COMP3012/G53CMP: Lecture 1

Administrative Details 2018
and
Introduction to Compiler Construction

Henrik Nilsson

University of Nottingham, UK

Finding People and Information (1)

- Henrik Nilsson
  Room A08, Computer Science Building
  e-mail: `nhn@cs.nott.ac.uk`
- Teaching Assistants:

	www.cs.nott.ac.uk/
Martin Handley	~psxmah
Guerric Chupin	~psxgc4
Jennifer Hackett	~psxjlha

Finding People and Information (2)

- Main module web page:
  `www.cs.nott.ac.uk/~psznhn/G53CMP`
- Moodle: `moodle.nottingham.ac.uk/`course/view.php?id=68635
- Direct questions concerning lectures and coursework to the ***Moodle G53CMP Forum***.

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

Notes on Lectures 2018

- Two lectures on Thursdays, 16:00–18:00
- Note: A24 first hour, A06 second hour
- Always a break 16:50–17:00
- ***No lectures on 11 October!***

Aims and Motivation (1)

Why study Compiler Construction?

- Why did you opt to take this module?
- More generally, what do you think are good reasons to take this module?

Aims and Motivation (2)

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 (3)

Why?

The ACM/IEEE 2013 CS Curriculum Guidelines:

Aims and Motivation (4)

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

Thus, “capstone” module tying everything together.

Aims and Motivation (5)

Or, in terms of modules, G53CMP directly draws from/informs:

- *G52LAC*: formal language theory, grammars, (D)FAs
- *G51MCS*, *G52ACE*: formal reasoning, structural induction
- *G51PGA*, *G51PGP*: programming, understanding programming languages
- *G51CSF*, *G51CSA*: how computers work
- *G54FOP/FPP*: programming language theory

Aims and Motivation (6)

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
- Standard Chartered Bank
- Jane Street

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 lecture overview on the G53CMP web 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*.

Books seem a bit old?

Sure! They focus on core principles of lasting value that it pays off to learn.

Cf. ACM/IEEE 2013 Curriculum Guidelines

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!

Medium of Instruction

Haskell used as medium of instruction 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 used in text books on compilers.
- In practice, often a ***good choice for implementing*** compilers (and much else beside).

Assessment

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 %.

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.

Resit: ***100 % exam***

Assessment (2)

Why such emphasis on the coursework?

- 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

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 from the module web page (Part I: 17 Oct.). ***Study these instructions very carefully!***

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

Coursework deadlines:

- Part I: Monday ***5 November***, 15:00.
- Part II: Monday ***3 December***, 15:00.

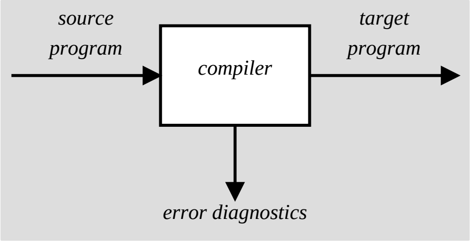
Oral examinations during the lab sessions the following two Fridays; i.e. ***7*** and ***14 December***.

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

First lab session: Friday 19 October, 13:00–15:00.

What is a Compiler? (1)

Compilers are ***program translators***:



source program

compiler

target program

error diagnostics

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)

Compilers vs. Interpreters

Interpreters are another class of translators:

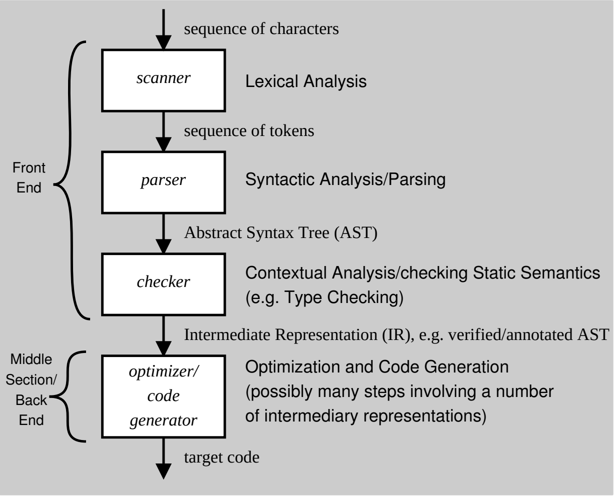
- Compiler: translates a program once and for all into target language.
- Interpreter: effectively translates (the used parts of) a source program every time it is run.
- Techniques like ***Just-In-Time Compilation*** (JIT) blurs this distinction.
- Compilers and interpreters sometimes used together, 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)



sequence of characters

scanner — Lexical Analysis

sequence of tokens

parser — Syntactic Analysis/Parsing

Abstract Syntax Tree (AST)

checker — Contextual Analysis/checking Static Semantics (e.g. Type Checking)

Front End

Intermediate Representation (IR), e.g. verified/annotated AST

optimizer/ code generator — Optimization and Code Generation (possibly many steps involving a number of intermediary representations)

Middle Section/ Back End

target code

Inside the Compiler (3)

- 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 (4)

- 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).

Example: TXL into C compiler

Scenario:

- We wish to develop a compiler for TXL: *Trivial eXpression Language*.
- To save ourselves some effort, we are going to compile TXL into C, and then use an existing C compiler (GCC) to translate into executable machine code.

Informal TXL Syntax and Semantics

Some examples of TXL programs, ***concrete syntax***, and their intended meaning, ***semantics***:

- `1 + 3`
  Semantics: *4*
- `1 + (3 * (2 + 2))`
  Semantics: *13*
- `let x = 3 * 7 in x + 3`
  Semantics: *24*

This is ***dynamic semantics***: what does a program mean when run?

Inside the Compiler (5)

- Contextual Analysis/Checking Static Semantics:
  - Resolve meaning of symbols.
  - Report undefined symbols.
  - Type checking.
  - . . .

Informal TXL Syntax and Semantics

- `let x = 3 * 7 in let x = x * 3 in x - 21`
  Semantics: *???*

Some ***static semantics*** possibilities:

- Disallow re-definition of entities already in scope.
- Allow nested scopes, decide how to disambiguate; e.g., closest containing scope.
- Recursive definitions or not? I.e., is the defined entity in scope in its own definition?

We opt for nesting, closest containing scope, no recursion. (Dynamic) semantics: *42*

Informal TXL Syntax and Semantics

- `let x = 3 in y + x`
  Semantics: *???*

Some possibilities:

- Insist all variables be defined. The program can then be ***statically*** rejected as ***meaningless***.
- Catch use of undefined variables ***dynamically***, making the meaning of the program ***undefined***.
- Assume some default value, like 0, for variables that are not explicitly defined.

Inside the Compiler (5)

- Optimization:
  - Code improvements aiming at making it run faster and/or use less space, energy, etc.
  - Almost always ***heuristics***: cannot ***guarantee*** optimal result.
- Code Generation:
  - Output the appropriate sequence of target language instructions.
  - Might involve further ***low-level*** (target-specific) optimization.