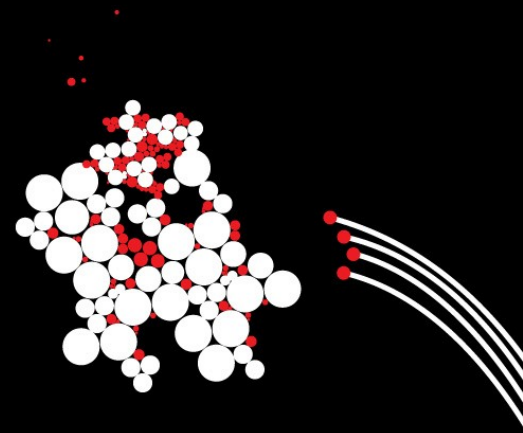


UNIVERSITY OF TWENTE.

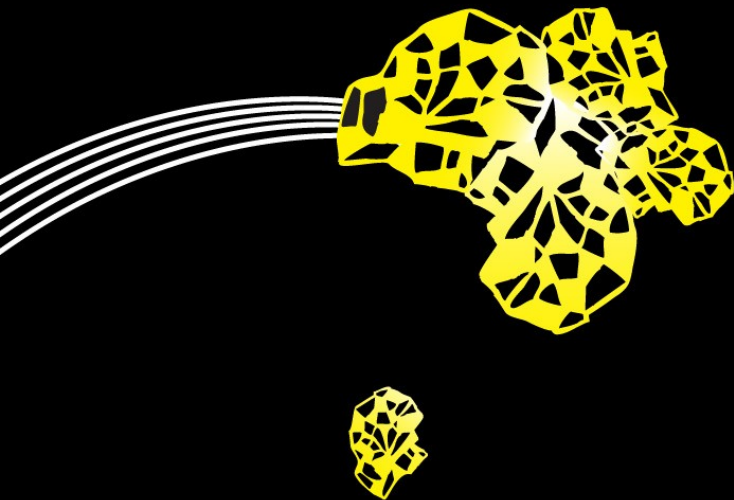


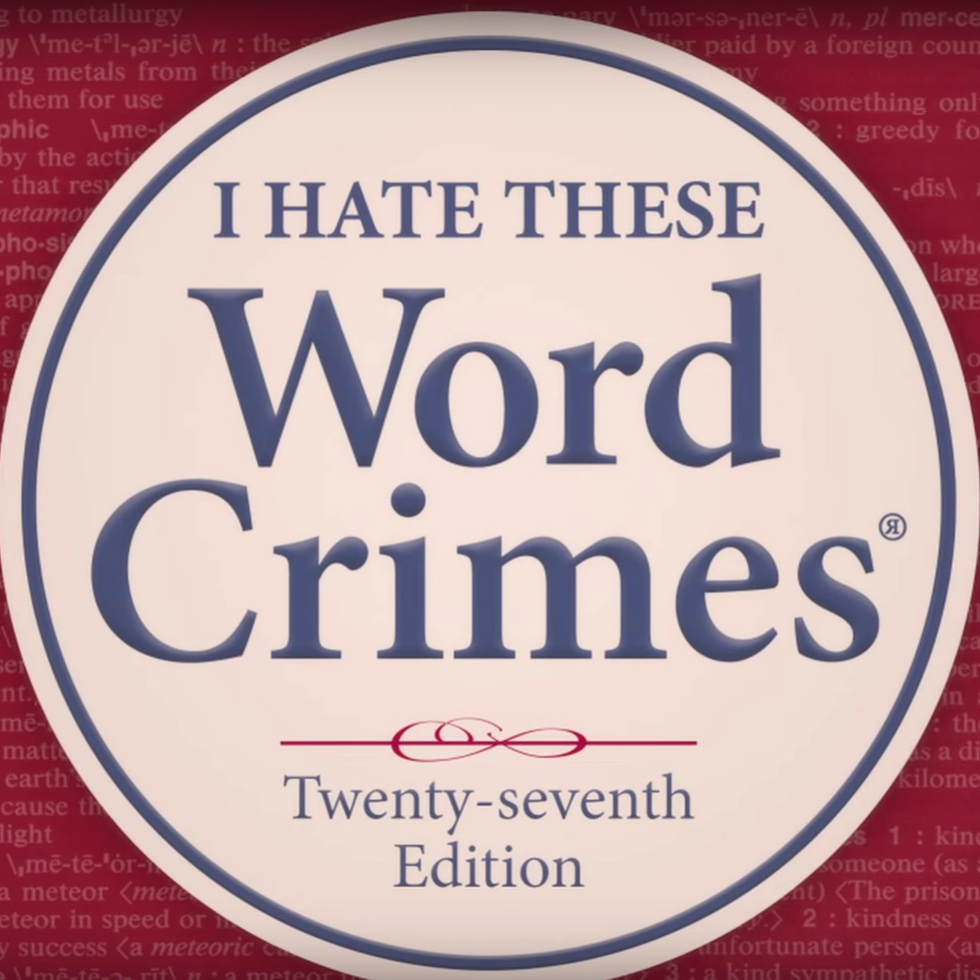
COMPILER CONSTRUCTION:

THE STRUCTURE OF A COMPILER

MODULE 8: PROGRAMMING PARADIGMS

23 APRIL 2018





<https://youtu.be/8Gv0H-vPoDc>

SYNTAX

- In natural language, there is potential ambiguity

- Examples (A)

1. Fruit flies like a banana
2. Time flies like an arrow

- Examples (B)

(Source: The Definitive Antlr4 Reference)

3. To my Ph.D. supervisor, for whom no thanks is too much

(Source: The Definitive Antlr4 Reference)

4. You can't put too much water into a nuclear reactor

3. To my Ph.D. supervisor, for whom no thanks is too much

- Difference between (A) and (B)?
 4. You can't put too much water into a nuclear reactor
 - (A): confusion about the word categories of "flies", "like" → *syntax*
- Difference between (A) and (B)?
 - (B): confusion about the *meaning* of the words → *semantics*
 - (A): confusion about the word categories of "flies", "like" → *syntax*
 - In 3, does "no" mean "zero" or "there isn't any [thanks] that"?
 - (B): confusion about the *meaning* of the words → *semantics*
 - In 4, does "can't" mean "it's dangerous" or "it's impossible"?
 - In 3, does "no" mean "zero" or "there isn't any [thanks] that"?
- In programming languages, both are to be avoided
 - In 4, does "can't" mean "it's dangerous" or "it's impossible"?
 - Precise syntax definition; but how?
- In programming languages, both are to be avoided
 - Precise syntax definition; but how?

COMPILATION STEP 1: SCANNING (A.K.A. LEXING)

- Establish basic building blocks (*words*) and categories for those
 - 1. For natural languages? 2. For programming languages?
 1. Verbs, nouns, adjectives, particles, punctuation marks
 2. Keywords, symbol combinations
- Go through the input text, and
 - Recognize the basic building blocks
 - Tag each building block with its category

Good computer science students attend the lecture

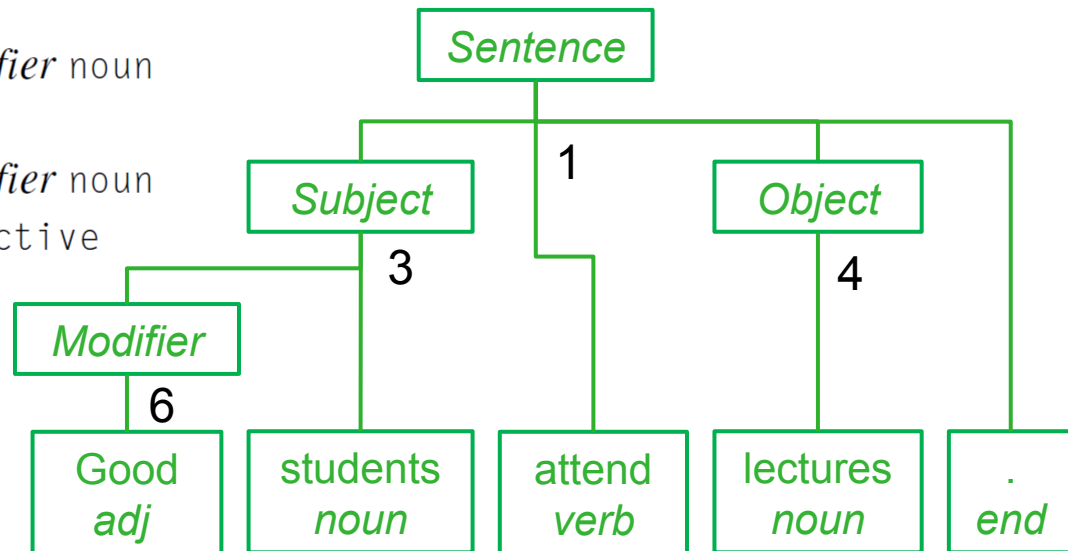
Good <i>adj</i>	computer <i>noun</i>	science <i>noun</i>	students <i>noun</i>	attend <i>verb</i>	the <i>part</i>	lecture <i>noun</i>
--------------------	-------------------------	------------------------	-------------------------	-----------------------	--------------------	------------------------

- Terminology in compiler construction
 - Basic building blocks: *tokens*
 - Categories: *token types*
- Note: spaces are not tokens!

COMPILATION STEP 2: PARSING

- Define (syntactic) structure of language
 - 1. For natural languages? 2. For programming languages?
 1. Subject, verb phrase, object
 2. Class, method, assignment, while-loop
- Go through the token list, and group words together
 - This is driven by *grammar rules*

1	<i>Sentence</i>	→	<i>Subject</i> verb <i>Object</i> endmark
2	<i>Subject</i>	→	noun
3	<i>Subject</i>	→	<i>Modifier</i> noun
4	<i>Object</i>	→	noun
5	<i>Object</i>	→	<i>Modifier</i> noun
6	<i>Modifier</i>	→	adjective
	...		



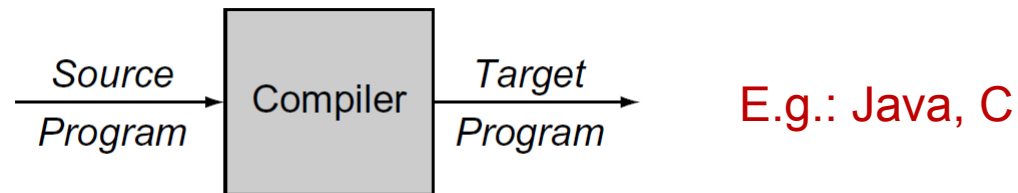
- *Used from right to left in actual parsing*

COMPILATION STEP 3: TYPE CHECKING

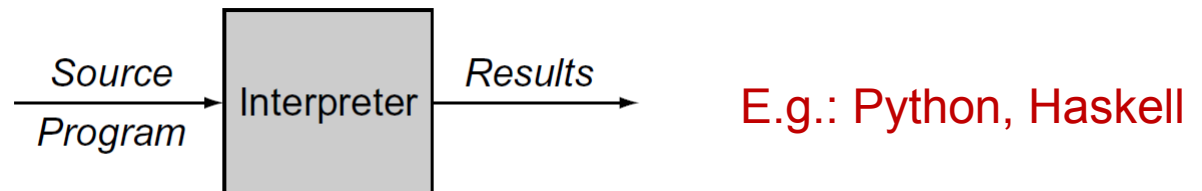
- Check that building blocks in a structure actually belong together
 - 1. For natural languages? 2. For programming languages?
 1. E.g.: singular/plural (*A students attend this lectures*)
 2. E.g.: Data type or method signature correspondence (*int x = null;*)
- Also driven by rules
 - Scanning: *regular grammar/DFA*
 - Parsing: *context-free grammar*
 - Type checking: *context-sensitive rules*

STRUCTURE OF A COMPILER

- A *compiler* is a program that
 - takes an arbitrary program written in its source language, and
 - produces an equivalent program in its target language



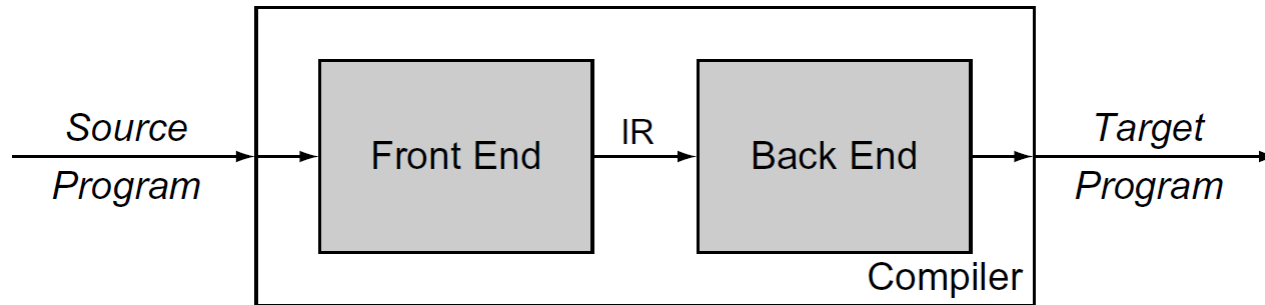
- An *interpreter* is a program that
 - takes an arbitrary program in its source language, and
 - executes it



- What about Java (compiled to bytecode) versus C (to machine code)?
 - Both are compilers, with different source and target languages
 - The JVM acts as an interpreter for bytecode (as source language)

TWO-PHASE COMPILERS

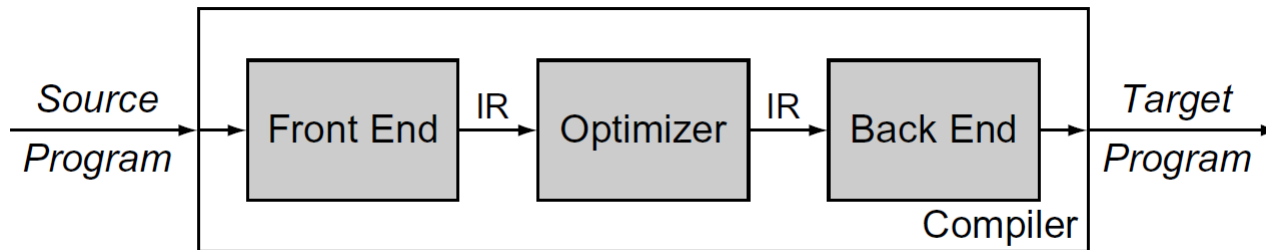
- Compilation process can be usefully broken up
 - Front end: translates to internal representation
 - Back end: produces target language from internal representation
 - Intermediate representation: simple & comprehensive, not readable



- Advantage: intermediate representation can be reused
 - Multiple source languages to the same IR
 - IR to multiple target languages (*retargeting*)
 - Well-known example: `gcc` (= *gnu compiler collection*)
 - `gnu` = `gnu's not unix` (front-recursive acronym, nerd humour)
- UNIVERSITY OF TWENTE | [Source book LLVMC \(Appendix A\)](#) – also used as example target language | CC141: Structure of a Compiler

THREE-PHASE COMPILERS

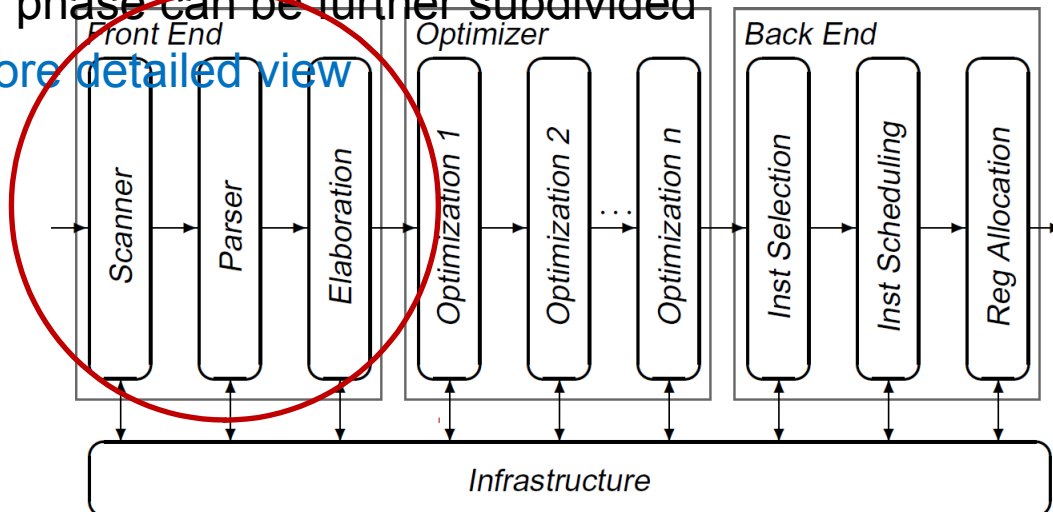
- We can play around with the intermediate representation
 - Optimize: rewrite so program gets faster/smaller/less power-hungry



- Each phase can be further subdivided

- More detailed view

scanning/
parsing/
type checking



EXAMPLE OPTIMIZATION

- What can we improve in

```
b ← ...
c ← ...
a ← 1
for i = 1 to n
  read d
  a ← a × 2 × b × c × d
end
```

unnecessarily
recomputed

```
b ← ...
c ← ...
a ← 1
t ← 2 × b × c
for i = 1 to n
  read d
  a ← a × d × t
end
```

move out
of loop

BACK END STEP 1: INSTRUCTION SELECTION

□ To be implemented: $a \leftarrow a \times 2 \times b \times c \times d$

▪ Available instructions:

- r_1 : register i
- c_1 : literal value

ILOC Operation		Meaning
loadAI	$r_1, c_2 \Rightarrow r_3$	$\text{Memory}(r_1+c_2) \rightarrow r_3$
loadI	$c_1 \Rightarrow r_2$	$c_1 \rightarrow r_2$
mult	$r_1, r_2 \Rightarrow r_3$	$r_1 \times r_2 \rightarrow r_3$
storeAI	$r_1 \Rightarrow r_2, c_3$	$r_1 \rightarrow \text{Memory}(r_2+c_3)$

▪ Initial solution:

reg with base mem address of vars offset of variable a

```

loadAI rarp, @a => ra // load 'a'
loadI 2 => r2 // constant 2 into r2
loadAI rarp, @b => rb // load 'b'
loadAI rarp, @c => rc // load 'c'
loadAI rarp, @d => rd // load 'd'
mult ra, r2 => ra // ra ← a × 2
mult ra, rb => ra // ra ← (a × 2) × b
mult ra, rc => ra // ra ← (a × 2 × b) × c
mult ra, rd => ra // ra ← (a × 2 × b × c) × d
storeAI ra => rarp, @a // write ra back to 'a'
    
```

BACK END STEP 3: REGISTER ALLOCATION

- We've been very generous with registers; can we do better?
 - Instead of 5 registers (including r_{arp}), we can use 3
 - Add to self rather than multiply by 2; combine registers for b, c, d

```
loadAI  rarp, @a => r1           // load 'a'
add     r1, r1  => r1           // r1 ← a × 2
loadAI  rarp, @b => r2           // load 'b'
mult    r1, r2  => r1           // r1 ← (a × 2) × b
loadAI  rarp, @c => r2           // load 'c'
mult    r1, r2  => r1           // r1 ← (a × 2 × b) × c
loadAI  rarp, @d => r2           // load 'd'
mult    r1, r2  => r1           // r1 ← (a × 2 × b × c) × d
storeAI r1      => rarp, @a     // write ra back to 'a'
```

BACK END STEP 2: INSTRUCTION SCHEDULING

- Instruction executing is pipelined
 - Some instructions can be started while others are still running
 - In particular, during external memory fetches (lasting 3 cycles)

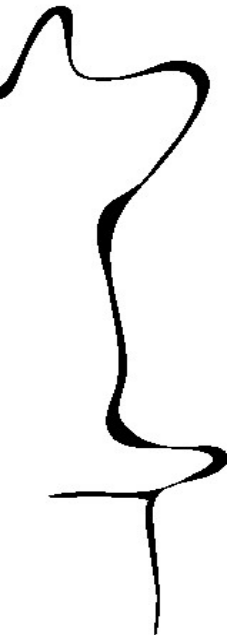
1	3	loadAI	$r_{arp}, @a \Rightarrow r_1$	// load 'a'
4	4	add	$r_1, r_1 \Rightarrow r_1$	// $r_1 \leftarrow a \times 2$
5	7	loadAI	$r_{arp}, @b \Rightarrow r_2$	// load 'b'
8	9	mult	$r_1, r_2 \Rightarrow r_1$	// $r_1 \leftarrow (a \times 2) \times b$
10	12	loadAI	$r_{arp}, @c \Rightarrow r_2$	// load 'c'

13	14	1	3	loadAI	$r_{arp}, @a \Rightarrow r_1$	// load 'a'
15	17	2	4	loadAI	$r_{arp}, @b \Rightarrow r_2$	// load 'b'
18	19	3	5	loadAI	$r_{arp}, @c \Rightarrow r_3$	// load 'c'
20	22	4	4	add	$r_1, r_1 \Rightarrow r_1$	// $r_1 \leftarrow a \times 2$
		5	6	mult	$r_1, r_2 \Rightarrow r_1$	// $r_1 \leftarrow (a \times 2) \times b$
		6	8	loadAI	$r_{arp}, @d \Rightarrow r_2$	// load 'd'
		7	8	mult	$r_1, r_3 \Rightarrow r_1$	// $r_1 \leftarrow (a \times 2 \times b) \times c$
		9	10	mult	$r_1, r_2 \Rightarrow r_1$	// $r_1 \leftarrow (a \times 2 \times b \times c) \times d$
		11	13	storeAI	$r_1 \Rightarrow r_{arp}, @a$	// write r_a back to 'a'

Improved at the cost of one more register



SEEN TODAY

- 
- Structure of compiler
 - Compiler versus interpreter
 - Two- and three-phase compilers
 - Intermediate representation
 - Front end
 - Scanning
 - Parsing
 - Type checking
 - Optimizer
 - Back end
 - Instruction selection
 - Instruction scheduling
 - Register allocation
- 