and interpreters sometimes used together, e.g. Java: Java compiled into Java byte code, byte code interpreted by a Java Virtual Machine (JVM), JVM might use JIT.

Inside the Compiler (1)

Traditionally, a compiler is broken down into several phases:

- ***Scanner***: lexical analysis
- ***Parser***: syntactic analysis
- ***Checker***: contextual analysys (e.g. type checking)
- ***Optimizer***: code improvement
- ***Code generator***

Inside the Compiler (2)

- Lexical Analysis:
  - Verify that input character sequence is lexically valid.
  - Group characters into sequence of lexical symbols, ***tokens***.
  - Discard white space and comments (typically).

Inside the Compiler (3)

- Syntactic Analysis/Parsing
  - Verify that the input program is syntactically valid, i.e. conforms to the ***Context Free Grammar*** of the language.
  - Determine the program structure.
  - Construct a representation of the program reflecting that structure without unnecessary details, usually an ***Abstract Syntax Tree*** (AST).

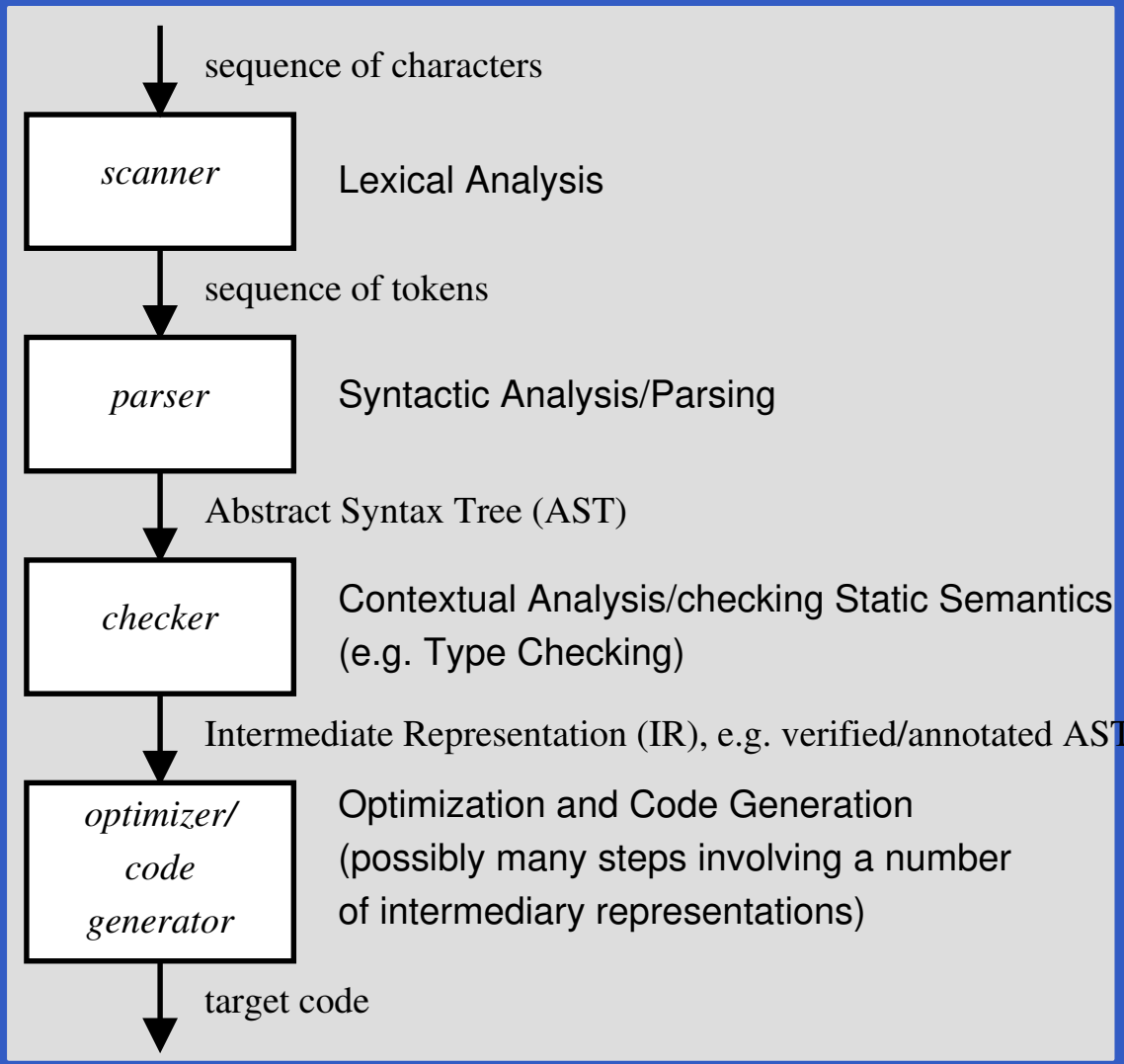
Inside the Compiler (4)

- Contextual Analysis/Checking Static Semantics:
  - Resolve meaning of symbols.
  - Report undefined symbols.
  - Type checking.
  - ...
- Optimization:
  - Code improvements aiming at making it run faster and/or use less space, energy, etc.
  - Almost always ***heuristics***: cannot ***guarantee*** optimal result.

Inside the Compiler (5)

- Code Generation:
  - Output the appropriate sequence of target language instructions.
  - Might involve further ***low-level*** (target-specific) optimization.

Inside the Compiler (6)



Inside the Compiler (7)

- ***Front end***: Scanner, Parser, Contextual checker.
  Depends more heavily on the ***source language***.
- ***Middle section*** ("Middle end"): Optimizer.
  Operates on an Intermediary Representation (IR) that could be fairly independent of source and target language. Hence potentially reusable!
- ***Back end***: Code Generator
  Depends heavily on the ***target language***.