Five-Minute Review

1. What is a *variable*?
2. What is a *class*? An *object*?
3. What is a *package*?
4. What is a *method*? A *constructor*?
5. What is an *object variable*?

Programming – Lecture 3

Expressions *etc.* (Chapter 3)

- Aside: Context Free Grammars
- Expressions
- Primitive types
- Aside: representing integers
- Constants, variables
- Identifiers
- Variable declarations
- Arithmetic expressions
- Operator precedence
- Assignment statements
- Booleans

Aside: Context-Free Grammars (CFGs)

Can specify **syntax** of a program (or parts of a program) as CFG

Note: “Aside” indicates that this material is not covered in the book, but still part of the class content, also relevant for exam.

For further reference, see e.g.:
https://en.wikipedia.org/wiki/Context-free_grammar

Why You Should Care About CFGs

The Java® Language Specification

Table of Contents

1. Introduction
1.1. Organization of the Specification
1.2. Example Programs
1.3. Notation
1.4. Relationship to Predefined Classes and Interfaces
1.5. Feedback
1.6. References
2. Grammars
2.1. Context-Free Grammars
2.2. The Lexical Grammar
2.3. The Syntactic Grammar
2.4. Grammar Notation
...

Context-Free Grammars (CFGs)

From the Java Language Standard, Sec. 2.1:

A *context-free grammar* consists of a number of *productions*.

Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

https://docs.oracle.com/javase/specs/jls/se9/html/jls-2.html#jls-2.1

Context-Free Grammars (CFGs)

Formally: *CFG* defined by 4-tuple $G = (V, \Sigma, R, S)$

- V is a set of *nonterminal characters* or *variables*
- Σ, the *alphabet*, is finite set of *terminals*.
- R, the set of *(rewrite) rules* or *productions*, is relation from V to $(V \cup \Sigma)^*$, i.e., a set of ordered pairs of elements from V and $(V \cup \Sigma)^*$, respectively
- $S \in V$ is the *start variable* (or *start/goal symbol*)

Note: * is the *Kleene Star*. For any set X, X^* denotes the set of strings composed of concatenating 0 or more elements of X.

Example: $\{ 0, 1 \}^* = \{ \varepsilon, 0, 1, 00, 01, 10, 11, 000, ... \}$, where ε denotes the *empty string*

Language of CFG

For any strings u, $v \in (V \cup \Sigma)^*$,
u *directly yields* v (written $u \Rightarrow v$)
if $\exists(\alpha, \beta) \in R$ with $\alpha \in V$ and u_1, $u_2 \in (V \cup \Sigma)^*$ and
$u = u_1 \alpha u_2$ and $v = u_1 \beta u_2$.
Thus, v is a result of applying the rule (α, β) to u.

Language of grammar $G = (V, \Sigma, R, S)$ is the set
$L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$
where $\Rightarrow^*$ is reflexive transitive closure of $\Rightarrow$

Example: Well-Formed Parentheses

Well-formed: (), (()), ()(), ()(()), …

Ill-formed: ε, (,),)(, ((), …

$G = (V, \Sigma, R, S)$ with

- Variables $V = \{ S \}$
- Alphabet $\Sigma = \{$ (,) $\}$
- Productions $R = \{ S \rightarrow SS, S \rightarrow (S), S \rightarrow () \}$

May also write R as $S \rightarrow SS \mid (S) \mid ()$

Note: one may argue that ε is also well-formed – which could also be handled with a further production $S \rightarrow \varepsilon$

$S \rightarrow SS \mid (S) \mid ()$

Claim: The string (()()) is *valid*, i.e., in $L(G)$.

Proof: consider the *derivation*

$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (()S) \Rightarrow (()())$

However, the string)(is not in $L(G)$,
since there is no derivation from S to)(

Trees in CS

Our trees grow downwards!

R: *Root*

L: *Leaf*

I: *Internal node* (i.e., not a leaf)

Typically, root is an internal node (when not?)

```
      R
    / | \
   L  I  L
     / \
    I   I
   / \ / \
  L  L L  L
```

Parse Trees

May use *parse trees* as compact representation for derivation.

Internal nodes are variables,
Leafs are terminals.

$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow (()S) \Rightarrow (()())$

is a derivation that follows from parse tree on right.

Recall: $S \rightarrow SS \mid (S) \mid ()$

```
       S
      /|\
     ( S )
      / \
     S   S
    / \ / \
   (  )(  )
```

Example: Parenthesized Sums

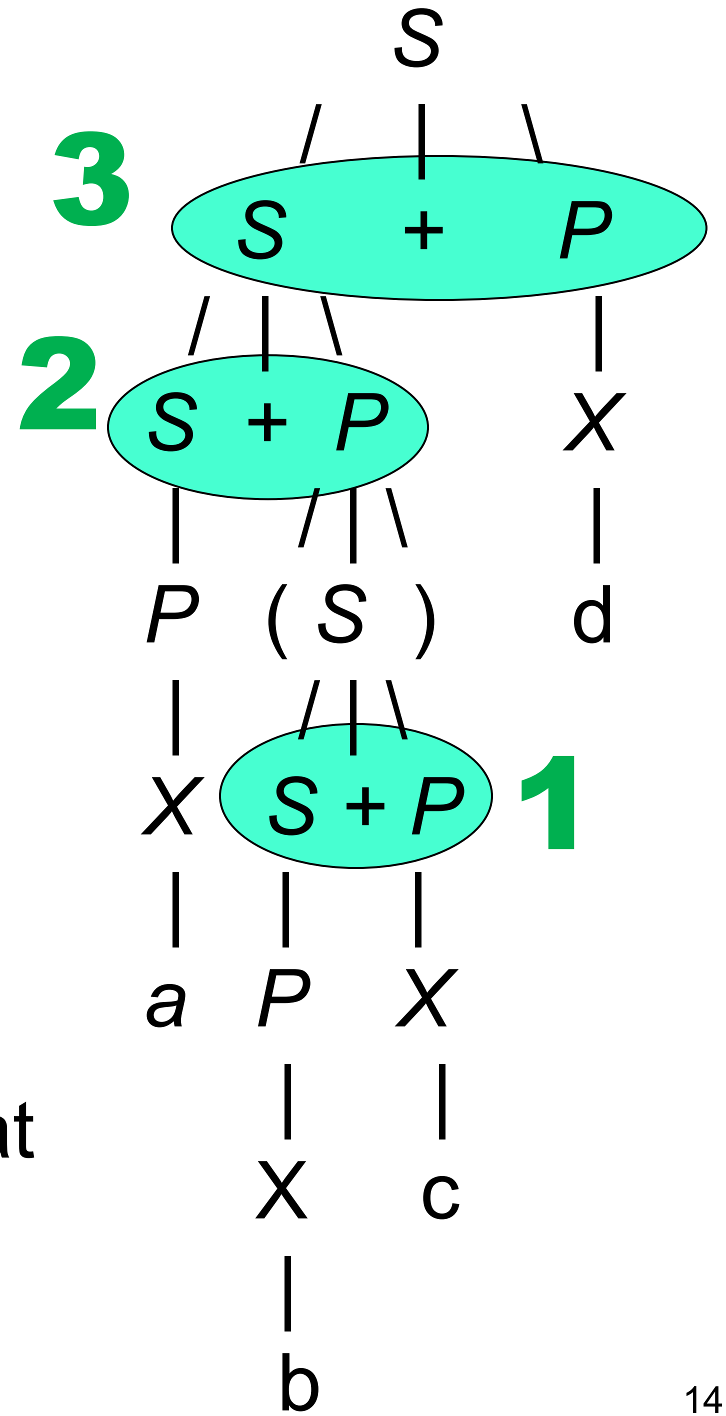
a + b, u, x + (y + z), ...

$G = (V, \Sigma, R, S)$ with

- Variables $V = \{ S, P, X \}$
- Alphabet $\Sigma = \{$ (,), +, a, ..., z $\}$
- Productions:
  $S \rightarrow S + P \mid P$
  $P \rightarrow (S) \mid X$
  $X \rightarrow \text{a} \mid \ldots \mid \text{z}$

$S \rightarrow S + P \mid P$
$P \rightarrow (S) \mid X$
$X \rightarrow \text{a} \mid \ldots \mid \text{z}$

Parse tree for a + (b + c) + d:



Parsing done bottom-up; lower position in parse tree is parsed/evaluated earlier

Parentheses evaluated first

Note that *above rules* imply that + is evaluated left-to-right (*left-associative*)

Note on Notation

Recall: formally, set of productions is a relation.

Can write this in different ways:

Set notation:

$R = \{ (S, SS), (S, (S)), (S, ()) \}$

Verbose arrow notation:

$S \rightarrow SS,\ S \rightarrow (S),\ S \rightarrow ()$

Compact arrow notation:

$S \rightarrow SS \mid (S) \mid ()$

Multiline notation:

S:

SS

(S)

$()$

Context-Free Languages

L is a *context-free language* (CFL),
iff ("if and only if") there exists a CFG G,
such that $L = L(G)$

Example: Is $L_2 = \{ a^n b^n : n \in \mathbb{N} \}$ context-free?

Yes, $L_2 = L((\{ S \}, \{ a, b \}, \{ (S, aSb), (S, \varepsilon) \}, S))$

Example: Is $L_3 = \{ a^n b^n c^n : n \in \mathbb{N} \}$ context-free?

No, there is no CFG G with $L_3 = L(G)$.

Proof: see
https://en.wikipedia.org/wiki/Pumping_lemma_for_context-free_languages

Note: CFLs are a superset of *regular languages*. E.g., L_2 is not regular.

So, is Java context free?

No.

CFGs don't address, e.g., variable declarations/bindings.

But CFGs make the *syntax* precise, which is important both for programmers and parsers.

Backus-Naur Form (*BNF*)

BNF is another notation for CFGs

- Close to compact arrow notation
- Use "::=" instead of arrow, "<...>" for variables

Well-formed parentheses example in BNF:

<*S*> ::= <*S*><*S*> | (<*S*>) | ()

Extended Backus-Naur Form (*EBNF*)

Typically puts terminals into quotes (" or ')

Typically no "<...>" for variables

[*X*] denotes 0 or 1 occurrences of *X*

S ::= a [b] c abbreviates S ::= a c | a b c

{*X*} denotes 0 or more occurrences of *X*

S ::= a {b} c abbreviates S ::= a T c, T ::= b T | ε

(*X*) defines a group

S ::= a (b | c) d abbreviates S ::= a b d | a c d

Java Lexical Grammar

- Is a CFG
- Terminals are from Unicode character set
- Translate into *input symbols* that, with whitespace and comments discarded, form terminal symbols (*tokens*) for *Java Syntactic Grammar*
- Notation is variant of EBNF

See also https://docs.oracle.com/javase/specs/jls/se9/html/jls-2.html#jls-2.4

Example: Java Decimal Numerals

- Want to prohibit leading 0 (except in 0 itself), to avoid clash with octal numeral
- Therefore, must be 0 or begin with non-zero
- Allow underscores, but not at beginning or end

```
DecimalNumeral:
  0
  NonZeroDigit [Digits]
  NonZeroDigit Underscores Digits

NonZeroDigit:
  (one of)
  1 2 3 4 5 6 7 8 9

Digits:
  Digit
  Digit [DigitsAndUnderscores] Digit

DigitsAndUnderscores:
  DigitOrUnderscore {DigitOrUnderscore}

Digit:
  0
  NonZeroDigit

DigitOrUnderscore:
  Digit
  _

Underscores:
  _ {_}
```

https://docs.oracle.com/javase/specs/jls/se9/html/jls-3.html#jls-DecimalNumeral

Expressions

```
int total = n1 + n2;
```

Expression: consists of *terms* (**`n1`**, **`n2`**), or *operands*, joined by *operators* (**`+`**, **`*`**, **`=`**, ...)

Term:
- *Literal*, a.k.a. (*unnamed*) *constant* (**`3.14`**)
- Variable (**`n1`**), including *named constants* (**`PI`**, as in **`static final PI = 3.14`**)
- Method call (**`Math.abs(n1)`**)
- Expression enclosed in parentheses

Primitive Types

Data type: set of values (*domain*) + set of operators

Type	*Domain*	*Common operators*
`byte`	8-bit integers in the range –128 to 127	*The arithmetic operators:* `+` add `*` multiply `–` subtract `/` divide `%` remainder
`short`	16-bit integers in the range –32768 to 32767	
`int`	32-bit integers in the range –2146483648 to 2146483647	*The relational operators:* `==` equal to `!=` not equal `<` less than `<=` less or equal `>` greater than `>=` greater or equal
`long`	64-bit integers in the range –9223372036754775808 to 9223372036754775807	
`float`	32-bit floating-point numbers in the range $\pm 1.4 \times 10^{-45}$ to $\pm 3.4028235 \times 10^{38}$	*The arithmetic operators except* `%` *The relational operators*
`double`	64-bit floating-point numbers in the range $\pm 4.39 \times 10^{-322}$ to $\pm 1.7976931348623157 \times 10^{308}$	
`char`	16-bit characters encoded using Unicode	*The relational operators and* +, -, ...
`boolean`	the values `true` and `false`	*The logical operators:* `&&` and `||` or `!` not *The relational operators:* `==` equal to `!=` not equal

Numbers

This is covered further in Ch. 7

Decimal, binary, octal, hexadecimal notation

$42_{10} = 00101010_2 = 52_8 = 2A_{16}$

K, M, G, T

Decimal: 10^3, 10^6, 10^9, 10^{12}

Binary: 2^{10}, 2^{20}, 2^{30}, 2^{40}

In Java:

Prefix "0"/"0x" means octal/hex literal

$012 \triangleq 10$, $0x12 \triangleq 18$

Aside: Encoding Integers

Computers represent integers in w *bits* $x_i \in \{ 0, 1 \}$

$X = x_{w-1}\, x_{w-2} \ldots x_1\, x_0$

For unsigned int's, X encodes value $B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$

E.g., $B2U(101) = 1*4 + 0*2 + 1*1 = 5$

For signed int's, X encodes $B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$

This is *two's complement* encoding

E.g., for w=3, $B2T(101) = –1*4 + 0*2 + 1*1 = –3$

Sign bit

In Java: w = 8 (**`byte`**), 16 (**`short`**/**`char`**), 32 (**`int`**), or 64 (**`long`**)

In Java, all integral types are signed, except for **`char`**

See also https://docs.oracle.com/javase/specs/jls/se9/html/jls-4.html#jls-4.2

Bit-Wise Operators

`byte x = 42;` // **`x`** encoded as $0010\ 1010_2$

`byte y = 15;` // **`y`** encoded as $0000\ 1111_2$

`byte z = -16;` // **`z`** encoded as $1111\ 0000_2$

Bit-wise operators refer to binary encodings

AND: **`x & y`** = 10 // $0000\ 1010_2$

OR: **`x | y`** = 47 // $0010\ 1111_2$

Shift left: **`y << 2`** = 60 // $0011\ 1100_2$

Arithmetic shift right: **`y >> 2`** = 3 // $0000\ 0011_2$

`z >> 2` = -4 // $1111\ 1100_2$

Logical shift right: **`y >>> 2`** = 3 // $0000\ 0011_2$

`z >>> 2` = 60 // $0011\ 1100_2$

More on bit-wise operators in Lecture 7

Abstract Data Types (ADTs)

ADT = set of variables
+ set of operations
+ specification

Specification may be informal prose and/or mathematical equations that must hold (e.g., commutative/distributive/associative laws).

ADT *abstracts* from implementation.

In Java, typically *implement* ADT as class.

Identifiers

Identifier: name of variable, class, method etc.

- Must begin with letter or underscore
- Remaining characters must be letters, digits, or underscores
- Must not be one of Java's reserved words:

`abstract`	**`continue`**	**`for`**	**`new`**	**`switch`**
`assert`	**`default`**	**`goto`**	**`package`**	**`synchronized`**
`boolean`	**`do`**	**`if`**	**`private`**	**`this`**
`break`	**`double`**	**`implements`**	**`protected`**	**`throw`**
`byte`	**`else`**	**`import`**	**`public`**	**`throws`**
`case`	**`enum`**	**`instanceof`**	**`return`**	**`transient`**
`catch`	**`extends`**	**`int`**	**`short`**	**`try`**
`char`	**`final`**	**`interface`**	**`static`**	**`void`**
`class`	**`finally`**	**`long`**	**`strictfp`**	**`volatile`**
`const`	**`float`**	**`native`**	**`super`**	**`while`**

Coding Advice – Naming Conventions

Classes: `UpperCamelCaseNouns`

Methods: `lowerCamelCaseVerbs`

Constants: `UPPER_CASE`

Variables: `lowerCamelCase`

Avoid single-character variable names, except for "temporary" ones:

- integers: `i`, `j`, `k` ...
- char's: `c`, `d`, `e` ...

Try to use English names:
e.g., use `counter` instead of `zaehler`

See also https://en.wikipedia.org/wiki/Naming_convention_(programming)
http://www.oracle.com/technetwork/java/codeconventions-135099.html#367

Variable Variations

Local variable: declared within method

Instance variable (or *non-static field*): declared as part of a class (without **`static`**), one per object

Class variables (or *static field*): declared as part of class (with **`static`**), only one for class

Scoping

Scope: part of program where variable is visible

Scope of local variables: from declaration until end of enclosing *block* (sequence of statements enclosed in braces, see Lec. 4)

Shadowing (or *hiding*): multiple variables of same name have overlapping scope.

In Java:

- local variables shadow fields
  (useful e.g. for *setters*, see later)
- *no* shadowing of local variables
  (local variable names must be unique within method, unlike e.g. for functions in C)

Operators and Operands

Binary operators – take two operands

+, -, /, *, ==, <, >, &&, ||, &, |, ^, <<, >>, ...

Unary operators – take one operand

+, -, ++, --, !

Ternary operator – takes three operands

? :

Type Casts

`int` *op* `int` ⟹ `int`

`int` *op* `double` ⟹ `double`

`double` *op* `double` ⟹ `double`

```
double c = 100;
double f = 9 / 5 * c + 32;
```

Casting: `(`*type*`)` *expression*

```
double f = (double) 9 / 5 * c + 32;
```

Aside: Expression Evaluation

Different operators may be ordered by *precedence*:

An operand between operators of different precedence is bound to operator of higher precedence

* has *higher* precedence than +

```
2 + 3 * 4  == 2 + (3 * 4) !=  (2 + 3) * 4
```

3 bound to *, not to +

Operators of same precedence level ordered by *associativity*:

+ is *left*-associative, operands between +'s bound to left +

```
1 + 1E100 + -1E100 == (1 + 1E100) + -1E100
                   !=  1 + (1E100 + -1E100)
false + true + ""  == (false + true) + ""
                   !=  false + (true + "")
```

1E100/**true** bound to left +, not to right +

See also http://introcs.cs.princeton.edu/java/11precedence/

Level	Operator	Description	Associativity

16	`[]`
	`.`
	`()`
		access array element
		access object member
		parentheses
			left to right

15	`++` `--`	unary post-increment unary post-decrement	not associative

14	`++`
	`--`
	`+`
	`-`
	`!`
	`~`
		unary pre-increment
		unary pre-decrement
		unary plus
		unary minus
		unary logical NOT
		unary bitwise NOT
			right to left

13	`()` `new`	cast object creation	right to left
12	`* / %`	multiplicative	left to right
11	`+ -` `+`	additive string concatenation	left to right

[http://introcs.cs.princeton.edu/java/11precedence/]

10	<< >> >>>	shift	left to right
9	< <= > >= instanceof	relational	not associative
8	== !=	equality	left to right
7	&	bitwise AND	left to right
6	^	bitwise XOR	left to right
5	|	bitwise OR	left to right
4	&&	logical AND	left to right
3	||	logical OR	left to right
2	?:	ternary	right to left
1	= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	right to left

[http://introcs.cs.princeton.edu/java/11precedence/]

Aside: Expression Evaluation

Precendence and associativity ...

- govern which operands belong to which operator
- imply paren's
- can be overridden by paren's

Precedence, associativity and paren's tell us how to construct a *fully parenthesized* expression,
which makes all bindings of operands to operators explicit:

```
2 + 3 * (4 + 5)   ==   2 + (3 * (4 + 5))
```

Once expression is fully parenthesized, don't need to consider precedence and associativity any more.

Aside: Expression Evaluation

To perform an operation, we **first** evaluate operands, **then** apply operator to results.

(Special case: short-circuit evaluation for &&, || – see later)

Do this *recursively*: if evaluating an operand entails performing an operation, the same rule applies again.

Operands of operator ordered by *evaluation direction*:

Java evaluates *left-to-right* (undefined in C or C++!)

This matters when operand evaluation has *side effects* (such as assigning new values to variables)

With `i` initially 0: `i + 2 * ++i == 2`

Wait a minute ... `*` has *higher* precedence than `+`, but operands of `*` are evaluated *after* left operand of `+`?

Explanation: evaluation direction, see next slide

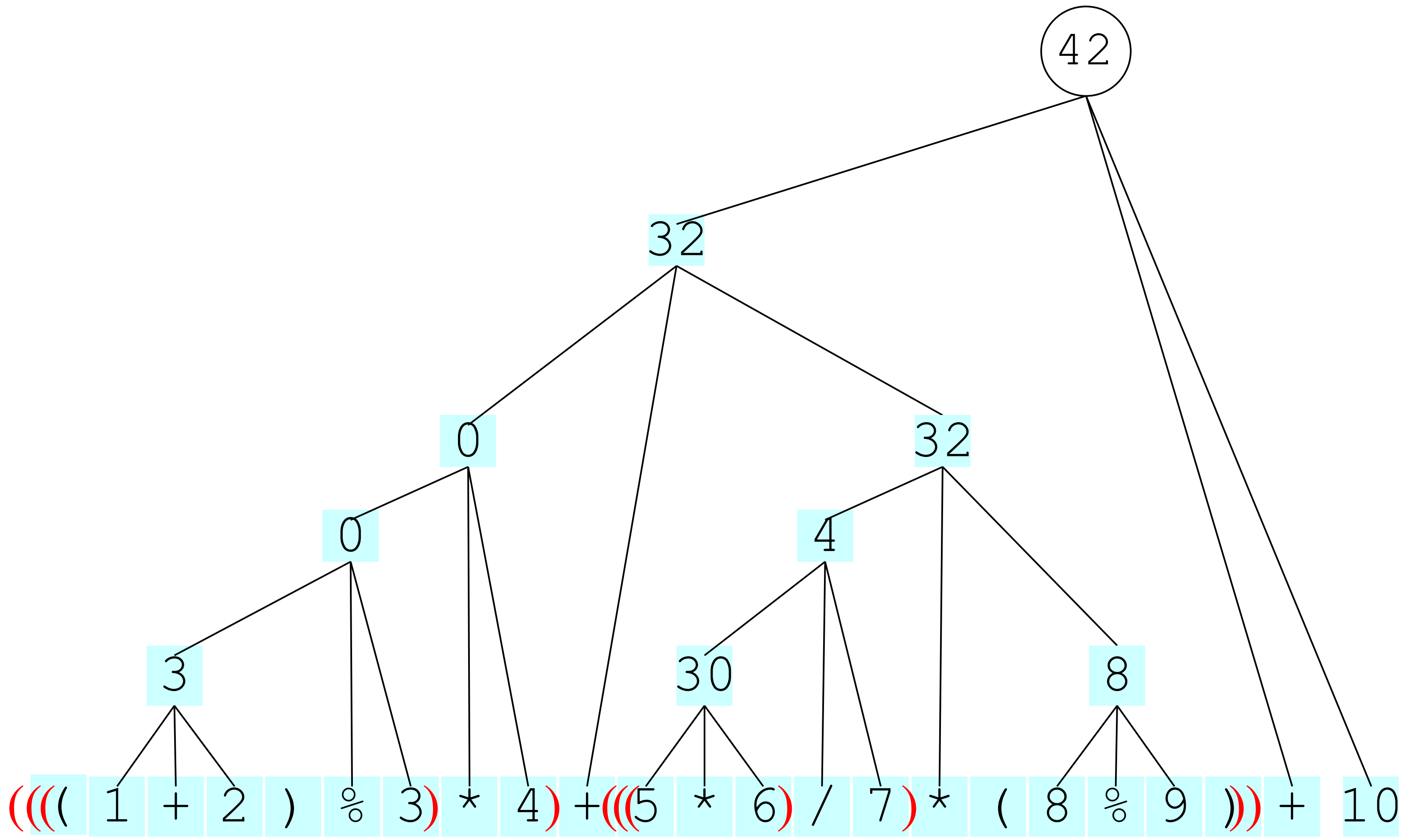
Aside: Expression Evaluation

What happens exactly:

- Fully parenthesized expression: `i + (2 * (++i))`
- We thus have a sum with 2 operands.
- To compute sum, we **first** evaluate the **left** operand, **then** evaluate the **right** operand, **then** compute the sum of both.

1. Evaluating **left** operand `i` yields `0 + (2 * (++i))`
2. **Right** operand `2 * (++i)` is a product, with again 2 operands – thus recursively apply left-to-right rule:
   1. **Left** operand `2` of product is already evaluated: `0 + (2 * (++i))`
   2. Evaluating **right** operand `++i` of product sets `i` to 1 (pre-increment), and yields `0 + (2 * 1)`
   3. Computing **product** yields `0 + 2`
3. Computing **sum** yields `2`

Exercise: Precedence Evaluation



To get started: we have a sum (root of parse tree), whose left operand is another sum, whose left operand is a product, whose left operand is a modulo operation, whose left operand is the sum "**`1 + 2`**".

Coding Advice – Naming, Paren's

- Use meaningful variable names
- Don't use "magic numbers",
  use named constants instead
- Add paren's if precedence may not be obvious

Example: Replace

```
help || me == read && that != thing
```

by

```
help || ((me == read) && (that != thing))
```

Assignments

variable `=` *expression* `;`

Shorthand assignment:

variable *op*`=` *expression* `;`

`int x = 0; x += 1.0;` is equivalent to

`int x = 0; x = (int) (x + 1.0);`

Omitting the `(int)` cast would result in an error

Pre-*increment*

`++`*variable*`;`

`++x;` equivalent to `x += 1;`

`y = ++x;` equivalent to

`x += 1; y = x;`

Post-*increment*

variable`++;`

`x++;` equivalent to `x += 1;`

`y = x++;` equivalent to

`y = x; x += 1;`

Assignment Expressions

- Assignments are also expressions, with assignment operator (=, +=, etc.)
- Left operand must be an “L-Value”, i.e., something that points to a storage location, i.e., a variable
- Assigned value is also value of assignment expression

```
int x, y = (x = 1) + (y = 2) + (x += 3);
```

results in x = 4, y = 7

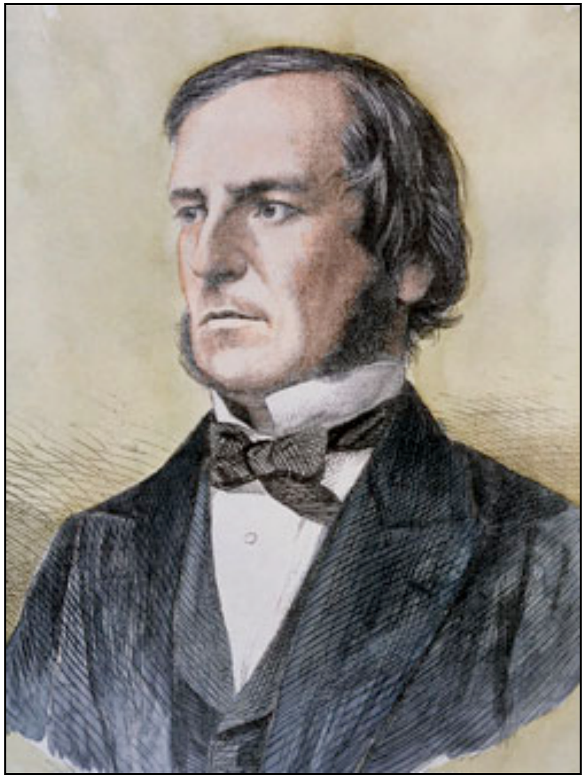
Coding advice: don’t use shorthand assignments (including pre/post-increment etc.) as expressions

Bad: y = x++;

Good: x++; y = x;

Booleans

Boolean values: **`true, false`**



George Boole
(1791-1871)

Logical operators on Booleans:

`&&   ||`

These *short-circuit*:
right operand evaluated only when needed

Other logical operators on Booleans:

`==   !=   !   &   |   ^`

These don't short-circuit

Relational operators producing Booleans:

`<   <=   ==   >=   >   !=`

Coding Advice – Don't confuse "=" and "=="

```
if (oneFlag = otherFlag) {
   ...
}
```

If you *really* mean this, write instead:

```
oneFlag = otherFlag;
if (oneFlag) {
   ...
}
```

But what was *probably* meant:

```
if (oneFlag == otherFlag) {
   ...
}
```

Summary

- Expressions = terms + operators
- Primitive data types: **`int`**, **`double`**, ...
- Simplest terms: constants, variables
- Declarations: ***`type name = value;`***
- Expression evaluation: paren's, precedence, associativity and (in Java) left-to-right evaluation
- Assignments: ***`variable = expression;`***
- Relational operators produce Booleans
- Can operate on Booleans

From Next Week Onwards – We Will Move!

- For both *Vorlesung* and *Globalübung*
- Old: https://uni-kiel.zoom.us/j/85625455567?pwd=SFhGbTcrdGZNVndzenZXdjVmd09GUT09
- New: https://uni-kiel.zoom.us/j/87923834205?pwd=SmV3TDJWWjg2bkIycXVTVWR3blEwUT09