

COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia

<http://www.cse.unsw.edu.au/~cs3131>

<http://www.cse.unsw.edu.au/~cs9102>

Copyright @2025 Jingling Xue

Lectures

1. In-person lectures (with recordings available)
2. Lecture times: may be used for providing feedback to our assignments
3. Moodle used only for viewing the lecture recordings

Outline

1. Administrivia
2. Subject overview
3. Lexical analysis \implies Assignment 1

Important Facts

- Lecturer + Subject Admin:

Name: Jingling Xue

Office: K17 – 501L

Email: j.xue@unsw.edu.au

Important messages will be displayed on the subject home page and urgent messages also sent to you by email.

- Check the course emails/notices **at least every day**
- Check the questions/ansers at the course forum
- Contact me at **j.xue@** rather than ~~{es3131,es9102}@~~

Handbook Entry

COMP3131/9102: Programming Languages and Compilers

Prerequisite: COMP2911 (or good knowledge on OO, C++ and/or Java)

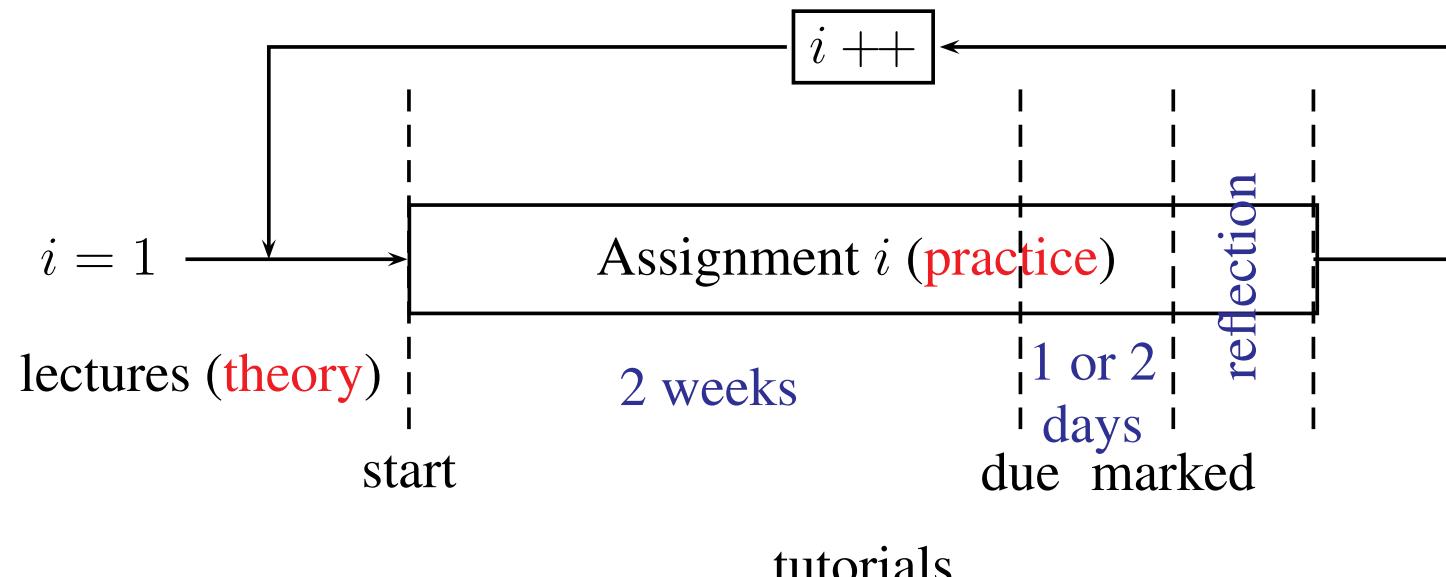
Covers the **fundamental principles in programming languages** and **implementation techniques for compilers** (emphasis on compiler **front ends**). Course contents selected from: program syntax and semantics, formal translation of programming languages, finite-state recognisers and regular expressions, context-free parsing techniques such as LL(k) and LR(k), attribute grammars, syntax-directed translation, type checking, code optimisation and code generation.

Project: implementation of a compiler in a **main-stream** programming language for a **non-trivial** programming language.

A Variant of C \Rightarrow VC

Java

Teaching Strategies



reflection: on i and concepts, fix its bugs, . . .

- Project-centered rather than project-augmented
- Strike a balanced approach between theory and practice

Learning Strategies

- Complete each assignment on time!
- Understand theories introduced in lectures and apply them when implementing your assignment modules
- Tutorials — more examples worked out to ensure your understandings of compiler principles introduced in lectures
- Consultations:
 - Course Forum
 - Individual consultations:
 - * Ask questions by email
 - * Make an individual appointment

Learning Outcomes

- CLO1 : Write a scanner, recogniser, parser, semantic analyser and code generator without the aid of automatic generators
- CLO2 : Understand how to specify the syntax and semantics of a programming language
- CLO3 : Specify and analyse the lexical, syntactic and semantic structures of advanced language features
- CLO4 : Design the data structures and support required for compiling advanced language features
- CLO5 : Gain proficiency in writing sizable object-oriented programs in Java, incorporating the use of packages, inheritance, dynamic dispatching, and the visitor design pattern
- CLO6 : Improve understanding of virtual machines, with a specific focus on the Java virtual machine (JVM)

Studying Materials

- Textbook:
Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman,
Compilers: Principles, Techniques, and Tools, 2/E, Addison Wesley,
2007. ISBN-10: 0321486811. ISBN-13: 9780321486813.
- <https://www.cse.unsw.edu.au/~cs3131>:
<https://www.cse.unsw.edu.au/~cs9102>:
 - Overhead transparencies
 - Supplementary on-line materials
- Reading: suggested in the lecture notes each week

But **lecture notes + tutorial questions and solutions + assignment specs**
should be roughly sufficient for all programming assignments

Textbook (not Compulsory)

- 1st edition: **The Red Dragon Book**
- 2nd edition: **The Purple Dragon Book**
- A reference to, say, Section 3.1 of Dragon book means a reference to both books.
- Otherwise, a specific reference such as “See Section 3.1 of Red Dragon Book or Section 3.3 of Purple Dragon Book” will be used.

Programming Assignments

- Five **compulsory** programming assignments
 - Very detailed specifications
 - Need to follow the specs quite closely
- One **optional** bonus programming assignment
 - Minimal specification
 - Justify your design decisions (if required)

Basic Programming Assignments

Writing a compiler in Java to translate VC into Java bytecode:

1. **Scanner** – reads text and produces **tokens**
2. **Recogniser** – reads tokens and checks syntax errors
3. **Parser** – builds abstract syntax tree (**AST**)
4. **Static Semantics** – checks semantics at compile time
5. **Code Generator** – generates Java bytecode

Notes:

- A description of VC is already available on the home page.
- The **recogniser** is part of the **parser**; separating both simplifies the construction of the parsing component.

Compiler Project Policies

- Policies
 - All are **individual** assignments
 - No illegal collaborations allowed
 - Penalties applied to late assignments

Non-standard penalty rates used due to the nature of this course!

- No **incompletes** – assignment k depends on assignment $k - 1$!
- Class discussions:
 - Course forum: on-line

Plagiarism

- CSE will adopt a uniform set of penalties for all programming assignments in all courses
 - A wide range of penalties
 - See the “Course Outline[link](#) of the course home page

Extensions

- Very few in the past (only genuine reasons considered)
- Why not?
 - Each assignment builds on the previous ones
 - Each assignment will usually be marked within 48 hours of its submission deadline
- The same practice this year

Marking Criteria for Programming Assignments

- Evaluated on correctness by using various test cases.
- Some are provided with each assignment but you are expected to design your own (see
<https://www.cse.unsw.edu.au/~cs3131/25T1/Info/FAQs.html>)
- **No subjective marking**

Lectures

- 10 weeks
- Week 6 (Flexibility Week)
 - no lectures
 - no tutorials

Tutorials

- More on mastering the **fundamental principles** of the subject
- Tutorials starts from week 2
- **Solutions** for week k available on-line in week $k + 1$
- Tutors: Mr Adam Stucci and Owen Chai
- **Online Tutorials:** zoom meeting links on course page

Assessment (Due Dates Tentative)

Component	Marks	Due
Scanner	12	Week 2
Recogniser	12	Week 3/4
Parser	18	Week 5
Static Semantics	30	Week 8
Code Generator	28	Week 10
Final Exam	100	April/May

- **PROGRAMMING:** your marks for all assignments (out of 100)
- **EXAM:** your marks for the exam (out of 100)
- **BONUS:** your bonus marks (out of 5)
- final mark = $\min(0.5 \times P + 0.5 \times E + B, 100)$

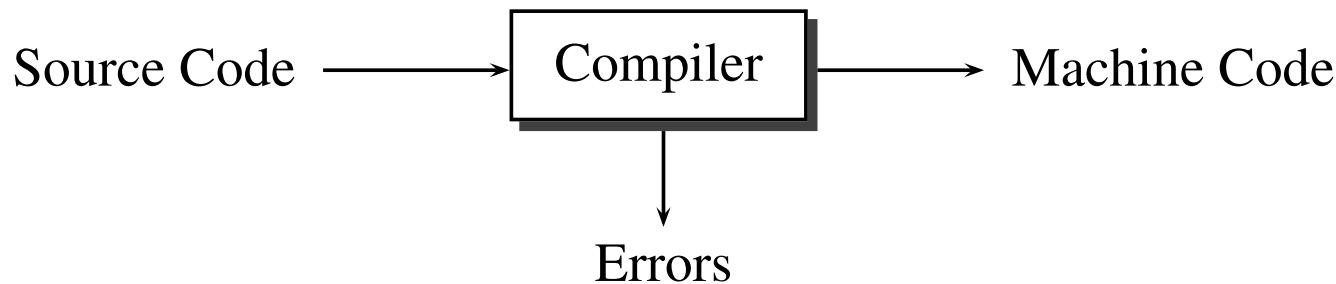
- You will pass the course if you obtain 50 or above.

Recogniser and Parser are two different components of the same assignment.

Outline

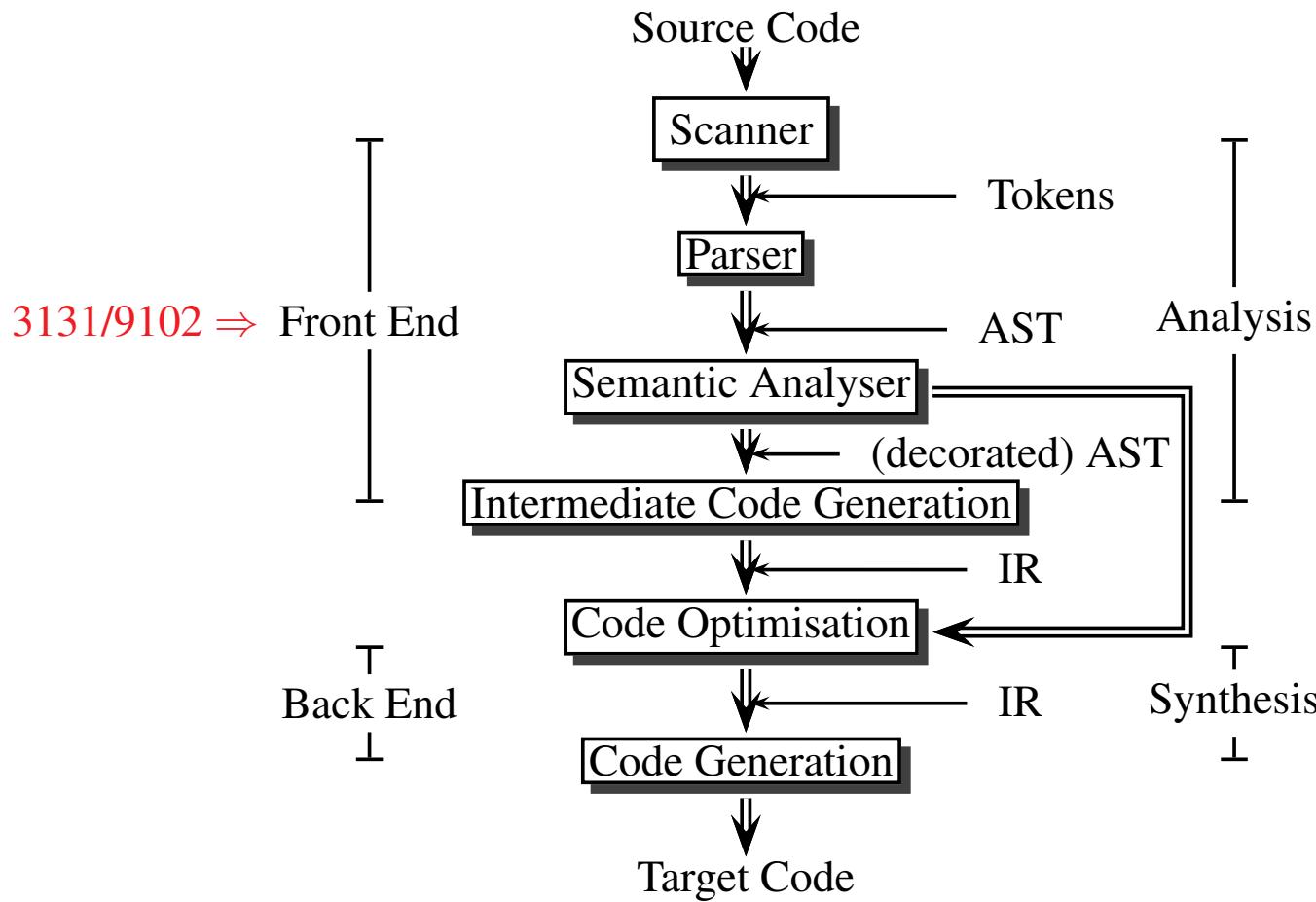
1. Administrivia ✓
2. Subject overview
3. Lexical analysis \Rightarrow Assignment 1

What Is a Compiler?



- recognise legal (and illegal) programs
- generate correct, hopefully efficient, code
- open-source compilers:
 - C/C++: GNU, LLVM, Open64
 - Java: Maxine, Jalapeno
 - Javascript: Google's Closure

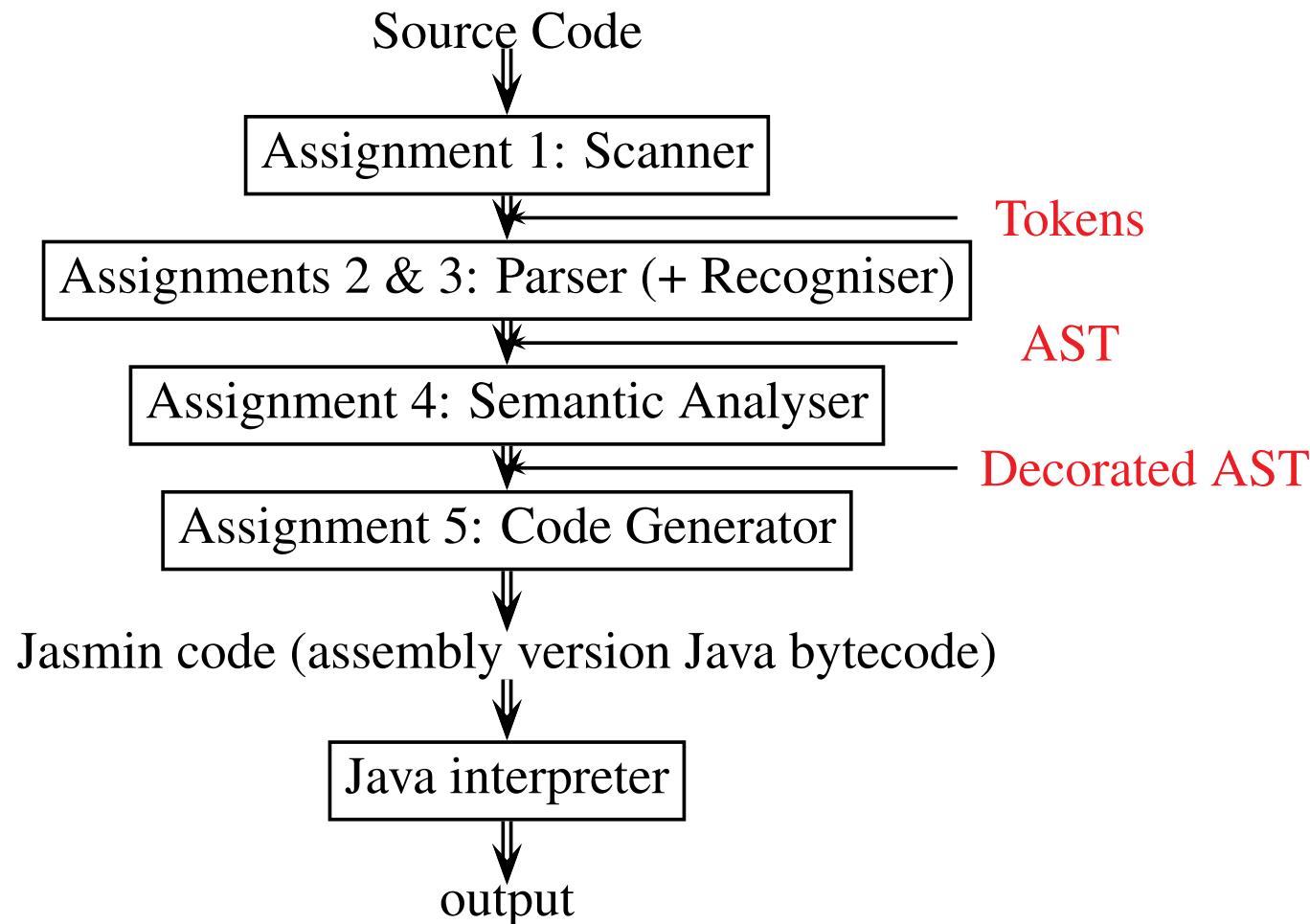
The Typical Structure of a Compiler



Informally, error handling and symbol table management also called “phases”.

- (1) **Analysis**: breaks up the program into pieces and creates an intermediate representation (IR), and
- (2) **Synthesis**: constructs the target program from the IR

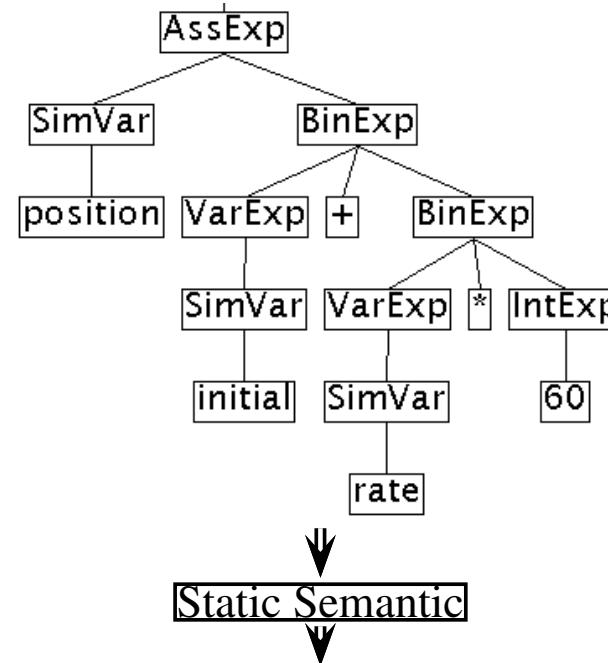
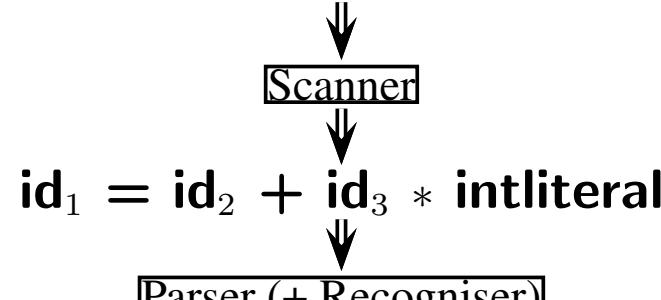
The VC Compiler – Marked Only for Correctness



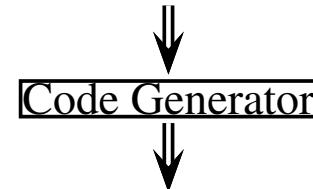
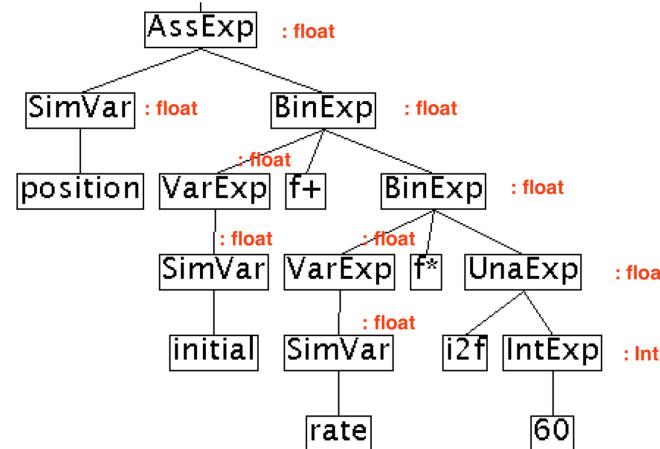
Correspond to the first four components in Slide 22

The Example for JVM (from Our VC Compiler)

position = initial + rate * 60 (all of float type)



The Example for JVM (Cont'd)



	index	variable	value
<hr/>			
<code>float_3</code>			
<code>float_4</code>			
<code>bipush 60</code>			
<code>i2f</code>			
<code>fmul</code>	2	position	
<code>fadd</code>	3	initial	1.0
<code>fstore_2</code>	4	rate	2.0

Lexical Analysis

Scanner

- groups characters into **tokens** – the basic unit of syntax

```
position = initial + rate * 60
```

becomes

1. The identifier position
2. The assignment operator =
3. The identifier initial
4. The plus sign
5. The identifier rate
6. The multiplication sign
7. The integer constant 60.

- character string forming a token is a **lexeme**
- eliminates white space (blanks, tabs and returns)

Syntax Analysis

Parser

- groups tokens into grammatical phrases
- represents the grammatical phases as an AST
- produces meaningful error messages
- attempts error detection and recovery

The syntax of a language is typically specified by a **CFG** (Context-Free Grammar).

The typical arithmetic expressions are defined:

$$\begin{aligned}\langle \text{expr} \rangle &\rightarrow \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{expr} \rangle - \langle \text{term} \rangle \mid \langle \text{term} \rangle \\ \langle \text{term} \rangle &\rightarrow \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle / \langle \text{factor} \rangle \mid \langle \text{factor} \rangle \\ \langle \text{factor} \rangle &\rightarrow (\langle \text{expr} \rangle) \mid \mathbf{ID} \mid \mathbf{INTLITERAL}\end{aligned}$$

Semantic Analysis

Semantic Analyser

- Checks the program for semantic errors
 - variables used before defined
 - operands called with compatible types
 - procedures called with the right number and types of arguments
- An important task: **type checking**
 - reals cannot be used to index an array
 - type conversions when some operand coercions are permitted
- The **symbol table** will be consulted

Name	Type	...
initial	float	...
position	float	...
rate	float	...

(Intermediate) Code Generation

(Intermediate) Code Generator generates an explicit IR

- Important IR properties:
 - ease of generation
 - ease of translation into machine instructions
- Subtle decisions in the IR design have major effects on the speed and effectiveness of the compiler.
- Popular IRs:
 - Abstract Syntax trees (ASTs)
 - Directed acyclic graphs (DAGs)
 - Postfix notation
 - Three address code

Topic, Theory and Tools

Topic	Theory	Tools
lexical analysis	REs, NFA, DFA	scanner generator (lex, JFlex)
syntactic analysis	CFGs, LL(k) and LR(k)	parser generator (yacc, JavaCC, CUP)
semantic analysis	attribute grammars, type checking	formal semantics
code optimisation	loop optimisations, ...	data-flow engines
code generation	syntax-directed translation	automatic code generators (tree tilings)

Error Detection, Reporting and Recovery

- **Detection:**
 - Lexical errors: e.g., "123 \Rightarrow unterminated string
 - Syntax Errors: e.g., forgetting a closing parenthesis
 - Semantic Errors: e.g., incompatible operands for an operator
- **Report** as accurately as possible the locations where errors occur.
- After detecting an error, can **recover** and proceed, allowing further errors in the source program to be detected.

You can optionally implement error recovery in your parser.

VC

- **comments:** Java-like // and /* */
- **Types:**
 - primitive: void, int, float and boolean
 - array: int[], float[], boolean[]
- **variables:** global and local
- **Literals:** integers, reals, boolean, strings
- **Expressions:** conditional, relational, arithmetic and call
- **Statements:** if, for, while, assignment, break, continue, return
- **Functions:** the parameters are passed by value

Read the VC specification to become familiar with the language

Syllabus

1. Lexical analysis

- crafting a scanner by hand
- regular expressions, NFA and DFA
- scanner generator (e.g., lex and JLex)

2. Context-free grammars

3. Syntactic analysis

- abstract syntax trees (ASTs)
- recursive-descent parsing and LL(k)
- bottom-up parsing and LR(k) – **not covered**
- Parser generators (e.g., yacc, JavaCC and JavaCUP)

4. Semantic analysis

- symbol table
- identification (i.e., binding)
- type checking

5. Code generation

- syntax-directed translation
- Jasmin assembly language
- Java Virtual Machines (JVMs)

Lectures (Tentative)

1. Week 1: Intro, lexical analysis, DFAs and NFAs
2. Week 2: CFGs + parsing
3. Week 3: Parsing + Abstract syntax trees (ASTs)
4. Week 4: Attribute grammars
5. Week 5: Static semantics
6. **Week 6 (Flexibility Week): No Lecture & tutorial**
7. Week 7: JVM + Jasmin
8. Week 8: Code generation
9. Week 9: DFAs + NFAs
10. Week 10: Table-Driven Parsing + Revision

What Are Lectures for?

- Introduce new material mostly on the theoretical aspect of compiling
 - REs and parsing \implies automatic scanner and parser generators
 - Usually assessed in the final exam
- Guide you for implementing your VC compiler.
 - Introduce important design issues
 - Explain how to use the supplied classes – but the on-line description of each assignment should mostly suffice

COMP3131/9102 Is Challenging and Fun

- Project is challenging
- One of the few opportunities for writing a large, complex software
- But
 - you learn how languages and compilers work, and
 - you will improve your programming and software engineering skills

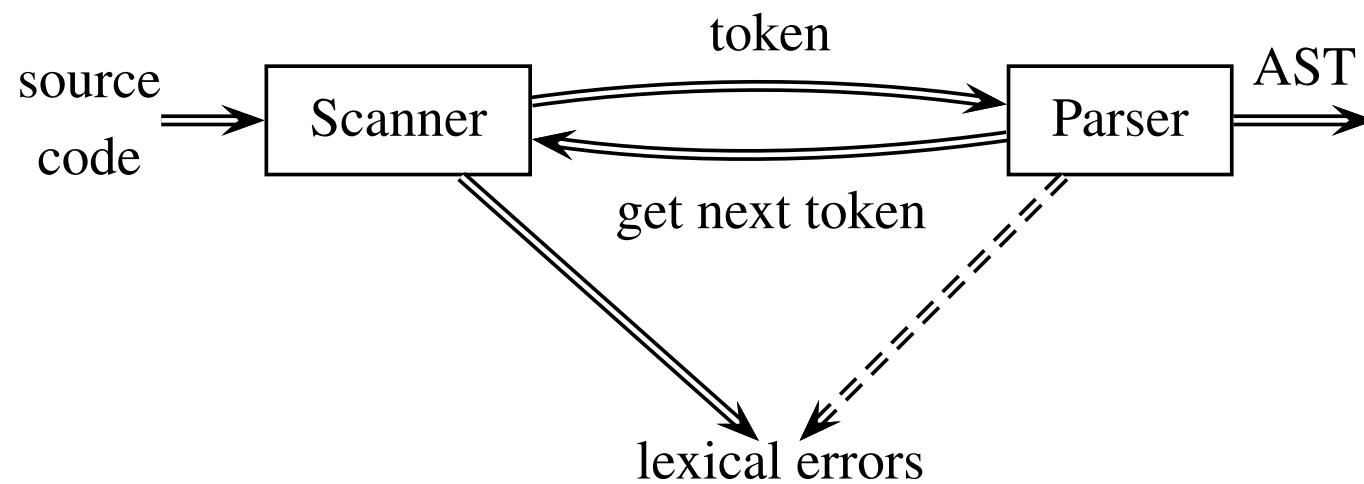
Outline

1. Administrivia ✓
2. Subject overview ✓
3. Lexical analysis \Rightarrow Assignment 1

Lexical Analysis

1. The role of a scanner
2. Import concepts
 - Tokens
 - Lexemes (i.e., spellings)
 - Patterns
3. Design issues in crafting a scanner by hand \implies Assignment 1

The Role of the Scanner



- The **tokens** to programming languages are what the words to natural languages.
- The scanner operates as a subroutine called by the parser when it needs a new token in the input stream.

Tokens

- The tokens in VC are classified as follows:
 - identifiers (e.g., sum, i, j)
 - keywords (e.g., int, if or while)
 - operators (e.g., “+” or “*”, “ \leq ”)
 - separators (e.g., “{”, “}”, “;”)
 - literals (integer, real, boolean and string constants)
- The exact token set depends on the programming language in question (and the grammar used).

The assignment operator token is “:=” in Pascal and “=” in C.

- Analogously, in natural languages, “types” of tokens (i.e., words): verb, noun, article, adjective, etc. The exact word set depends on the natural language in question.

Lexemes (i.e., Spellings of Tokens)

- The **lexeme** of a token: the character sequence (i.e., the actual text of) forming the token.
- Examples:

Token	Token Type	Lexeme
rate_1	ID	rate_1
i	ID	i
+	+	+
<=	<=	<=
while	while	while
100	INTLITERAL	100
1.1e2	FLOATLITERAL	1.1e2
true	BOOLEANLITERAL	true

(Token) Patterns

- **Pattern**: a rule describing the set of lexemes that can represent a particular token.
- The pattern is said to **match** each string in the set.

Token Type	Pattern	Lexeme (i.e., spelling)
INTLITERAL	a string of decimal digits	127, 0
FLOATLITERAL	fill a verbal spec here for C!	127.1, .1, 1.1e2
ID	a string of letters, digits and underscores beginning with a letter or underscore	sum, line_num
+	the character ‘+’	+
while	the letters ‘w’, ‘h’, ‘i’, ’l’, ’e’	while

- Need a formal notation for tokens \Rightarrow REs, NFA, DFA (2nd Lecture this week)
- But today's lecture sufficient for doing Assignment 1

Regular Expressions for Integer and Real Numbers in C

- Integers:

`intLiteral`: digit (digit)*

digit: 0|1|2|...|9

- Reals:

`floatLiteral`: digit* fraction exponent?

| digit+.

| digit+.?exponent

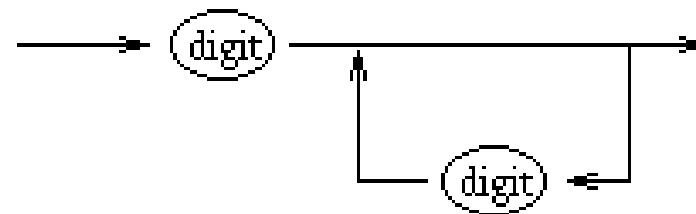
digit: 0|1|2|...|9

fraction: . digit+

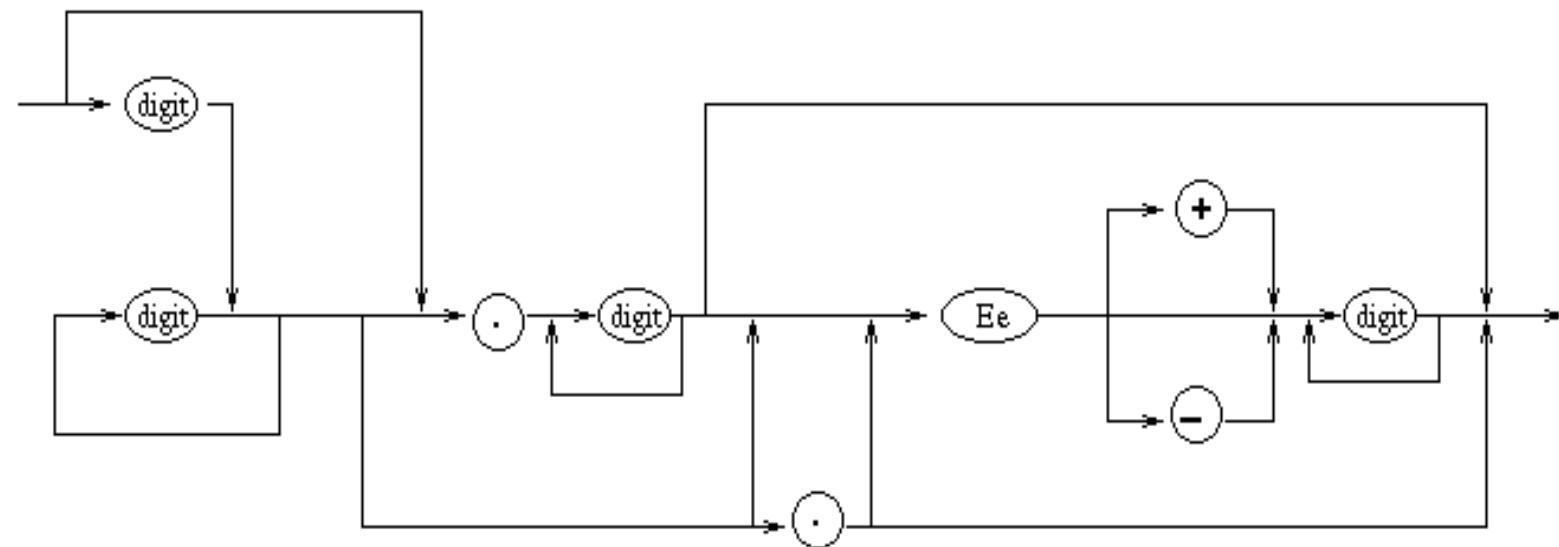
exponent: (E|e) (+|-)? digit+

Finite State Machines for Integers and Reals

- Integers (DFA):



- Reals (NFA):



Assignment 1: Scanner

Scanner.java: a skeleton of the scanner program
(to be completed)

Token.java: The class for representing all the tokens
and for distinguishing between identifiers
and keywords

SourceFile.java: The class for handling the source file

SourcePosition.java: The class for defining the position
of a token in the source file

vc.java: a driver program for testing your scanner

Design Issues in Hand-Crafting a Scanner (for VC)

1. What are the tokens of the language? – see `Token.java`
2. Are keywords reserved? – `yes` in VC, as in C and Java
3. How to distinguish identifiers and keywords? – see `Token.java`
4. How to handle the end of file? – return a special Token
5. How to represent the tokens? – see `Token.java`
6. How to handle whitespace and comments? – throw them away
7. What is the structure of a scanner? – see `Scanner.java`
8. How to detect and recover from lexical errors?
9. How many characters of lookahead are needed to recognise a token?

PL/1 Has No Reserved Words

- A legal but bizarre PL/1 statement:

if then = else then if = then; else then = if

Keywords such as **IF** and **THEN** can be used as identifiers.

- Another legal PL/1 snippet:

```
real      integer;  
integer  real;
```

- Two approaches to distinguishing identifiers from reserved words:
 - The scanner interacts with the parser
 - Regard the identifiers and keyword as having the same token type, leaving the task of distinguishing them to the parser

VC, like C/C++ and Java, has reserved words, as defined in the VC spec and highlighted in Token.java

How to Represent a Token?

Token	Representation
<hr/>	
sum	new Token(Token.ID, "sum", sourcePosition);
123	new Token(Token.INTLITERAL, "123", sourcePosition);
1.1	new Token(Token.FLOATLITERAL, "1.1", sourcePosition);
+	new Token(Token.PLUS, "+", sourcePosition);
,	new Token(Token.COMMA, ",", sourcePosition);

sourcePosition is an instance of the Class SourcePosition:

- charStart: the beginning column position of the token
- charFinish: the ending column position of the token
- lineStart=lineFinish: the number of the line where the token is found.

Blanks Aren't Token Delimiters in FORTRAN

- In the statement

DO 10 I = 1,100

DO is a keyword. However, in the statement

DO 10 I = 1.100

DO10I is an identifier.

The scanner must **look ahead** many characters to distinguish the two cases.

- These Fortran Programs are the syntactically identical:

DO 10 STEP=1, 10 10 WRITE(*,*) 'HELLO!' END	DO10STEP=1, 10 10 WRITE(*,*) 'HELLO!' END
DO 10 S T E P=1, 10 10 WRITE(*,*) 'HELLO!' END	DO 10 STEP=1, 10 10 W RITE(*,*) 'HELLO!' E N D

Whitespaces and comments are token delimiters in VC, as in C/C++ and Java.

The Structure of a Hand-Written Scanner

```
public final class Scanner {  
  
    getToken() {  
  
        // 1. skip whitespace and comments  
        // 2. form the next token  
        switch (currentChar) {  
            case '(':  
                accept();  
                return the token representation for '('  
  
            case '<':  
                accept();  
                if (currentChar == '=') {  
                    accept(); // get the next char  
                    return the token representation for '<='  
                } else  
        }  
    }  
}
```

```
        return the token representation for '<'

    case '.':
        attempting to recognise a float

    ...

    default:
        return an error token
    }

    ...

    return new Token(kind, spelling, sourcePosition);
}
```

You need to think about how to recognise efficiently ids, keywords, integers, etc.

My accept() Method in the Scanner Class

```
private void accept() {  
    currentChar = sourceFile.getNextChar();  
  
    inc my counter for the current line number, if necessary  
  
    inc my counter for the current char column number  
  
    perhaps, you can also accumulate the current lexeme here  
}
```

Make sure you count tabs in terms of the number of blank spaces correctly.
Read the FAQs for Assignment 1.

Maintaining Two Scanner Invariants

Every time when the scanner is called to return the next token:

1. `currentChar` is pointing to either the beginning of
 - some whitespace or
 - some comment or
 - a token
2. Scanner always returns the **longest possible match** in the remaining input

Input	Tokens
<code>>=</code>	<code>">="</code> not <code>">"</code> and <code>"="</code>
<code>//</code>	end-of-line comments not <code>"/"</code> and <code>"\\"</code>

Lookahead

1.2e+ 3 ---> "1.2" "e" "+" "3" (four tokens)

~~~~~

||||

| \

| |

| +---- three chars of lookahead required:

"e", "+" and " "

current char

The output from your scanner:

```
Kind = 35 [<float-literal>], spelling = “1.2”, position = 1(1)..1(3)
Kind = 33 [<id>], spelling = “e”, position = 1(4)..1(4)
Kind = 11 [+], spelling = “+”, position = 1(5)..1(5)
Kind = 34 [<int-literal>], spelling = “3”, position = 1(7)..1(7)
Kind = 39 [$], spelling = “$”, position = 2(1)..2(1)
```

// the following function is provided in the supporting code

```
private char inspectChar(int nthChar) {
    return sourceFile.inspectChar(nthChar);
}
```

## Maximal Munch (in C++)

- Each token formed is the longest possible

- Consequences:

- Syntactically legal:

i---j ==> i-- - j ;

- Syntactically legal only if some spaces are in between:

```
template <typename T, typename CON=deque<T>>
class stack { ... }
```

- But the following is ok now in C++11:

```
template <typename T, typename CON=deque<T>>
```

## Lexical Errors

Lexical errors (see the Assignment 1 spec):

- (1) /\* -> prints an error message (unterminated comment)
- (2) |, ^, %, etc. -> returns an error token and continues lexical analysis
- (3) Unterminated string
- (4) Illegal escape character

## Reading

- Textbook: Chapter 1 and Sections 3.1 – 3.2
- Read the “Course Outline” on the course home page:
- Assignment 1 spec is available on the subject home page

**Next Class:** Regular Expressions, NFA and DFA

Our Course Learning Outcomes (CLOs) can be refined in the next three slides.

## Learning Objectives

- Learn important compiler techniques, algorithms and tools
- Learn to use compilers (and debuggers) more efficiently
- Improve understanding of program behaviour  
(e.g., syntax, semantics, typing, scoping, binding)
- Improve programming and software engineering skills  
(e.g., OO, visitor design pattern)
- Learn to build a large and reliable system
- See many basic CS concepts at work
- Prepare you for some advanced topics, e.g., compiler backend optimisations (for GPUs, FPGAs, multicores, embedded processors)

## Knowledge Outcomes

- finite state automata and how they relate to regular expressions
- context free grammars and how they relate to context-free parsing
- formal language specification strategies
- top-down and bottom-up parsing styles
- attribute grammars
- type checking
- Java virtual machine (JVM)
- code generation techniques
- visitor design pattern

## Skill Outcomes

- ability to write scanners, parsers, semantic analysers and code generators
- ability to use compiler construction tools: lexers + parsers
- understand how to specify the syntax and semantics of a language
- understand code generation
- understand and use the data structures and algorithms employed within the compilation process
- ability to write reasonably large OO programs in Java using packages, inheritance, dynamic dispatching and visitor design pattern
- understand virtual machines, in particular, JVM