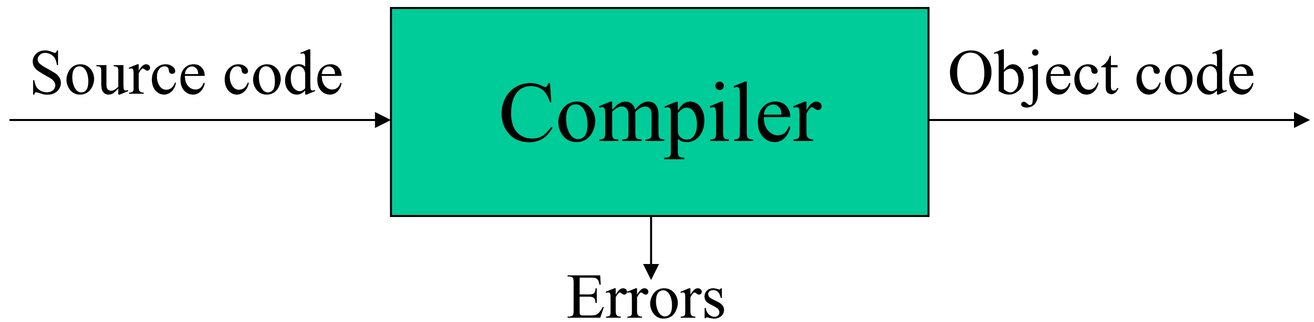
Lecture 2: General Structure of a Compiler



(from last lecture) The compiler:-

- must generate correct code.
- must recognise errors.
- analyses and synthesises.

In today's lecture:

more details about the compiler's structure.

Conceptual Structure:two major phases

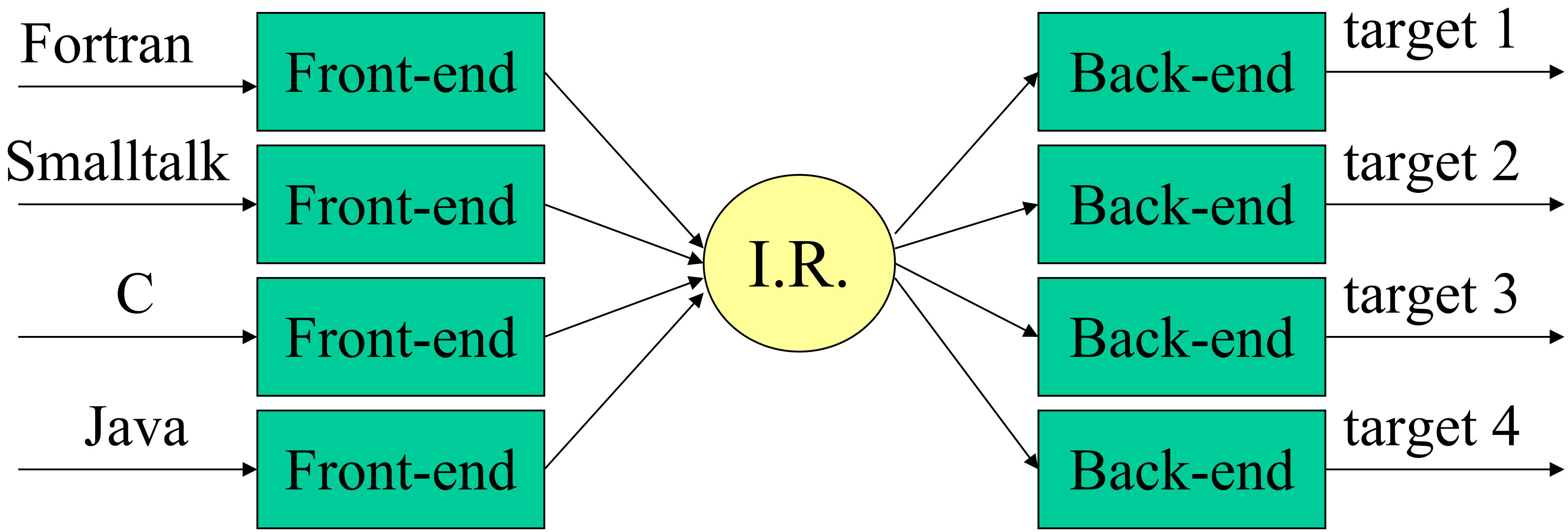
- **Front-end** performs the **analysis** of the source language:
  - Recognises legal and illegal programs and reports errors.
  - “understands” the input program and collects its semantics in an IR.
  - Produces IR and shapes the code for the back-end.
  - Much can be automated.
- **Back-end** does the target language **synthesis**:
  - Chooses instructions to implement each IR operation.
  - Translates IR into target code.
  - Needs to conform with system interfaces.
  - Automation has been less successful.

A problem which we don’t know how to solve in less than exponential time

- Typically front-end is **<u>O(n),</u>** while back-end is **<u>NP-complete</u>**.

What is the implication of this separation (front-end: analysis; back-end:synthesis) in building a compiler for, say, a new language?

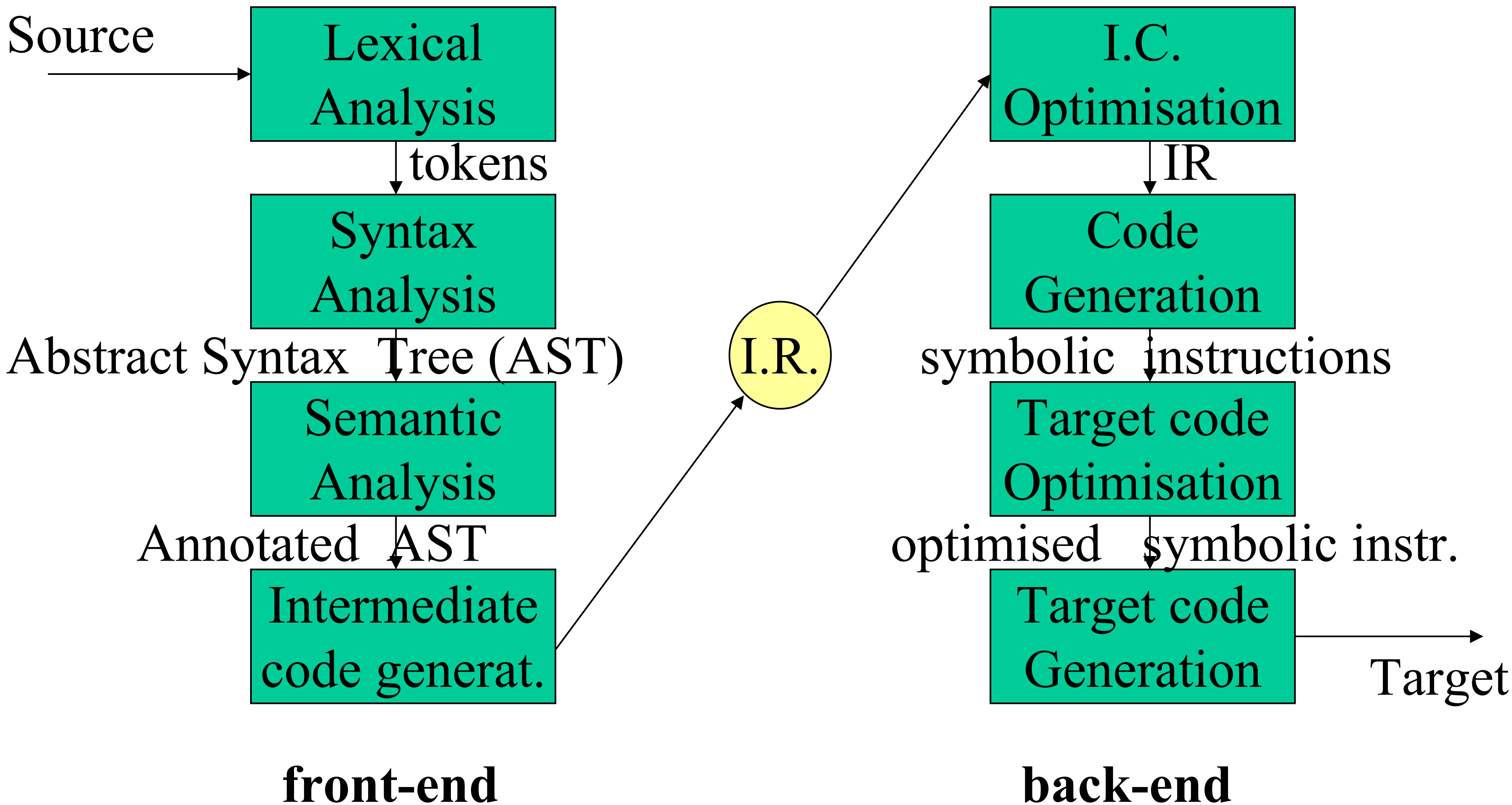
m×n compilers with m+n components!



- All language specific knowledge must be encoded in the front-end
- All target specific knowledge must be encoded in the back-end

But: in practice, this strict separation is not free of charge.

General Structure of a compiler



Lexical Analysis (Scanning)

- Reads characters in the source program and groups them into words (basic unit of syntax)
- Produces words and recognises what sort they are.
- The output is called token and is a pair of the form <*type, lexeme*> or <*token_class, attribute*>
- E.g.: **`a=b+c`** becomes <id,**`a`**> <=,> <id,**`b`**> <+,> <id,**`c`**>
- Needs to record each id attribute: keep a **symbol table**.
- Lexical analysis eliminates white space, etc…
- Speed is important - use a specialised tool: e.g., flex - a tool for generating **scanners**: programs which recognise lexical patterns in text; for more info: **`% man flex`**

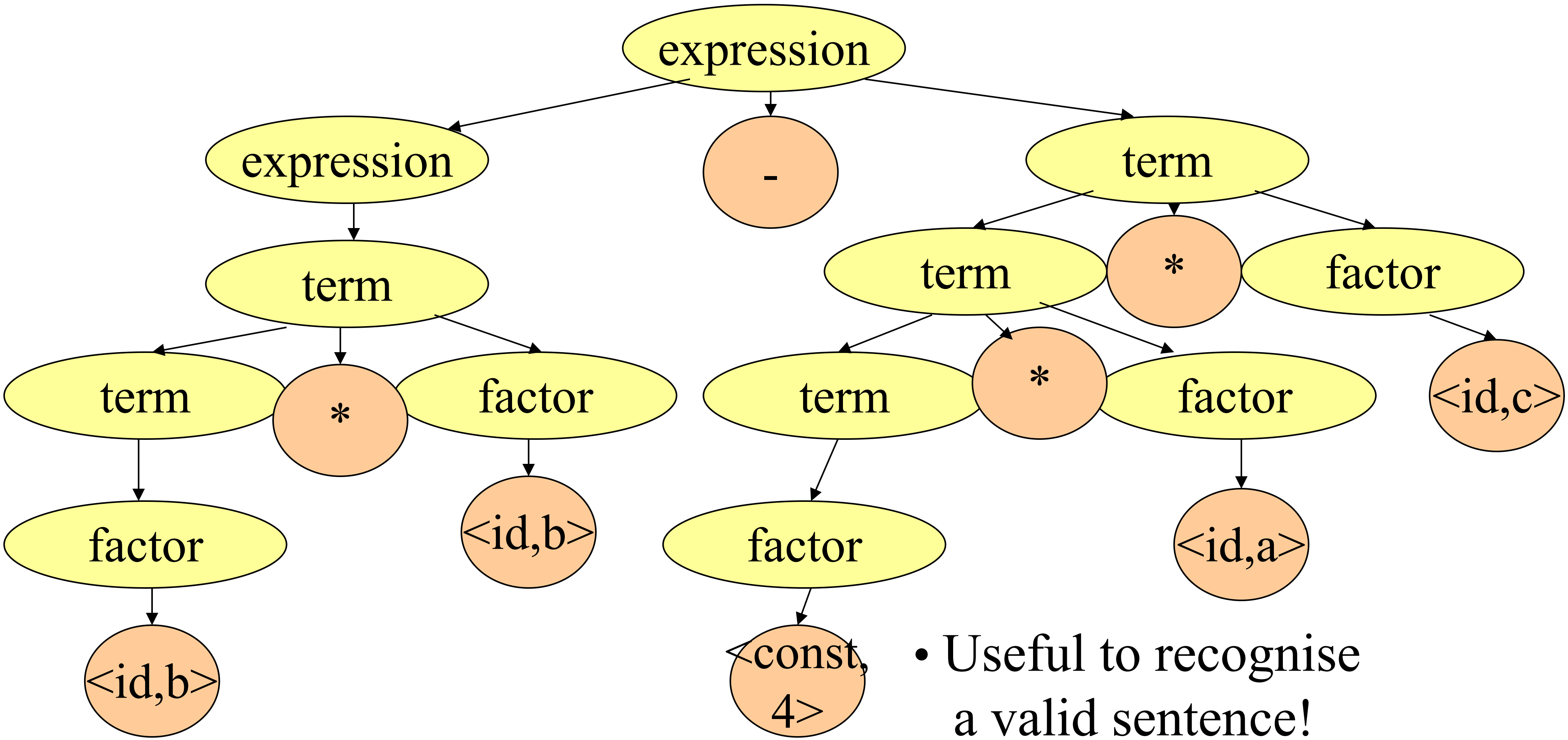
Syntax (or syntactic) Analysis (Parsing)

- Imposes a hierarchical structure on the token stream.
- This hierarchical structure is usually expressed by recursive rules.
- Context-free grammars formalise these recursive rules and guide syntax analysis.
- Example:

```
expression → expression ‘+’ term | expression ‘-’ term | term
term → term ‘*’ factor | term ‘/’ factor | factor
factor → identifier | constant | ‘(‘ expression ‘)’
```

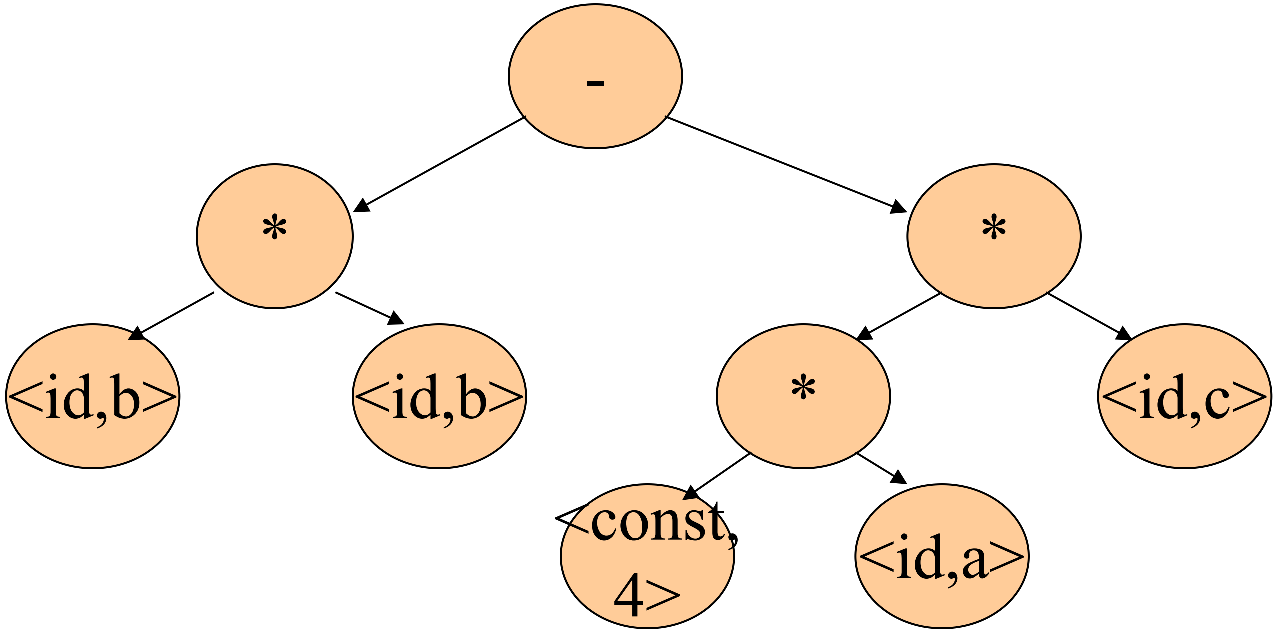
(this grammar defines simple algebraic expressions)

Parsing: parse tree for *b*b-4*a*c*



- Useful to recognise a valid sentence!
- Contains a lot of unneeded information!

AST for *b*b-4*a*c*



- An Abstract Syntax Tree (AST) is a more useful data structure for internal representation. It is a compressed version of the parse tree (summary of grammatical structure without details about its derivation)
- ASTs are one form of IR

Semantic Analysis (context handling)

- Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results.
- Examples:
  - type checking: report error if an operator is applied to an incompatible operand.
  - check flow-of-controls.
  - uniqueness or name-related checks.

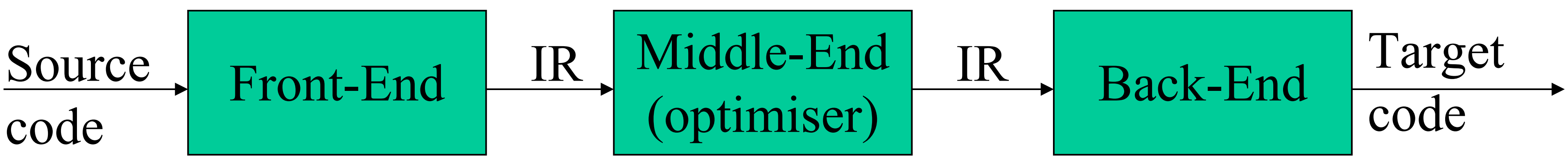
Intermediate code generation

- Translate language-specific constructs in the AST into more general constructs.
- A criterion for the level of "generality": it should be straightforward to generate the target code from the intermediate representation chosen.
- Example of a form of IR (3-address code):

```
tmp1=4
tmp2=tmp1*a
tmp3=tmp2*c
tmp4=b*b
tmp5=tmp4-tmp3
```

Code Optimisation

- The goal is to improve the intermediate code and, thus, the effectiveness of code generation and the performance of the target code.
- Optimisations can range from trivial (e.g. constant folding) to highly sophisticated (e.g, in-lining).
- For example: replace the first two statements in the example of the previous slide with: **`tmp2=4*a`**
- Modern compilers perform such a range of optimisations, that one could argue for:



Code Generation Phase

- Map the AST onto a linear list of target machine instructions in a symbolic form:
  - Instruction selection: a pattern matching problem.
  - Register allocation: each value should be in a register when it is used (but there is only a limited number): NP-Complete problem.
  - Instruction scheduling: take advantage of multiple functional units: NP-Complete problem.
- Target, machine-specific properties may be used to optimise the code.
- Finally, machine code and associated information required by the Operating System are generated.

Some historical notes...

Emphasis of compiler construction research:

- 1945-1960: code generation
  - need to “prove” that high-level programming can produce efficient code (“automatic programming”).
- 1960-1975: parsing
  - proliferation of programming languages
  - study of formal languages reveals powerful techniques.
- 1975-...: code generation and code optimisation

Knuth (1962) observed that “*in this field there has been an unusual amount of parallel discovery of the same technique by people working independently*”

Historical Notes: the Move to Higher-Level Programming Languages

- Machine Languages (1st generation)
- Assembly Languages (2nd generation) – early 1950s
- High-Level Languages (3rd generation) – later 1950s
- 4th generation higher level languages (SQL, Postscript)
- 5th generation languages (logic based, eg, Prolog)
- Other classifications:
  - Imperative (how); declarative (what)
  - Object-oriented languages
  - Scripting languages

See exercise 1.3.1 in Aho2

Finally...

Parts of a compiler can be generated automatically using generators based on formalisms. E.g.:

- Scanner generators: flex
- Parser generators: bison

<u>Summary</u>: the structure of a typical compiler was described.

<u>Next time</u>: Introduction to lexical analysis.

<u>Reading</u>: Aho2, Sections 1.2, 1.3; Aho1, pp. 1-24; Hunter, pp. 1-15 (try the exercises); Grune [rest of Chapter 1 up to Section 1.8] (try the exercises); Cooper & Torczon (1st edition), Sections 1.4, 1.5.