Compilers and computer architecture: introduction

Martin Berger [1]

Thanks to Chad MacKinney, Alex Jeffery, Justin Crow, Jim Fielding, Shaun Ring and Vilem Liepelt for suggestions and corrections. Thanks to Benjamin Landers for the RARS simulator.
Thanks to Alex Aiken for his Compiler MOOC that this course was heavily inspired by.

September 2019

[1] Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312.

Administrative matters: lecturer

Administrative matters: lecturer

- Name: Martin Berger

Administrative matters: lecturer

- Name: Martin Berger
- Email: `M.F.Berger@sussex.ac.uk`

Administrative matters: lecturer

- Name: Martin Berger
- Email: `M.F.Berger@sussex.ac.uk`
- Web: `https://users.sussex.ac.uk/~mfb21/compilers`

Administrative matters: lecturer

- Name: Martin Berger
- Email: `M.F.Berger@sussex.ac.uk`
- Web: `https://users.sussex.ac.uk/~mfb21/compilers`
- Lecture notes etc: `https://users.sussex.ac.uk/~mfb21/compilers/material.html` Linked from Canvas

Administrative matters: lecturer

- Name: Martin Berger
- Email: `M.F.Berger@sussex.ac.uk`
- Web: `https://users.sussex.ac.uk/~mfb21/compilers`
- Lecture notes etc: `https://users.sussex.ac.uk/~mfb21/compilers/material.html` Linked from Canvas
- Office hour: after the Wednesdays lectures, and on request (please arrange by email, see `https://users.sussex.ac.uk/~mfb21/cal` for available time-slots)

Administrative matters: lecturer

- Name: Martin Berger
- Email: `M.F.Berger@sussex.ac.uk`
- Web: `https://users.sussex.ac.uk/~mfb21/compilers`
- Lecture notes etc: `https://users.sussex.ac.uk/~mfb21/compilers/material.html` Linked from Canvas
- Office hour: after the Wednesdays lectures, and on request (please arrange by email, see `https://users.sussex.ac.uk/~mfb21/cal` for available time-slots)
- My room: Chichester II, 312

Administrative matters: dates, times and assessment

Administrative matters: dates, times and assessment

- Lectures: Two lectures per week,
  Wednesday: 11-12 Lec PEV1-1A7
  Friday: 17-18 RICH-AS3

Administrative matters: dates, times and assessment

- Lectures: Two lectures per week,
  Wednesday: 11-12 Lec PEV1-1A7
  Friday: 17-18 RICH-AS3
- Tutorials: please see your timetables. The TA is Shaun Ring `sr410@sussex.ac.uk`

Administrative matters: dates, times and assessment

- Lectures: Two lectures per week,
  Wednesday: 11-12 Lec PEV1-1A7
  Friday: 17-18 RICH-AS3
- Tutorials: please see your timetables. The TA is Shaun Ring `sr410@sussex.ac.uk`
- There will (probably) be PAL sessions, more soon.

Administrative matters: dates, times and assessment

- Lectures: Two lectures per week,
  Wednesday: 11-12 Lec PEV1-1A7
  Friday: 17-18 RICH-AS3
- Tutorials: please see your timetables. The TA is Shaun Ring `sr410@sussex.ac.uk`
- There will (probably) be PAL sessions, more soon.
- Assessment: coursework (50%) and by unseen examination (50%). Both courseworks involve writing parts of a compiler. Due dates for courseworks: Fri, 8 Nov 2019, and Fri, 20 Dec 2019, both 18:00.

Questions welcome!

Questions welcome!

Please, ask questions ...

- during the lesson
- at the end of the lesson
- in my office hours (see `https://users.sussex.ac.uk/~mfb21/cal` for available time-slots)
- by email `M.F.Berger@sussex.ac.uk`
- on Canvas
- in the tutorials
- in the course's Discord channel (invite is on Canvas)
- any other channels (e.g. Telegram, TikTok ...)?

Please, don't wait until the end of the course to tell me about any problems you may encounter.

Prerequisites

Prerequisites

Good Java programming skills are indispensable.This course is **not** about teaching you how to program. “Good” in this context means you can do most questions on e.g.

`https://leetcode.com/`

classified as “Easy” without problems (= without looking up the answer, and in 1 hour or less). I also recommed that you familiarise yourself with the material on “Shell Tools and Scripting” and “Command-line Environment” in:

`https://missing.csail.mit.edu/`

Prerequisites

Good Java programming skills are indispensable.This course is **not** about teaching you how to program. “Good” in this context means you can do most questions on e.g.

`https://leetcode.com/`

classified as “Easy” without problems (= without looking up the answer, and in 1 hour or less). I also recommed that you familiarise yourself with the material on “Shell Tools and Scripting” and “Command-line Environment” in:

`https://missing.csail.mit.edu/`

It helps if you have already seen e.g. regular expressions, FSMs etc. But we will cover all this from scratch.

Prerequisites

Good Java programming skills are indispensable.This course is **not** about teaching you how to program. “Good” in this context means you can do most questions on e.g.

`https://leetcode.com/`

classified as “Easy” without problems (= without looking up the answer, and in 1 hour or less). I also recommed that you familiarise yourself with the material on “Shell Tools and Scripting” and “Command-line Environment” in:

`https://missing.csail.mit.edu/`

It helps if you have already seen e.g. regular expressions, FSMs etc. But we will cover all this from scratch.

It helps if you have already seen a CPU, e.g. know what a register is or a stack pointer.

Course content

Course content

I'm planning to give a fairly orthodox compilers course that shows you all parts of a compiler. At the end of this course you should be able to write a fully blown compiler yourself and implement programming languages.

Course content

I'm planning to give a fairly orthodox compilers course that shows you all parts of a compiler. At the end of this course you should be able to write a fully blown compiler yourself and implement programming languages.

We will also look at computer architecture, although more superficially.

Course content

I’m planning to give a fairly orthodox compilers course that shows you all parts of a compiler. At the end of this course you should be able to write a fully blown compiler yourself and implement programming languages.

We will also look at computer architecture, although more superficially.

This will take approximately 9 weeks, so we have time at the end for some advanced material. I’m happy to tailor the course to your interest, so please let me know what you want to hear about.

Coursework

Coursework

Evaluation of assessed courseworks will (largely) be by automated tests. This is quite different from what you've seen so far. The reason for this new approach is threefold.

Coursework

Evaluation of assessed courseworks will (largely) be by automated tests. This is quite different from what you've seen so far. The reason for this new approach is threefold.

- Compilers are complicated algorithms and it's beyond human capabilities to find subtle bugs.
- Realism. In industry you don't get paid for being nice, or for having code that "almost" works.
- Fairness. Automatic testing removes subjective element.

Coursework

Evaluation of assessed courseworks will (largely) be by automated tests. This is quite different from what you've seen so far. The reason for this new approach is threefold.

- Compilers are complicated algorithms and it's beyond human capabilities to find subtle bugs.
- Realism. In industry you don't get paid for being nice, or for having code that "almost" works.
- Fairness. Automatic testing removes subjective element.

Note that if you make a basic error in your compiler then it is quite likely that **every** test fails and you will get 0 points. So it is really important that you test your code before submission thoroughly. I encourage you to share tests and testing frameworks with other students: as tests are not part of the deliverable, you make share them. Of course the compiler must be written by yourself.

Plan for today's lecture

Plan for today's lecture

Whirlwind overview of the course.

Plan for today's lecture

Whirlwind overview of the course.

- Why study compilers?

Plan for today's lecture

Whirlwind overview of the course.

- Why study compilers?
- What is a compiler?

Plan for today's lecture

Whirlwind overview of the course.

- Why study compilers?
- What is a compiler?
- Compiler structure

Plan for today's lecture

Whirlwind overview of the course.

- Why study compilers?
- What is a compiler?
- Compiler structure
- Lexical analysis

Plan for today's lecture

Whirlwind overview of the course.

- Why study compilers?
- What is a compiler?
- Compiler structure
- Lexical analysis
- Syntax analysis

Plan for today's lecture

Whirlwind overview of the course.

- Why study compilers?
- What is a compiler?
- Compiler structure
- Lexical analysis
- Syntax analysis
- Semantic analysis, type-checking

Plan for today's lecture

Whirlwind overview of the course.

- Why study compilers?
- What is a compiler?
- Compiler structure
- Lexical analysis
- Syntax analysis
- Semantic analysis, type-checking
- Code generation

Why study compilers?

Why study compilers?

To become a good programmer, you need to understand what happens ’under the hood’ when you write programs in a high-level language.

Why study compilers?

To become a good programmer, you need to understand what happens 'under the hood' when you write programs in a high-level language.

To understand low-level languages (assembler, C/C++, Rust, Go) better. Those languages are of prime importance, e.g. for writing operating systems, embedded code and generally code that needs to be fast (e.g. computer games, ML e.g. TensorFlow).

Why study compilers?

To become a good programmer, you need to understand what happens 'under the hood' when you write programs in a high-level language.

To understand low-level languages (assembler, C/C++, Rust, Go) better. Those languages are of prime importance, e.g. for writing operating systems, embedded code and generally code that needs to be fast (e.g. computer games, ML e.g. TensorFlow).

Most large programs have a tendency to embed a programming language. The skill quickly to write an interpreter or compiler for such embedded languages is invaluable.

Why study compilers?

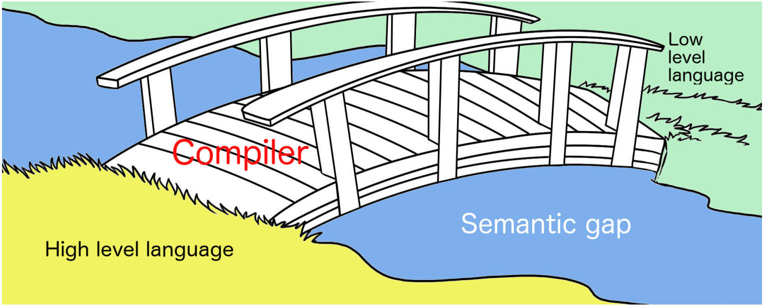
To become a good programmer, you need to understand what happens ’under the hood’ when you write programs in a high-level language.

To understand low-level languages (assembler, C/C++, Rust, Go) better. Those languages are of prime importance, e.g. for writing operating systems, embedded code and generally code that needs to be fast (e.g. computer games, ML e.g. TensorFlow).

Most large programs have a tendency to embed a programming language. The skill quickly to write an interpreter or compiler for such embedded languages is invaluable.

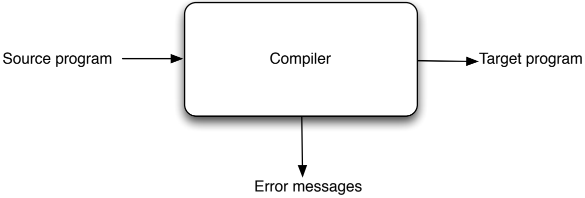
But most of all: compilers are extremely amazing, beautiful and one of the all time great examples of human ingenuity. After 70 years of refinement compilers are a paradigm case of beautiful software structure (modularisation). I hope it inspires you.

Overview: what is a compiler?



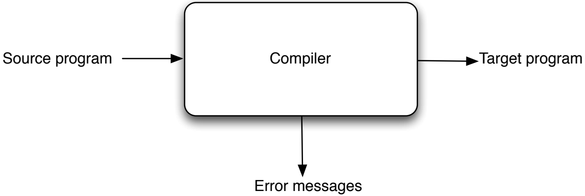
Overview: what is a compiler?

A compiler is a program that translates programs from one programming language to programs in another programming language. The translation should preserve meaning (what does "preserve" and "meaning" mean in this context?).



Overview: what is a compiler?

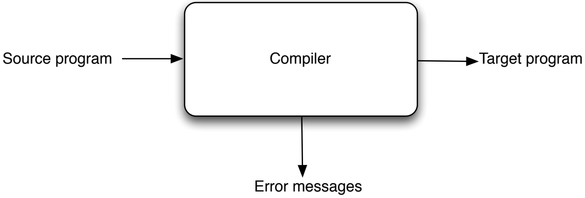
A compiler is a program that translates programs from one programming language to programs in another programming language. The translation should preserve meaning (what does “preserve” and “meaning” mean in this context?).



Typically, the input language (called source language) is more high-level than the output language (called target language)

Overview: what is a compiler?

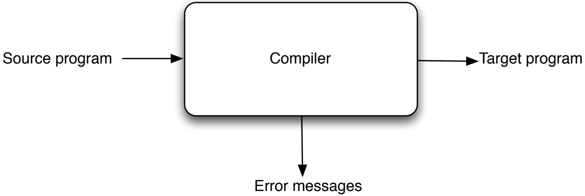
A compiler is a program that translates programs from one programming language to programs in another programming language. The translation should preserve meaning (what does “preserve” and “meaning” mean in this context?).



Typically, the input language (called source language) is more high-level than the output language (called target language)
Examples

Overview: what is a compiler?

A compiler is a program that translates programs from one programming language to programs in another programming language. The translation should preserve meaning (what does "preserve" and "meaning" mean in this context?).



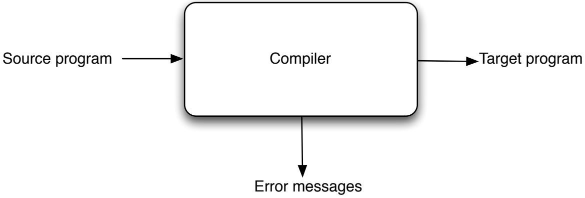
Typically, the input language (called source language) is more high-level than the output language (called target language)

Examples

- Source: Java, target: JVM bytecode.

Overview: what is a compiler?

A compiler is a program that translates programs from one programming language to programs in another programming language. The translation should preserve meaning (what does “preserve” and “meaning” mean in this context?).

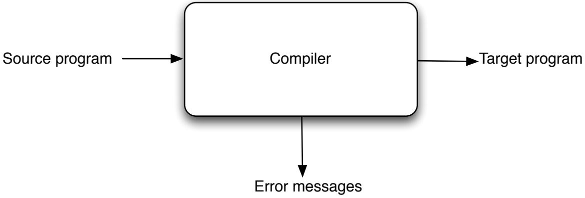


Typically, the input language (called source language) is more high-level than the output language (called target language)
Examples

- Source: Java, target: JVM bytecode.
- Source: JVM bytecode, target: ARM/x86 machine code

Overview: what is a compiler?

A compiler is a program that translates programs from one programming language to programs in another programming language. The translation should preserve meaning (what does "preserve" and "meaning" mean in this context?).



Typically, the input language (called source language) is more high-level than the output language (called target language)

Examples

- Source: Java, target: JVM bytecode.
- Source: JVM bytecode, target: ARM/x86 machine code
- Source: TensorFlow, target: GPU/TPU machine code.

Example translation: source program

Example translation: source program

Here is a little program. (What does it do?)

```
int testfun( int n ){
  int res = 1;
  while( n > 0 ){
      n--;
      res *= 2; }
      return res; }
```

Example translation: source program

Here is a little program. (What does it do?)

```
int testfun( int n ){
  int res = 1;
  while( n > 0 ){
      n--;
      res *= 2; }
      return res; }
```

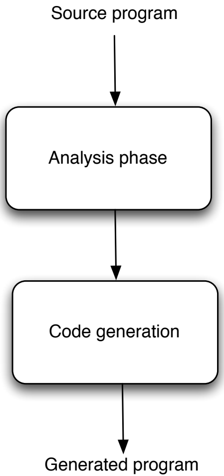
Using `clang -S` this translates to the following x86 machine code ...

Example translation: target program

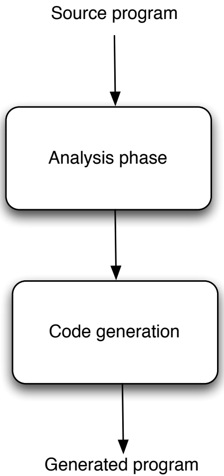
```
_testfun:                               ## @testfun
        .cfi_startproc
                                        ## BB#0:
        pushq   %rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp2:
        .cfi_def_cfa_register %rbp
        movl    %edi, -4(%rbp)
        movl    $1, -8(%rbp)
LBB0_1:                                 ## =>This Inner Loop Header: Depth=1
        cmpl    $0, -4(%rbp)
        jle     LBB0_3

                                        ## BB#2:
                                        ##   in Loop: Header=BB0_1 Depth=1
        movl    -4(%rbp), %eax
        addl    $4294967295, %eax       ## imm = 0xFFFFFFFF
        movl    %eax, -4(%rbp)
        movl    -8(%rbp), %eax
        shll    $1, %eax
        movl    %eax, -8(%rbp)
        jmp     LBB0_1
LBB0_3:
        movl    -8(%rbp), %eax
        popq    %rbp
        retq
        .cfi_endproc
```

Compilers have a beautifully simple structure

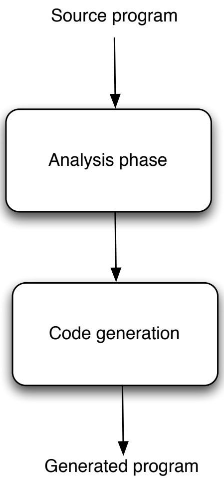


Compilers have a beautifully simple structure



In the analysis phase two things happen:

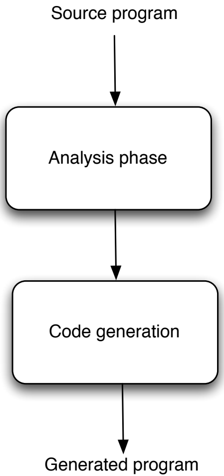
Compilers have a beautifully simple structure



In the analysis phase two things happen:

- Analysing if the program is well-formed (e.g. checking for syntax and type errors).

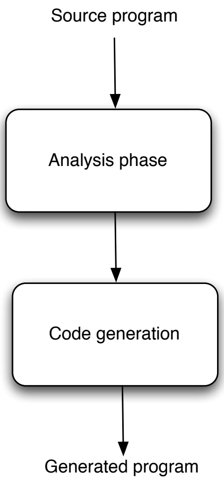
Compilers have a beautifully simple structure



In the analysis phase two things happen:

- Analysing if the program is well-formed (e.g. checking for syntax and type errors).
- Creating a convenient (for a computer) representation of the source program structure for further processing. (Abstract syntax tree (AST), symbol table).

Compilers have a beautifully simple structure

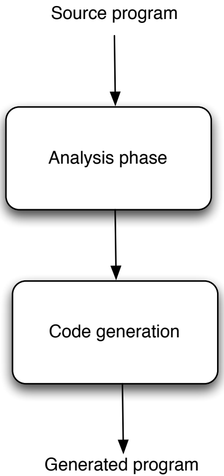


In the analysis phase two things happen:

- Analysing if the program is well-formed (e.g. checking for syntax and type errors).
- Creating a convenient (for a computer) representation of the source program structure for further processing. (Abstract syntax tree (AST), symbol table).

The executable program is then generated from the AST in the code generation phase.

Compilers have a beautifully simple structure



In the analysis phase two things happen:

- Analysing if the program is well-formed (e.g. checking for syntax and type errors).
- Creating a convenient (for a computer) representation of the source program structure for further processing. (Abstract syntax tree (AST), symbol table).

The executable program is then generated from the AST in the code generation phase.

Let's refine this.

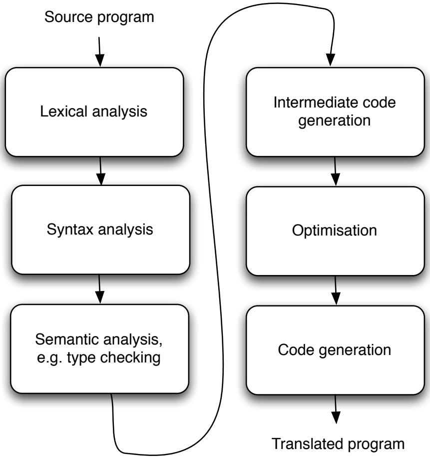
Compiler structure

Compiler structure

Compilers have a beautifully simple structure. This structure was arrived at by breaking a hard problem (compilation) into several smaller problems and solving them separately. This has the added advantage of allowing to retarget compilers (changing source or target language) quite easily.

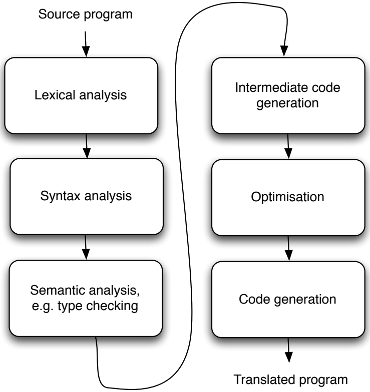
Compiler structure

Compilers have a beautifully simple structure. This structure was arrived at by breaking a hard problem (compilation) into several smaller problems and solving them separately. This has the added advantage of allowing to retarget compilers (changing source or target language) quite easily.



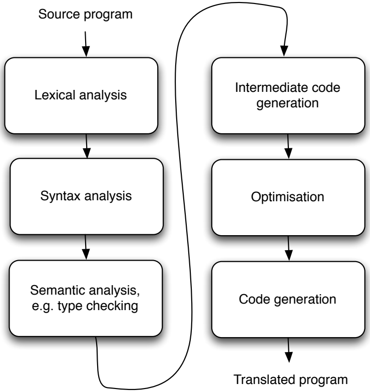
Compiler structure

Compiler structure



Interesting question: when do these phases happen?

Compiler structure

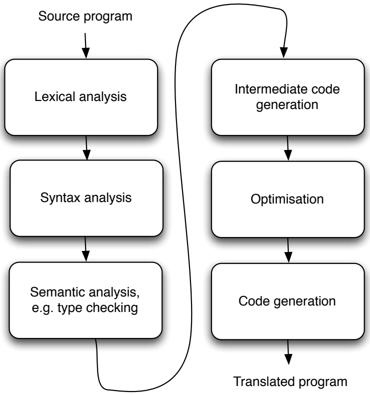


Interesting question: when do these phases happen?

In the past, all happend at ... compile-time. Now some happen at run-time in Just-in-time compilers (JITs). This has profound influences on choice of algorithms and performance.

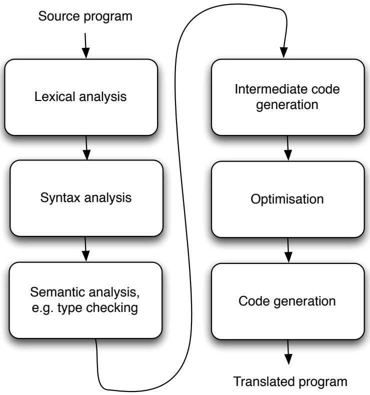
Compiler structure

Compiler structure



Another interesting question: do you note some thing about all these phases?

Compiler structure



Another interesting question: do you note some thing about all these phases?

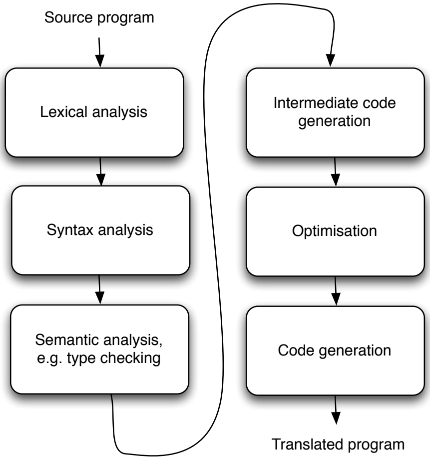
The phases are purely functional, in that they take one input, and return one output. Modern programming languages like Haskell, Ocaml, F#, Rust or Scala are ideal for writing compilers.

Phases: Overview

- Lexical analysis
- Syntactic analysis (parsing)
- Semantic analysis (type-checking)
- Intermediate code generation
- Optimisation
- Code generation

Phases: Lexical analysis

Phases: Lexical analysis



Phases: Lexical analysis

Phases: Lexical analysis

What is the input to a compiler?

Phases: Lexical analysis

What is the input to a compiler?

A (often long) string, i.e. a sequence of characters.

Phases: Lexical analysis

What is the input to a compiler?

A (often long) string, i.e. a sequence of characters.

Strings are not an efficient data-structure for a compiler to work with (= generate code from). Instead, compilers generate code from a more convenient data structure called “abstract syntax trees” (ASTs). We construct the AST of a program in two phases:

- Lexical anlysis. Where the input string is converted into a list of tokens.
- Parsing. Where the AST is constructed from a token list.

Phases: Lexical analysis

Phases: Lexical analysis

In the lexical analysis, a string is converted into a list of tokens.
Example: The program

Phases: Lexical analysis

In the lexical analysis, a string is converted into a list of tokens.
Example: The program

```
int testfun( int n ){
  int res = 1;
  while( n > 0 ){
      n--;
      res *= 2; }
      return res; }
```

Phases: Lexical analysis

In the lexical analysis, a string is converted into a list of tokens.
Example: The program

```
int testfun( int n ){
  int res = 1;
  while( n > 0 ){
      n--;
      res *= 2; }
      return res; }
```

Is (could be) represented as the list

```
T_int, T_ident ( "testfun" ), T_left_brack,
T_int, T_ident ( "n" ), T_rightbrack,
T_left_curly_brack, T_int, T_ident ( "res" ),
T_eq, T_num ( 1 ), T_semicolon, T_while, ...
```

Phases: Lexical analysis

```
T_int, T_ident ( "testfun" ), T_left_brack,
T_int, T_ident ( "n" ), T_rightbrack,
T_left_curly_brack, T_int, T_ident ( "res" ),
T_eq, T_num ( 1 ), T_semicolon, T_while, ...
```

Why is this interesting?

Phases: Lexical analysis

```
T_int, T_ident ( "testfun" ), T_left_brack,
T_int, T_ident ( "n" ), T_rightbrack,
T_left_curly_brack, T_int, T_ident ( "res" ),
T_eq, T_num ( 1 ), T_semicolon, T_while, ...
```

Why is this interesting?

- Abstracts from irrelevant detail (e.g. syntax of keywords, whitespace, comments).

Phases: Lexical analysis

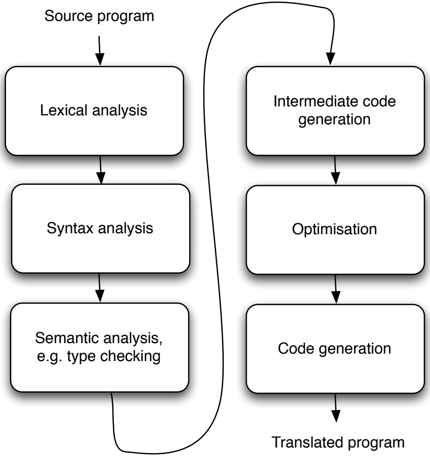
```
T_int, T_ident ( "testfun" ), T_left_brack,
T_int, T_ident ( "n" ), T_rightbrack,
T_left_curly_brack, T_int, T_ident ( "res" ),
T_eq, T_num ( 1 ), T_semicolon, T_while, ...
```

Why is this interesting?

- Abstracts from irrelevant detail (e.g. syntax of keywords, whitespace, comments).
- Makes the next phase (parsing) much easier.

Phases: syntax analysis (parsing)

Phases: syntax analysis (parsing)



Phases: syntax analysis (parsing)

Phases: syntax analysis (parsing)

This phase converts the program (list of tokens) into a tree, the AST of the program (compare to the DOM of a webpage). This is a very convenient data structure because syntax-checking (type-checking) and code-generation can be done by walking the AST (cf visitor pattern). But how is a program a tree?

Phases: syntax analysis (parsing)

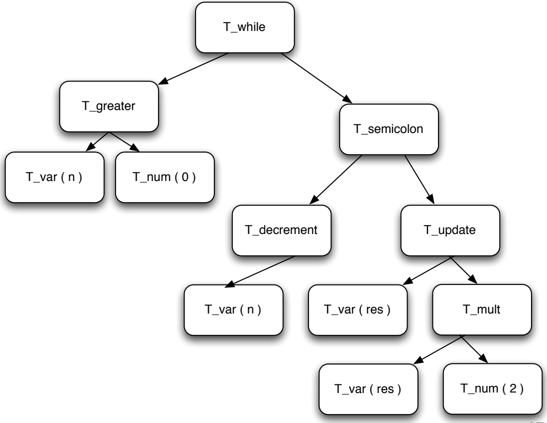
This phase converts the program (list of tokens) into a tree, the AST of the program (compare to the DOM of a webpage). This is a very convenient data structure because syntax-checking (type-checking) and code-generation can be done by walking the AST (cf visitor pattern). But how is a program a tree?

```
while( n > 0 ){
     n--;
     res *= 2; }
```

Phases: syntax analysis (parsing)

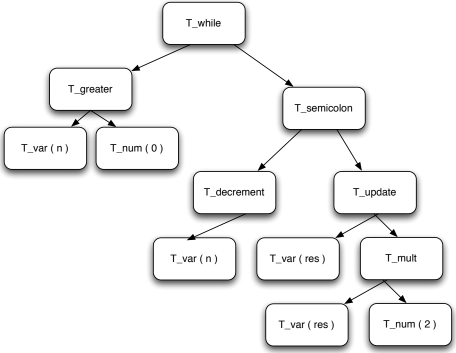
This phase converts the program (list of tokens) into a tree, the AST of the program (compare to the DOM of a webpage). This is a very convenient data structure because syntax-checking (type-checking) and code-generation can be done by walking the AST (cf visitor pattern). But how is a program a tree?

```
while( n > 0 ){
    n--;
    res *= 2; }
```

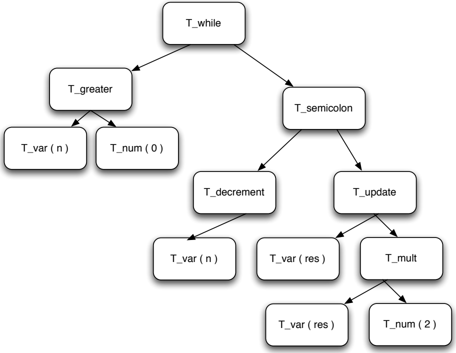


Phases: syntax analysis (parsing)

Phases: syntax analysis (parsing)

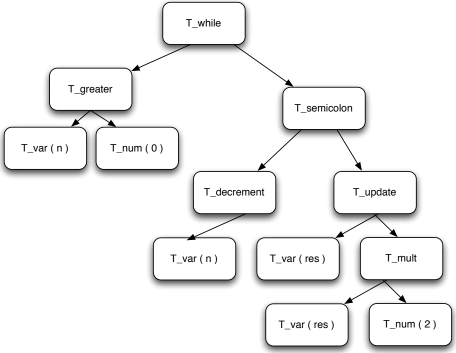


Phases: syntax analysis (parsing)



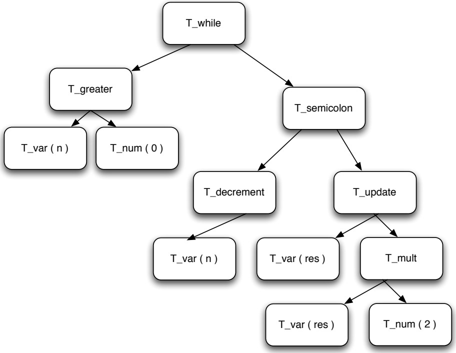
- The AST is often implemented as a tree of linked objects.

Phases: syntax analysis (parsing)



- The AST is often implemented as a tree of linked objects.
- The compiler writer must design the AST data structure carefully so that it is easy to build (during syntax analysis), and easy to walk (during code generation).

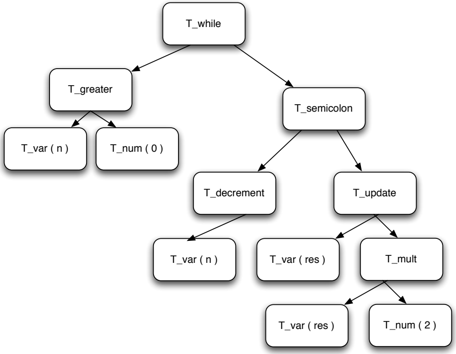
Phases: syntax analysis (parsing)



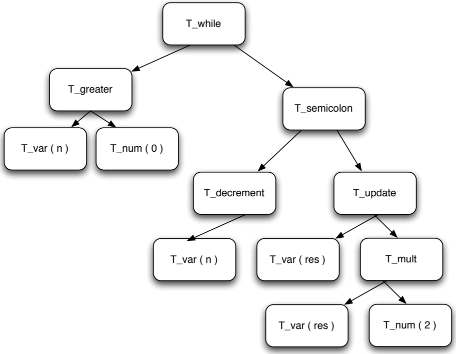
- The AST is often implemented as a tree of linked objects.
- The compiler writer must design the AST data structure carefully so that it is easy to build (during syntax analysis), and easy to walk (during code generation).
- The performance of the compiler strongly depends on the AST, so a lot of optimisation goes here for instustrial strength compilers.

Phases: syntax analysis (parsing)

Phases: syntax analysis (parsing)

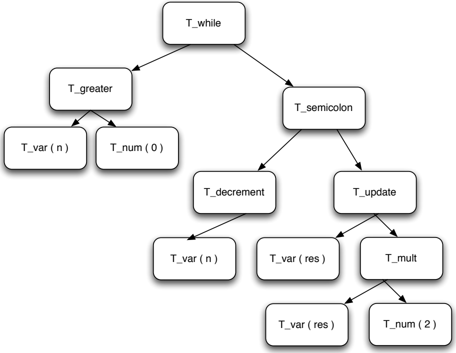


Phases: syntax analysis (parsing)



The construction of the AST has another important role: syntax checking, i.e. checking if the program is syntactically valid!

Phases: syntax analysis (parsing)

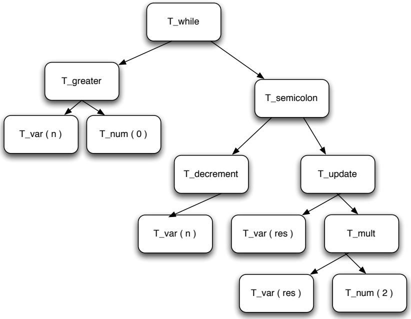


The construction of the AST has another important role: syntax checking, i.e. checking if the program is syntactically valid!

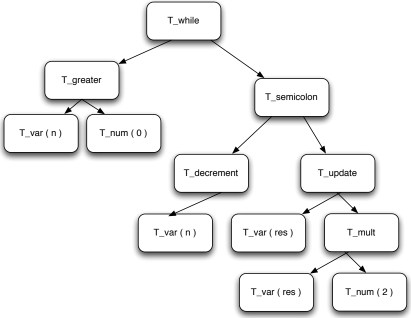
This dual role is because the rules for constructing the AST are essentially exactly the rules that determine the set of syntactically valid programs. Here the theory of formal languages (context free, context sensitive, and finite automata) is of prime importance. We will study this in detail.

Phases: syntax analysis (parsing)

Phases: syntax analysis (parsing)



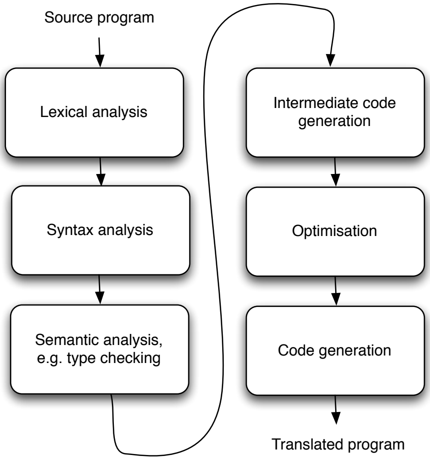
Phases: syntax analysis (parsing)



Great news: the generation of lexical analysers and parsers can be automated by using **parser generators** (e.g. lex, yacc). Decades of research have gone into parser generators, and in practise they generate better lexers and parsers than most programmers would be able to. Alas, parser generators are quite complicated beasts, and in order to understand them, it is helpful to understand formal languages and lexing/parsing. The best way to understand this is to write a toy lexer and parser.

Phases: semantic analysis

Phases: semantic analysis



Phases: semantic analysis

Phases: semantic analysis

While parsing can reject syntactically invalid programs, it cannot reject semantically invalid programs, e.g. programs with more complicated ’semantic’ mistakes are harder to catch. Examples.

Phases: semantic analysis

While parsing can reject syntactically invalid programs, it cannot reject semantically invalid programs, e.g. programs with more complicated 'semantic' mistakes are harder to catch. Examples.

```
void main() {
     i = 7
     int i = 7
      ...
```

Phases: semantic analysis

While parsing can reject syntactically invalid programs, it cannot reject semantically invalid programs, e.g. programs with more complicated 'semantic' mistakes are harder to catch. Examples.

```
void main() {
     i = 7
     int i = 7
      ...
```

```
if ( 3 + true ) > "hello" then ...
```

Phases: semantic analysis

While parsing can reject syntactically invalid programs, it cannot reject semantically invalid programs, e.g. programs with more complicated 'semantic' mistakes are harder to catch. Examples.

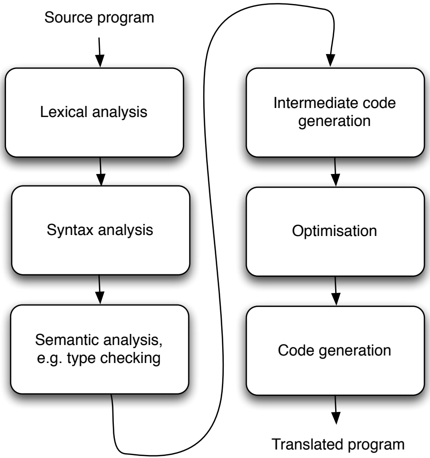
```
void main() {
     i = 7
     int i = 7
      ...
```

```
if ( 3 + true ) > "hello" then ...
```

They are caught with semantic analysis. The key technology are types. Modern languages like Scala, Rust, Haskell, Ocaml, F# employ type inference.

Phases: intermediate code generation

Phases: intermediate code generation



Phases: intermediate code generation

Phases: intermediate code generation

There are many different CPUs with different machine languages. Often the machine language changes subtly from CPU version to CPU version. It would be annoying if we had to rewrite large parts of the compiler. Fortunately, most machine languages are rather similar. This helps us to abstract almost the whole compiler from the details of the target language. The way we do this is by using in essence two compilers.

Phases: intermediate code generation

There are many different CPUs with different machine languages. Often the machine language changes subtly from CPU version to CPU version. It would be annoying if we had to rewrite large parts of the compiler. Fortunately, most machine languages are rather similar. This helps us to abstract almost the whole compiler from the details of the target language. The way we do this is by using in essence two compilers.

- Develop an intermediate language that captures the essence of almost all machine languages.

Phases: intermediate code generation

There are many different CPUs with different machine languages. Often the machine language changes subtly from CPU version to CPU version. It would be annoying if we had to rewrite large parts of the compiler. Fortunately, most machine languages are rather similar. This helps us to abstract almost the whole compiler from the details of the target language. The way we do this is by using in essence two compilers.

- Develop an intermediate language that captures the essence of almost all machine languages.
- Compile to this intermediate language.

Phases: intermediate code generation

There are many different CPUs with different machine languages. Often the machine language changes subtly from CPU version to CPU version. It would be annoying if we had to rewrite large parts of the compiler. Fortunately, most machine languages are rather similar. This helps us to abstract almost the whole compiler from the details of the target language. The way we do this is by using in essence two compilers.

- Develop an intermediate language that captures the essence of almost all machine languages.
- Compile to this intermediate language.
- Do compiler optimisations in the intermediate language.

Phases: intermediate code generation

There are many different CPUs with different machine languages. Often the machine language changes subtly from CPU version to CPU version. It would be annoying if we had to rewrite large parts of the compiler. Fortunately, most machine languages are rather similar. This helps us to abstract almost the whole compiler from the details of the target language. The way we do this is by using in essence two compilers.

- Develop an intermediate language that captures the essence of almost all machine languages.
- Compile to this intermediate language.
- Do compiler optimisations in the intermediate language.
- Translate the intermediate representation to the target machine language. This step can be seen as a mini-compiler.

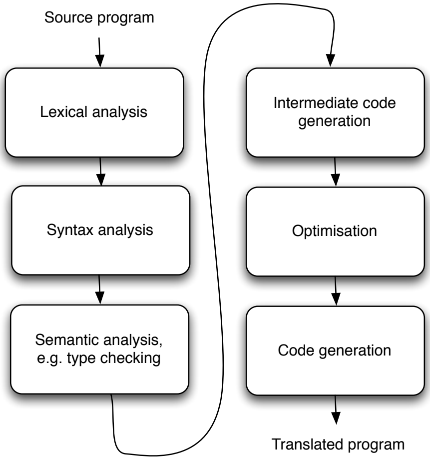
Phases: intermediate code generation

There are many different CPUs with different machine languages. Often the machine language changes subtly from CPU version to CPU version. It would be annoying if we had to rewrite large parts of the compiler. Fortunately, most machine languages are rather similar. This helps us to abstract almost the whole compiler from the details of the target language. The way we do this is by using in essence two compilers.

- Develop an intermediate language that captures the essence of almost all machine languages.
- Compile to this intermediate language.
- Do compiler optimisations in the intermediate language.
- Translate the intermediate representation to the target machine language. This step can be seen as a mini-compiler.
- If we want to retarget the compiler to a new machine language, only this last step needs to be rewritten. Nice data abstraction.

Phases: optimiser

Phases: optimiser



Phases: optimiser

Phases: optimiser

Translating a program often introduces various inefficiencies, make the program e.g. run slow, or use a lot of memories, or use a lot of power (important for mobile phones). Optimisers try to remove these inefficiencies, by replacing the inefficient program with a more efficient version (without changing the meaning of the program).

Phases: optimiser

Translating a program often introduces various inefficiencies, make the program e.g. run slow, or use a lot of memories, or use a lot of power (important for mobile phones). Optimisers try to remove these inefficiencies, by replacing the inefficient program with a more efficient version (without changing the meaning of the program).

Most code optimisations are problems are difficult (NP complete or undecidable), so optimisers are expensive to run, often (but not always) lead to modest improvements only. They are also difficult algorithmically. These difficulties are exacerbate for JITs because the are executed at program run-time.

Phases: optimiser

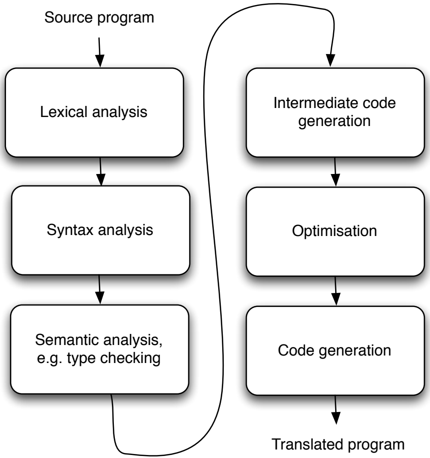
Translating a program often introduces various inefficiencies, make the program e.g. run slow, or use a lot of memories, or use a lot of power (important for mobile phones). Optimisers try to remove these inefficiencies, by replacing the inefficient program with a more efficient version (without changing the meaning of the program).

Most code optimisations are problems are difficult (NP complete or undecidable), so optimisers are expensive to run, often (but not always) lead to modest improvements only. They are also difficult algorithmically. These difficulties are exacerbate for JITs because the are executed at program run-time.

However, some optimisations are easy, e.g. inlining of functions: if a function is short (e.g. computing sum of two numbers), replacing the call to the function with its code, can lead to faster code. (What is the disadvantage of this?)

Phases: code generation

Phases: code generation



Phases: code generation

Phases: code generation

This straighforward phase translates the generated intermediate code to machine code. As machine code and intermediate code are much alike, this ’mini-compiler’ is simple and fast.

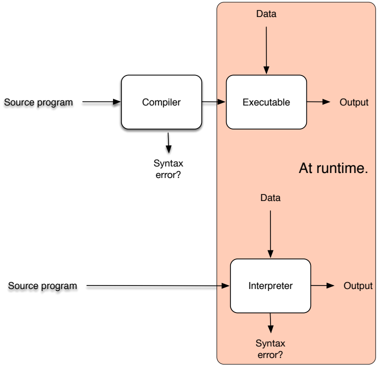
Compilers vs interpreters

Compilers vs interpreters

Interpreters are a second way to run programs.

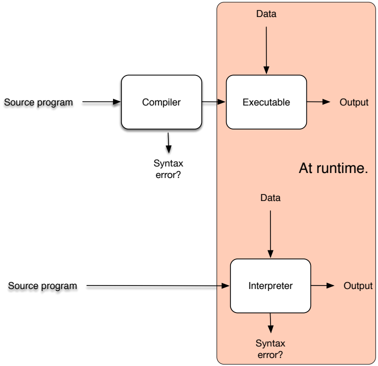
Compilers vs interpreters

Interpreters are a second way to run programs.



Compilers vs interpreters

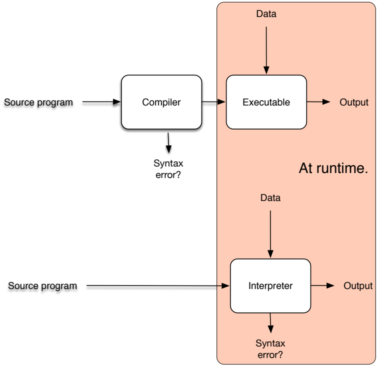
Interpreters are a second way to run programs.



- The advantage of compilers is that generated code is faster, because a lot of work has to be done only once (e.g. lexing, parsing, type-checking, optimisation). And the results of this work are shared in every execution. The interpreter has to redo this work everytime.

Compilers vs interpreters

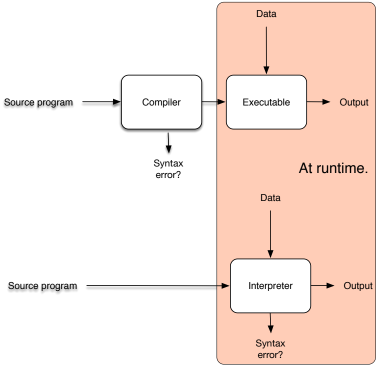
Interpreters are a second way to run programs.



- The advantage of compilers is that generated code is faster, because a lot of work has to be done only once (e.g. lexing, parsing, type-checking, optimisation). And the results of this work are shared in every execution. The interpreter has to redo this work everytime.
- The advantage of interpreters is that they are much simpler than compilers.

Compilers vs interpreters

Interpreters are a second way to run programs.



- The advantage of compilers is that generated code is faster, because a lot of work has to be done only once (e.g. lexing, parsing, type-checking, optimisation). And the results of this work are shared in every execution. The interpreter has to redo this work everytime.
- The advantage of interpreters is that they are much simpler than compilers.

We won't say much more about interpreters in this course.

Literature

Literature

Compilers are among the most studied and most well understood parts of informatics. Many good books exist. Here are some of my favourites, although I won't follow any of them closely.

Literature

Compilers are among the most studied and most well understood parts of informatics. Many good books exist. Here are some of my favourites, although I won't follow any of them closely.

- **Modern Compiler Implementation in Java** (second edition) by Andrew Appel and Jens Palsberg. Probably closest to our course. Moves quite fast.

Literature

Compilers are among the most studied and most well understood parts of informatics. Many good books exist. Here are some of my favourites, although I won't follow any of them closely.

- **Modern Compiler Implementation in Java** (second edition) by Andrew Appel and Jens Palsberg. Probably closest to our course. Moves quite fast.
- **Compilers - Principles, Techniques and Tools** (second edition) by Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. The first edition of this book is is the classic text on compilers, known as the "Dragon Book", but its first edition is a bit obsolete. The second edition is substantially expanded and goes well beyond the scope of our course. For my liking, the book is a tad long.

Literature

Literature

Some other material:

Literature

Some other material:

- **Engineering a Compiler**, by Keith Cooper, Linda Torczon.

Literature

Some other material:

- **Engineering a Compiler**, by Keith Cooper, Linda Torczon.
- The **Alex Aiken's Stanford University online course on compilers.** This course coveres similar ground as ours, but goes more in-depth. I was quite influenced by Aiken's course when I designed our's.

Literature

Some other material:

- **Engineering a Compiler**, by Keith Cooper, Linda Torczon.
- The **Alex Aiken's Stanford University online course on compilers.** This course coveres similar ground as ours, but goes more in-depth. I was quite influenced by Aiken's course when I designed our's.
- **Computer Architecture - A Quantitative Approach** (sixth edition) by John Hennessey and David Patterson. This is the 'bible' for computer architecture. It goes way beyond what is required for our course, but very well written by some of the world's leading experts on computer architecture. Well worth studying.

How to enjoy and benefit from this course

How to enjoy and benefit from this course

- Assessed coursework is designed to reinforce and integrate lecture material; it's designed to help you pass the exam

How to enjoy and benefit from this course

- Assessed coursework is designed to reinforce and integrate lecture material; it's designed to help you pass the exam
- Go look at the past papers - now.

How to enjoy and benefit from this course

- Assessed coursework is designed to reinforce and integrate lecture material; it's designed to help you pass the exam
- Go look at the past papers - now.
- Use the tutorials to get feedback on your solutions

How to enjoy and benefit from this course

- Assessed coursework is designed to reinforce and integrate lecture material; it's designed to help you pass the exam
- Go look at the past papers - now.
- Use the tutorials to get feedback on your solutions
- Substantial lab exercise should bring it all together

How to enjoy and benefit from this course

- Assessed coursework is designed to reinforce and integrate lecture material; it’s designed to help you pass the exam
- Go look at the past papers - now.
- Use the tutorials to get feedback on your solutions
- Substantial lab exercise should bring it all together
- Ask questions, in the lectures, in the labs, on Canvas or in person!

How to enjoy and benefit from this course

- Assessed coursework is designed to reinforce and integrate lecture material; it's designed to help you pass the exam
- Go look at the past papers - now.
- Use the tutorials to get feedback on your solutions
- Substantial lab exercise should bring it all together
- Ask questions, in the lectures, in the labs, on Canvas or in person!
- Design your own mini-languages and write compilers for them.

How to enjoy and benefit from this course

- Assessed coursework is designed to reinforce and integrate lecture material; it's designed to help you pass the exam
- Go look at the past papers - now.
- Use the tutorials to get feedback on your solutions
- Substantial lab exercise should bring it all together
- Ask questions, in the lectures, in the labs, on Canvas or in person!
- Design your own mini-languages and write compilers for them.
- Have a look at real compilers. There are many free, open-source compilers, g.g. GCC, LLVM, TCC, MiniML, Ocaml, the Scala compiler, GHC, the Haskell compiler.

Feedback

In this module, you will receive feedback through:

- The mark and comments on your assessment
- Feedback to the whole class on assessment and exams
- Feedback to the whole class on lecture understanding
- Model solutions
- Worked examples in class and lecture
- Verbal comments and discussions with tutors in class
- Discussions with your peers on problems
- Online discussion forums
- One to one sessions with the tutors

The more questions you ask, the more you participate in discussions, the more you engage with the course, the more feedback you get.

Questions?