UNIVERSITY OF TWENTE.

COMPILER CONSTRUCTION:

THE STRUCTURE OF A COMPILER

MODULE 8: PROGRAMMING PARADIGMS

23 APRIL 2019

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
  4. You can’t put too much water into a nuclear reactor
- Difference between (A) and (B)?
  - (A): confusion about the word categories of “flies”, “like” → *syntax*
  - (B): confusion about the *meaning* of the words → *semantics*
    - In 3, does “no” mean “zero” or “there isn’t any [thanks] that”?
    - In 4, does “can’t” mean “it’s dangerous” or “it’s impossible”?
- 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 *adj*	computer *noun*	science *noun*	students *noun*	attend *verb*	the *part*	lecture *noun*

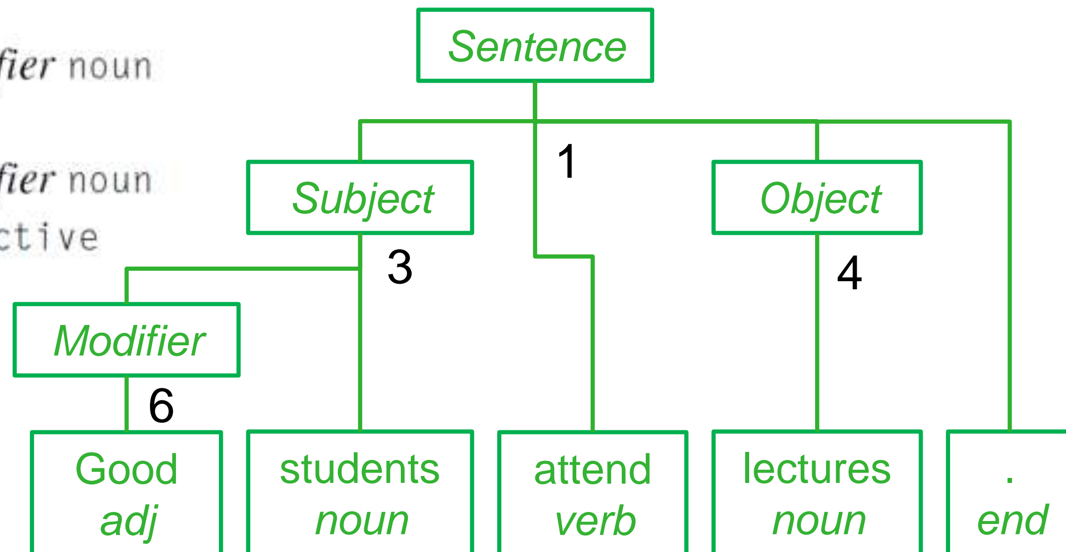
- Terminology in compiler construction
  - Basic building blocks: *tokens*
  - Categories: *token types*
- Note: spaces are not tokens!

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	*Sentence*	→	*Subject* verb *Object* endmark
2	*Subject*	→	noun
3	*Subject*	→	*Modifier* noun
4	*Object*	→	noun
5	*Object*	→	*Modifier* noun
6	*Modifier*	→	adjective
	...		

  - *Used from right to left in actual parsing*

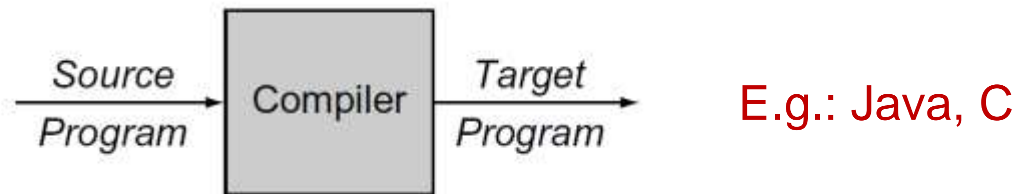


COMPILATION STEP 3: TYPE CHECKING

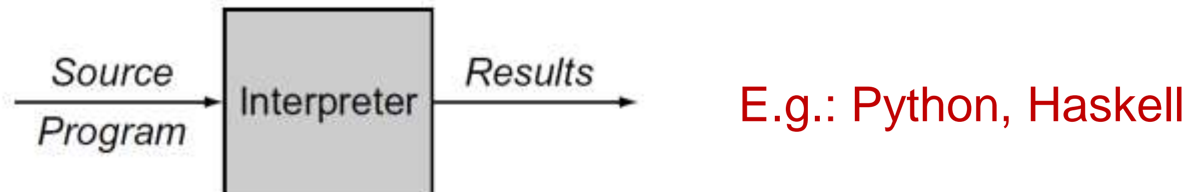
- Check that building blocks in a structure actually belong together
  - 1. For natural languages? 2. For programming languages?
    1. E.g.: singular/plural (A students attend this lectures)
    2. E.g.: Data type or method signature correspondence (**int** x = null;)
- Also driven by rules
  - Scanning: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



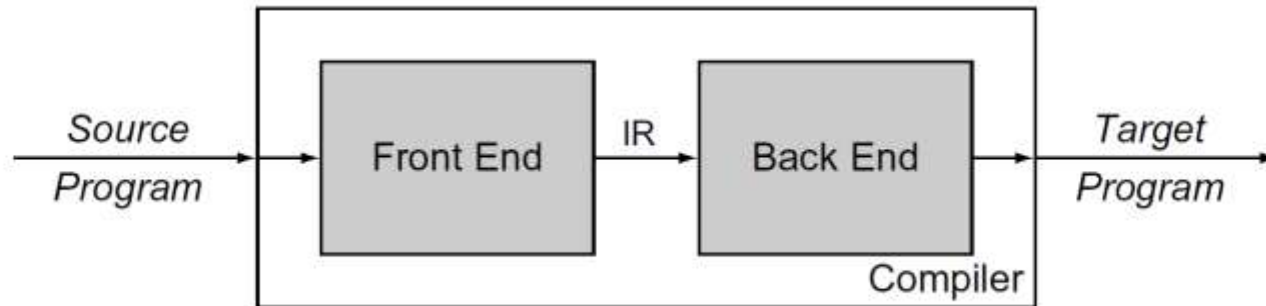
- An *interpreter* is a program that
  - takes an arbitrary program in its source language, and
  - executes it



- 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

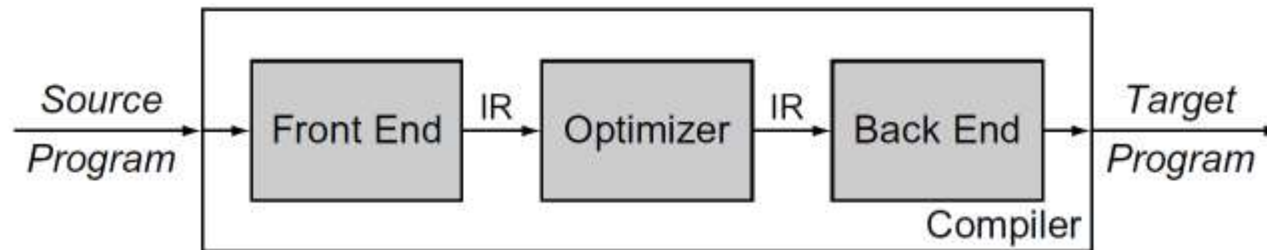
- Compilation process can be usefully broken up
  - Front end: translates to internal representation
  - Back end: produces target language from internal representation
  - Intermediate representation: simple & comprehensive, not readable



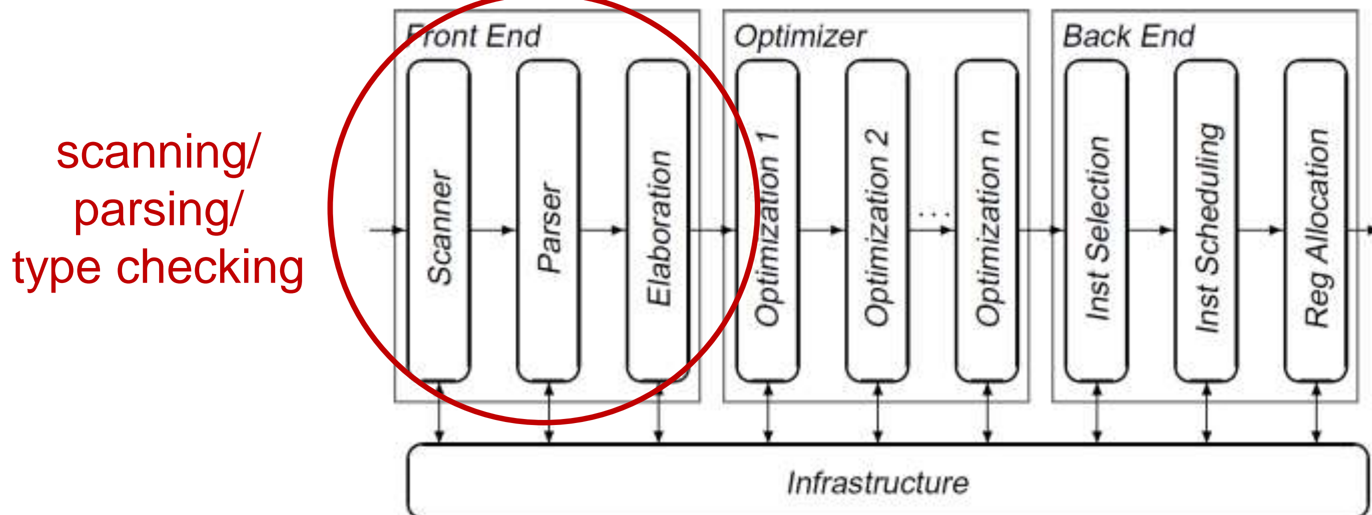
- Advantage: intermediate representation can be reused
  - Multiple source languages to the same IR
  - IR to multiple target languages (*retargeting*)
  - Well-known example: gcc (= *gnu compiler collection*)
    - supports many programming languages + many systems/processors
  - In book: ILOC (Appendix A) – also used as example target language

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



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_i: register i
  - c_i: literal value

ILOC Operation		Meaning
loadAI	$r_1, c_2 \Rightarrow r_3$	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	$r_1 \Rightarrow r_2, c_3$	$r_1 \rightarrow$ Memory(r_2+c_3)

- Initial solution:

reg with base mem addess of vars (points to r_{arp})

offset of variable a (points to @a)

```
loadAI   r_arp, @a  ⇒ r_a         // load 'a'
loadI    2          ⇒ r_2         // constant 2 into r_2
loadAI   r_arp, @b  ⇒ r_b         // load 'b'
loadAI   r_arp, @c  ⇒ r_c         // load 'c'
loadAI   r_arp, @d  ⇒ r_d         // load 'd'
mult     r_a, r_2   ⇒ r_a         // r_a ← a × 2
mult     r_a, r_b   ⇒ r_a         // r_a ← (a × 2) × b
mult     r_a, r_c   ⇒ r_a         // r_a ← (a × 2 × b) × c
mult     r_a, r_d   ⇒ r_a         // r_a ← (a × 2 × b × c) × d
storeAI  r_a        ⇒ r_arp, @a   // write r_a 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 memory fetches (3 cycles) and `mults` (2 cycles)

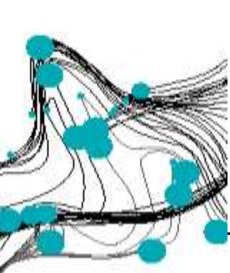
(sequential processor)

Start	End	Op	Operands	Comment
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			
15	17			
18	19			
20	22			

Start	End	Op	Operands	Comment
1	3	loadAI	$r_{arp}, @a \Rightarrow r_1$	// load 'a'
2	4	loadAI	$r_{arp}, @b \Rightarrow r_2$	// load 'b'
3	5	loadAI	$r_{arp}, @c \Rightarrow r_3$	// load 'c'
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

(pipelined processor)



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