Syntax of Programming Languages

Syntax and Semantics of Expressions

Loosely speaking:

- ***Syntax*** means “form”.
- ***Semantics*** means “meaning”.

In the context of programming languages:

Loosely speaking:

- ***Syntax*** means “form”.
- ***Semantics*** means “meaning”.

In the context of programming languages:

- The Java expressions `a – 3 – b` and `u – 7 – v` have the same *syntax*, but *different semantics*.

Loosely speaking:

- ***Syntax*** means “form”.
- ***Semantics*** means “meaning”.

In the context of programming languages:

- The Java expressions `a – 3 – b` and `u – 7 – v` have the *same syntax*, but *different semantics*.
- The Java expressions `a – 3 – b` and `((a – 3) – b)` have the *same semantics*, but *different syntax*.

Loosely speaking:

- ***Syntax*** means “form”.
- ***Semantics*** means “meaning”.

In the context of programming languages:

- The Java expressions `a – 3 – b` and `u – 7 – v` have the *same syntax*, but *different semantics*.
- The Java expressions `a – 3 – b` and `((a – 3) – b)` have the *same semantics*, but *different syntax*.
- The Lisp expression `(– (– 2 3) 4)` and the Java expression `2 – 3 – 4` have the *same semantics* but have *different syntax*.

Operators, Arities, and Operands

An ***operator of arity*** k, also called a k-***ary operator***, is a symbol that represents a function of k arguments.

Operators, Arities, and Operands

An ***operator of arity*** k, also called a k-***ary operator***, is a symbol that represents a function of k arguments.

- ***Binary*** means 2-ary.
- ***Unary*** means 1-ary.
- ***Ternary*** means 3-ary.

Operators, Arities, and Operands

An ***operator of arity*** *k*, also called a *k*-***ary operator***, is a symbol that represents a function of *k* arguments.

- ***Binary*** means 2-ary.
- ***Unary*** means 1-ary.
- ***Ternary*** means 3-ary.

When a *k*-ary operator appears in an expression, each of the *k* subexpressions whose values are the *k* arguments of the function it represents is called an ***operand*** of the operator.

Examples

Operators, Arities, and Operands

An ***operator of arity*** *k*, also called a *k*-***ary operator***, is a symbol that represents a function of *k* arguments.

- ***Binary*** means 2-ary.
- ***Unary*** means 1-ary.
- ***Ternary*** means 3-ary.

When a *k*-ary operator appears in an expression, each of the *k* subexpressions whose values are the *k* arguments of the function it represents is called an ***operand*** of the operator.

Examples

In a Java or C++ expression x – y + z

- + is a binary operator whose operands are x – y and z.
- – is a binary operator whose operands are x and y.

Operators, Arities, and Operands

An ***operator of arity*** *k*, also called a *k*-***ary operator***, is a symbol that represents a function of *k* arguments.

- ***Binary*** means 2-ary.
- ***Unary*** means 1-ary.
- ***Ternary*** means 3-ary.

When a *k*-ary operator appears in an expression, each of the *k* subexpressions whose values are the *k* arguments of the function it represents is called an ***operand*** of the operator.

Examples

In a Java or C++ expression x – y + z

- + is a binary operator whose operands are x – y and z.
- – is a binary operator whose operands are x and y.

In the Lisp expression (+ (– x y) 3 (* z z))

- + is a ternary operator whose operands are (– x y), 3, and (* z z).

Overview of Some Expression Notations

We will say that two expressions are ***equivalent*** if they have the same semantics.

Overview of Some Expression Notations

We will say that two expressions are ***equivalent*** if they have the same semantics.

Here is a Java expression: `f(g(h(1,2), f(3,4)), 5)`
We will now write equivalent expressions in:

Overview of Some Expression Notations

We will say that two expressions are ***equivalent*** if they have the same semantics.

Here is a Java expression: `f(g(h(1,2), f(3,4)), 5)`
We will now write equivalent expressions in:

- **Infix notation, not making use of precedence classes**
- **Infix notation, making use of precedence classes**
- **Lisp notation**
- **Prefix notation**
- **Postfix notation** (also called **Reverse Polish Notation**)

Overview of Some Expression Notations

We will say that two expressions are ***equivalent*** if they have the same semantics.

Here is a Java expression: f(g(h(1,2), f(3,4)), 5)
We will now write equivalent expressions in:

- **Infix notation, not making use of precedence classes**
- **Infix notation, making use of precedence classes**
- **Lisp notation**
- **Prefix notation**
- **Postfix notation** (also called **Reverse Polish Notation**)

Infix Notation

In ***infix*** notation, we write binary operators ***between*** their operands--e.g., f(3,4) would be written 3 f 4.

So the above Java expression is equivalent to the following infix expression:

Overview of Some Expression Notations

We will say that two expressions are ***equivalent*** if they have the same semantics.

Here is a Java expression: `f(g(h(1,2), f(3,4)), 5)`
We will now write equivalent expressions in:

- **Infix notation, not making use of precedence classes**
- **Infix notation, making use of precedence classes**
- **Lisp notation**
- **Prefix notation**
- **Postfix notation** (also called **Reverse Polish Notation**)

Infix Notation

In ***infix*** notation, we write binary operators ***between*** their operands--e.g., `f(3,4)` would be written `3 f 4`.

So the above Java expression is equivalent to the following infix expression: `((1 h 2) g (3 f 4)) f 5`

The designer of an infix notation will usually give ***precedence and associativity rules*** for the operators, to reduce the need for parentheses.

Suppose the notation designer specifies that:

The designer of an infix notation will usually give ***precedence and associativity rules*** for the operators, to reduce the need for parentheses.

Suppose the notation designer specifies that:

- f and g belong to the ***same*** precedence class, and that precedence class is ***left-associative***.
- h belongs a ***higher*** precedence class than f and g.

Now the above infix expression ((1 h 2) g (3 f 4)) f 5 is equivalent to:

The designer of an infix notation will usually give ***precedence and associativity rules*** for the operators, to reduce the need for parentheses.

Suppose the notation designer specifies that:

- f and g belong to the ***same*** precedence class, and that precedence class is ***left-associative***.
- h belongs a ***higher*** precedence class than f and g.

Now the above infix expression ((1 h 2) g (3 f 4)) f 5
is equivalent to: 1 h 2 g (3 f 4) f 5

The designer of an infix notation will usually give ***precedence and associativity rules*** for the operators, to reduce the need for parentheses.

Suppose the notation designer specifies that:

- f and g belong to the ***same*** precedence class, and that precedence class is ***left-associative***.
- h belongs a ***higher*** precedence class than f and g.

Now the above infix expression ((1 h 2) g (3 f 4)) f 5
is equivalent to: 1 h 2 g (3 f 4) f 5

An Analogous Example

In ordinary infix arithmetic notation:

- + and - belong to the same precedence class, and that precedence class is left-associative.
- * belongs to a higher precedence class than + and –.

Thus the infix expression ((1 * 2) – (3 + 4)) + 5
is equivalent to:

The designer of an infix notation will usually give ***precedence and associativity rules*** for the operators, to reduce the need for parentheses.

Suppose the notation designer specifies that:

- f and g belong to the ***same*** precedence class, and that precedence class is ***left-associative***.
- h belongs a ***higher*** precedence class than f and g.

Now the above infix expression ((1 h 2) g (3 f 4)) f 5
is equivalent to: 1 h 2 g (3 f 4) f 5

An Analogous Example

In ordinary infix arithmetic notation:

- + and - belong to the same precedence class, and that precedence class is left-associative.
- * belongs to a higher precedence class than + and –.

Thus the infix expression ((1 * 2) – (3 + 4)) + 5
is equivalent to: 1 * 2 – (3 + 4) + 5

Lisp Notation and Prefix Notation

The above Java expression is equivalent to the following Lisp expression:

```
f(g(h(1,2), f(3,4)), 5)
```

Lisp Notation and Prefix Notation

The above Java expression is equivalent to the following Lisp expression:

```
f(g(h(1,2), f(3,4)), 5)

(f (g (h 1 2) (f 3 4)) 5)
```

Lisp Notation and Prefix Notation

The above Java expression `f(g(h(1,2), f(3,4)), 5)`
is equivalent to the
following Lisp expression: `(f (g (h 1 2) (f 3 4)) 5)`

Prefix notation is *Lisp notation* *without parentheses*.
Thus the above Java expression is equivalent to the following prefix expression:

Lisp Notation and Prefix Notation

The above Java expression f(g(h(1,2), f(3,4)), 5)
is equivalent to the
following Lisp expression: (f (g (h 1 2) (f 3 4)) 5)

Prefix notation is *Lisp notation* *without parentheses*.
Thus the above Java expression is equivalent to the
following prefix expression: f g h 1 2 f 3 4 5

Lisp Notation and Prefix Notation

The above Java expression `f(g(h(1,2), f(3,4)), 5)`
is equivalent to the
following Lisp expression: `(f (g (h 1 2) (f 3 4)) 5)`

Prefix notation is *Lisp notation* *without parentheses*.
Thus the above Java expression is equivalent to the
following prefix expression: `f g h 1 2 f 3 4 5`

Prefix notation is not ambiguous ***if we know the arity of every operator***: Then we can “put parentheses back” to produce an equivalent Lisp expression.

If we do not know the arities of operators, then prefix notation can be ambiguous.

For example,

Lisp Notation and Prefix Notation

The above Java expression f(g(h(1,2), f(3,4)), 5)
is equivalent to the
following Lisp expression: (f (g (h 1 2) (f 3 4)) 5)

Prefix notation is *Lisp notation without parentheses*.
Thus the above Java expression is equivalent to the following prefix expression: f g h 1 2 f 3 4 5

Prefix notation is not ambiguous ***if we know the arity of every operator***: Then we can “put parentheses back” to produce an equivalent Lisp expression.

If we do not know the arities of operators, then prefix notation can be ambiguous.

For example, if + and – are binary then
+ 1 – 2 3 is equivalent to

Lisp Notation and Prefix Notation

The above Java expression is equivalent to the following Lisp expression:

```
f(g(h(1,2), f(3,4)), 5)
(f (g (h 1 2) (f 3 4)) 5)
```

Prefix notation is *Lisp notation without parentheses*. Thus the above Java expression is equivalent to the following prefix expression: `f g h 1 2 f 3 4 5`

Prefix notation is not ambiguous ***if we know the arity of every operator***: Then we can "put parentheses back" to produce an equivalent Lisp expression.

If we do not know the arities of operators, then prefix notation can be ambiguous.

For example, if + and – are binary then `+ 1 – 2 3` is equivalent to `(+ 1 (– 2 3))`,

Lisp Notation and Prefix Notation

The above Java expression f(g(h(1,2), f(3,4)), 5)
is equivalent to the
following Lisp expression: (f (g (h 1 2) (f 3 4)) 5)

Prefix notation is *Lisp notation* *without parentheses*.
Thus the above Java expression is equivalent to the
following prefix expression: f g h 1 2 f 3 4 5

Prefix notation is not ambiguous ***if we know the arity of every operator***: Then we can “put parentheses back” to produce an equivalent Lisp expression.

If we do not know the arities of operators, then
prefix notation can be ambiguous.

For example, if + and – are binary then
+ 1 – 2 3 is equivalent to (+ 1 (– 2 3)),
but if + is ternary and – is unary then
+ 1 – 2 3 is equivalent to

Lisp Notation and Prefix Notation

The above Java expression f(g(h(1,2), f(3,4)), 5)
is equivalent to the
following Lisp expression: (f (g (h 1 2) (f 3 4)) 5)

Prefix notation is *Lisp notation* *without parentheses*.
Thus the above Java expression is equivalent to the following prefix expression: f g h 1 2 f 3 4 5

Prefix notation is not ambiguous ***if we know the arity of every operator***: Then we can “put parentheses back” to produce an equivalent Lisp expression.

If we do not know the arities of operators, then prefix notation can be ambiguous.

For example, if + and – are binary then
+ 1 – 2 3 is equivalent to (+ 1 (– 2 3)),
but if + is ternary and – is unary then
+ 1 – 2 3 is equivalent to (+ 1 (– 2) 3).

“RpnLisp” Notation and Postfix Notation

In Lisp, a function call is written as a list whose ***first*** element is the function name.

Now consider a notation we’ll call ***rpnLisp*** that’s the same as Lisp except in that a function call is written as a list whose ***last*** element is the function name.

“RpnLisp” Notation and Postfix Notation

In Lisp, a function call is written as a list whose ***first*** element is the function name.

Now consider a notation we’ll call ***rpnLisp*** that’s the same as Lisp except in that a function call is written as a list whose ***last*** element is the function name.

Thus the Lisp expression (+ (- 1 2) (* 3 4 5) 6)
is equivalent to the
following rpnLisp expression:

“RpnLisp” Notation and Postfix Notation

In Lisp, a function call is written as a list whose ***first*** element is the function name.

Now consider a notation we’ll call ***rpnLisp*** that’s the same as Lisp except in that a function call is written as a list whose ***last*** element is the function name.

Thus the Lisp expression (+ (- 1 2) (* 3 4 5) 6)
is equivalent to the
following rpnLisp expression: ((1 2 -) (3 4 5 *) 6 +)

“RpnLisp” Notation and Postfix Notation

In Lisp, a function call is written as a list whose ***first*** element is the function name.

Now consider a notation we’ll call ***rpnLisp*** that’s the same as Lisp except in that a function call is written as a list whose ***last*** element is the function name.

Thus the Lisp expression (+ (- 1 2) (* 3 4 5) 6)
is equivalent to the
following rpnLisp expression: ((1 2 -) (3 4 5 *) 6 +)

Just as ***prefix*** notation is “Lisp notation without parentheses”, ***postfix*** *notation is “rpnLisp notation without parentheses”*.

So the above expressions are equivalent to this ***postfix*** expression:

“RpnLisp” Notation and Postfix Notation

In Lisp, a function call is written as a list whose ***first*** element is the function name.

Now consider a notation we’ll call ***rpnLisp*** that’s the same as Lisp except in that a function call is written as a list whose ***last*** element is the function name.

Thus the Lisp expression (+ (- 1 2) (* 3 4 5) 6)
is equivalent to the
following rpnLisp expression: ((1 2 -) (3 4 5 *) 6 +)

Just as ***prefix*** notation is “Lisp notation without parentheses”, ***postfix*** *notation is “rpnLisp notation without parentheses”*.

So the above expressions are equivalent to this
postfix expression: 1 2 - 3 4 5 * 6 +

“RpnLisp” Notation and Postfix Notation

In Lisp, a function call is written as a list whose ***first*** element is the function name.

Now consider a notation we’ll call ***rpnLisp*** that’s the same as Lisp except in that a function call is written as a list whose ***last*** element is the function name.

Thus the Lisp expression (+ (- 1 2) (* 3 4 5) 6)
is equivalent to the
following rpnLisp expression: ((1 2 -) (3 4 5 *) 6 +)

Just as ***prefix*** notation is “Lisp notation without parentheses”, ***postfix*** *notation is “rpnLisp notation without parentheses”*.

So the above expressions are equivalent to this ***postfix*** expression: 1 2 - 3 4 5 * 6 +

In ***rpnLisp***, ***rpn*** stands for “**r**everse **p**olish **n**otation”.

```
The above Java expression     f(g(h(1,2), f(3,4)), 5)
is equivalent to the
Lisp expression               (f (g (h 1 2) (f 3 4)) 5)
and this rpnLisp expression:
```

```
The above Java expression    f(g(h(1,2), f(3,4)), 5)
is equivalent to the
Lisp expression              (f (g (h 1 2) (f 3 4)) 5)
and this rpnLisp expression: (((1 2 h) (3 4 f) g) 5 f)
```

The above Java expression   f(g(h(1,2), f(3,4)), 5)
is equivalent to the
Lisp expression   (f (g (h 1 2) (f 3 4)) 5)
and this rpnLisp expression: (((1 2 h) (3 4 f) g) 5 f)

Thus the expression is equivalent
to this ***postfix*** expression: 1 2 h 3 4 f g 5 f

```
The above Java expression     f(g(h(1,2), f(3,4)), 5)
is equivalent to the
Lisp expression               (f (g (h 1 2) (f 3 4)) 5)
and this rpnLisp expression: (((1 2 h) (3 4 f) g) 5 f)
```

Thus the expression is equivalent
to this ***postfix*** expression: `1 2 h 3 4 f g 5 f`

As is the case with prefix notation, postfix notation isn’t ambiguous ***if we know the arity of each operator***:

If we do not know the arities of operators, then postfix notation can be ambiguous.

For example:

The above Java expression   f(g(h(1,2), f(3,4)), 5)
is equivalent to the
Lisp expression   (f (g (h 1 2) (f 3 4)) 5)
and this rpnLisp expression: (((1 2 h) (3 4 f) g) 5 f)

Thus the expression is equivalent
to this ***postfix*** expression: 1 2 h 3 4 f g 5 f

As is the case with prefix notation, postfix notation isn't ambiguous ***if we know the arity of each operator***:

If we do not know the arities of operators, then postfix notation can be ambiguous.

For example:

- If + and – are binary, the postfix expression 1 2 3 + –
  is equivalent to the rpnLisp expression   (1 (2 3 +) –)

The above Java expression   f(g(h(1,2), f(3,4)), 5)
is equivalent to the
Lisp expression   (f (g (h 1 2) (f 3 4)) 5)
and this rpnLisp expression: (((1 2 h) (3 4 f) g) 5 f)

Thus the expression is equivalent
to this ***postfix*** expression: 1 2 h 3 4 f g 5 f

As is the case with prefix notation, postfix notation isn’t ambiguous ***if we know the arity of each operator***:

If we do not know the arities of operators, then postfix notation can be ambiguous.

For example:

- If + and – are binary, the postfix expression 1 2 3 + –
  is equivalent to the rpnLisp expression   (1 (2 3 +) –)
  and equivalent to the Lisp expression (– 1 (+ 2 3)) => –4.

The above Java expression `f(g(h(1,2), f(3,4)), 5)`
is equivalent to the
Lisp expression `(f (g (h 1 2) (f 3 4)) 5)`
and this rpnLisp expression: `(((1 2 h) (3 4 f) g) 5 f)`

Thus the expression is equivalent
to this ***postfix*** expression: `1 2 h 3 4 f g 5 f`

As is the case with prefix notation, postfix notation isn’t ambiguous ***if we know the arity of each operator***:

If we do not know the arities of operators, then postfix notation can be ambiguous.

For example:

- If + and – are binary, the postfix expression `1 2 3 + –`
  is equivalent to the rpnLisp expression `(1 (2 3 +) –)`
  and equivalent to the Lisp expression `(– 1 (+ 2 3))` => –4.
- If + is ternary and – is unary, the expression `1 2 3 + –`
  is equivalent to the rpnLisp expression `((1 2 3 +) –)`
  and equivalent to the Lisp expression

```
The above Java expression    f(g(h(1,2), f(3,4)), 5)
is equivalent to the
Lisp expression              (f (g (h 1 2) (f 3 4)) 5)
and this rpnLisp expression: (((1 2 h) (3 4 f) g) 5 f)
```

Thus the expression is equivalent
to this ***postfix*** expression: `1 2 h 3 4 f g 5 f`

As is the case with prefix notation, postfix notation isn’t ambiguous ***if we know the arity of each operator***:

If we do not know the arities of operators, then postfix notation can be ambiguous.

For example:

- If + and – are binary, the postfix expression `1 2 3 + –` is equivalent to the rpnLisp expression `(1 (2 3 +) –)` and equivalent to the Lisp expression `(– 1 (+ 2 3))` => –4.
- If + is ternary and – is unary, the expression `1 2 3 + –` is equivalent to the rpnLisp expression `((1 2 3 +) –)` and equivalent to the Lisp expression `(– (+ 1 2 3))` => –6.

More on Infix Notation

Infix notation allows unary and binary operators,
but does ***not*** allow operators of arity > 2.

Binary operators are written between their operands.

More on Infix Notation

Infix notation allows unary and binary operators, but does ***not*** allow operators of arity > 2.

Binary operators are written between their operands.

The designer of an infix notation must specify, for each *unary* operator the notation allows, whether that unary operator is to be written as a ***prefix operator*** or is to be written as a ***postfix operator***:

More on Infix Notation

Infix notation allows unary and binary operators, but does ***not*** allow operators of arity > 2.

Binary operators are written between their operands.

The designer of an infix notation must specify, for each *unary* operator the notation allows, whether that unary operator is to be written as a ***prefix operator*** or is to be written as a ***postfix operator***:

- Prefix operators are written *before* their operands.
  - **Examples**:
- Postfix operators are written *after* their operands.
  - **Example**:

More on Infix Notation

Infix notation allows unary and binary operators, but does ***not*** allow operators of arity > 2.

Binary operators are written between their operands.

The designer of an infix notation must specify, for each *unary* operator the notation allows, whether that unary operator is to be written as a ***prefix operator*** or is to be written as a ***postfix operator***:

- Prefix operators are written *before* their operands.
  - **Examples**: `-` in a Java or C++ expression `-x`
    `++` in a Java or C++ expression `++i`
    `*` in a C++ expression `*ptr`
- Postfix operators are written *after* their operands.
  - **Example**:

More on Infix Notation

Infix notation allows unary and binary operators, but does ***not*** allow operators of arity > 2.

Binary operators are written between their operands.

The designer of an infix notation must specify, for each *unary* operator the notation allows, whether that unary operator is to be written as a ***prefix operator*** or is to be written as a ***postfix operator***:

- Prefix operators are written *before* their operands.
  - **Examples**: `-` in a Java or C++ expression `-x`
    `++` in a Java or C++ expression `++i`
    `*` in a C++ expression `*ptr`
- Postfix operators are written *after* their operands.
  - **Example**: `++` in a Java or C++ expression `i++`

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (*s.v.i.e.*) if one of the following is true:

1. e is a literal constant or an identifier.

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

1. e is a literal constant or an identifier.
2. e = (e_1), where e_1 is an s.v.i.e.

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

1. e is a literal constant or an identifier.
2. e = (e_1), where e_1 is an s.v.i.e.
3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

1. e is a literal constant or an identifier.
2. e = (e_1), where e_1 is an s.v.i.e.
3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.
4. e = **op** e_1 where e_1 is an s.v.i.e. and **op** is a prefix unary operator.

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

1. e is a literal constant or an identifier.
2. e = (e_1), where e_1 is an s.v.i.e.
3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.
4. e = **op** e_1 where e_1 is an s.v.i.e. and **op** is a prefix unary operator.
5. e = e_1 **op** where e_1 is an s.v.i.e. and **op** is a postfix unary operator.

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

1. e is a literal constant or an identifier.
2. e = (e_1), where e_1 is an s.v.i.e.
3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.
4. e = **op** e_1 where e_1 is an s.v.i.e. and **op** is a prefix unary operator.
5. e = e_1 **op** where e_1 is an s.v.i.e. and **op** is a postfix unary operator.

Rules 2 – 5 give decompositions of e into two (rules 4 & 5) or three (rule 2 & 3) substructures, but

Syntactically Valid Infix Expressions

An expression e is a ***syntactically valid infix expression*** (***s.v.i.e.***) if one of the following is true:

1. e is a literal constant or an identifier.
2. e = (e_1), where e_1 is an s.v.i.e.
3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.
4. e = **op** e_1 where e_1 is an s.v.i.e. and **op** is a prefix unary operator.
5. e = e_1 **op** where e_1 is an s.v.i.e. and **op** is a postfix unary operator.

Rules 2 – 5 give decompositions of e into two (rules 4 & 5) or three (rule 2 & 3) substructures, but some of these decompositions may ***violate*** the following important principle of syntax specification:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Let e be the following Java expression: x – y * z + w

Then   3. e = e_1 **op** e_2 where each of e_1 and e_2 is an
        s.v.i.e. and **op** is a binary operator.

gives the following decompositions of e:

Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Let e be the following Java expression: x – y * z + w

Then 3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.

gives the following decompositions of e:

(i) e_1 = x **op** = – e_2 = y * z + w

Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Let e be the following Java expression: x – y * z + w

Then 3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.

gives the following decompositions of e:

(i)	e_1 = x	**op** = –	e_2 = y * z + w
(ii)	e_1 = x – y	**op** = *	e_2 = z + w

Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Let e be the following Java expression: x – y * z + w

Then 3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.

gives the following decompositions of e:

(i)	e_1 = x	**op** = –	e_2 = y * z + w
(ii)	e_1 = x – y	**op** = *	e_2 = z + w
(iii)	e_1 = x – y * z	**op** = +	e_2 = w

Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Let e be the following Java expression: x – y * z + w

Then 3. e = e_1 **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.

gives the following decompositions of e:

(i)	e_1 = x	**op** = –	e_2 = y * z + w
(ii)	e_1 = x – y	**op** = *	e_2 = z + w
(iii)	e_1 = x – y * z	**op** = +	e_2 = w

But decompositions (i) and (ii) both ***violate*** the above principle, because (based on the semantics of Java), e is equivalent to: (x – y * z) + w

Example of How the Principle May be Violated

Recall that the principle is:

- ***The semantics of a structure should be easily definable in terms of the semantics of its syntactic substructures.***

Let e be the following Java expression: x – y * z + w

Then 3. $e = e_1$ **op** e_2 where each of e_1 and e_2 is an s.v.i.e. and **op** is a binary operator.

gives the following decompositions of e:

(i)	e_1 = x	**op** = –	e_2 = y * z + w
(ii)	e_1 = x – y	**op** = *	e_2 = z + w
(iii)	e_1 = x – y * z	**op** = +	e_2 = w

But decompositions (i) and (ii) both ***violate*** the above principle, because (based on the semantics of Java), e is equivalent to: (x – y * z) + w

- Sec. 2.5 of Sethi (assigned reading after Exam 1) gives another way to specify syntactically valid infix expressions ***that does not have this drawback***.

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Let e.value denote the value of e. Then:

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant,
   e.value = the value of the identifier / constant.

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant,
   e.value = the value of the identifier / constant.
2. If e = (e_1),
   e.value = e_1.value.

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant,
   e.value = the value of the identifier / constant.
2. If e = (e_1),
   e.value = e_1.value.

Otherwise, ***let*** **op** ***be the operator of*** e ***that should be applied last***. Then:

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant,
   e.value = the value of the identifier / constant.
2. If e = (e_1),
   e.value = e_1.value.

Otherwise, ***let* op *be the operator of* e *that should be applied last***. Then:

- If e is e_1 **op** e_2,
  e.value = result of applying **op** with e_1.value and
  e_2.value as the 1st and 2nd arguments.

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant,
   e.value = the value of the identifier / constant.
2. If e = (e_1),
   e.value = e_1.value.

Otherwise, *let* **op** *be the operator of* e *that should be applied last*. Then:

- If e is e_1 **op** e_2,
  e.value = result of applying **op** with e_1.value and
  e_2.value as the 1st and 2nd arguments.
- If e is **op** e_1 or e is e_1 **op**
  e.value = result of applying op to e_1.value.

Semantics of an Infix Expression e

The semantics of e tells you how e can be evaluated.

Let e.value denote the value of e. Then:

1. If e is an identifier or a literal constant,
   e.value = the value of the identifier / constant.
2. If e = (e_1),
   e.value = e_1.value.

Otherwise, ***let* op *be the operator of* e *that should be applied last*.** Then:

- If e is e_1 **op** e_2,
  e.value = result of applying **op** with e_1.value and e_2.value as the 1st and 2nd arguments.
- If e is **op** e_1 or e is e_1 **op**
  e.value = result of applying op to e_1.value.

Key Question: How can we determine which operator of e should be applied **last**?

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **and**

(b)

and

(c)

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e,   **and**
(b) **op** is ***not*** a prefix unary operator that
    is immediately after another operator in e, **and**
(c) **op** is ***not*** a postfix unary operator that
    is immediately before another operator in e.

Example:

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e, **and**

(b) **op** is ***not*** a prefix unary operator that is immediately after another operator in e, **and**

(c) **op** is ***not*** a postfix unary operator that is immediately before another operator in e.

Example: The top-level operators in the C++ expression

```
– x / – (y + (z – 3) – 2) * w ++ % v ++
```

are

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if
(a) **op** is ***not*** surrounded by parentheses in e, **and**
(b) **op** is ***not*** a prefix unary operator that
is immediately after another operator in e, **and**
(c) **op** is ***not*** a postfix unary operator that
is immediately before another operator in e.

Example: The top-level operators in the C++ expression

- x / - (y + (z - 3) - 2) * w ++ % v ++

are the ***1st*** **unary -**,

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if
(a) **op** is ***not*** surrounded by parentheses in e, **and**
(b) **op** is ***not*** a prefix unary operator that
is immediately after another operator in e, **and**
(c) **op** is ***not*** a postfix unary operator that
is immediately before another operator in e.

Example: The top-level operators in the C++ expression

```
- x / - (y + (z - 3) - 2) * w ++ % v ++
```

are the ***1st* unary -**, **/**,

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e,   **and**

(b) **op** is ***not*** a prefix unary operator that
    is immediately after another operator in e, **and**

(c) **op** is ***not*** a postfix unary operator that
    is immediately before another operator in e.

Example: The top-level operators in the C++ expression

- x / - (y + (z - 3) - 2) * w ++ % v ++

are the ***1st* unary -**, **/**, *****,

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if
(a) **op** is ***not*** surrounded by parentheses in e,   **and**
(b) **op** is ***not*** a prefix unary operator that
    is immediately after another operator in e, **and**
(c) **op** is ***not*** a postfix unary operator that
    is immediately before another operator in e.

Example: The top-level operators in the C++ expression
- x / - (y + (z - 3) - 2) * w ++ % v ++
are the ***1st unary*** **-**, **/**, *****, **%**,

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e, **and**

(b) **op** is ***not*** a prefix unary operator that
    is immediately after another operator in e, **and**

(c) **op** is ***not*** a postfix unary operator that
    is immediately before another operator in e.

Example: The top-level operators in the C++ expression

- x / - (y + (z - 3) - 2) * w ++ % v ++

are the ***1st* unary -**, **/**, *****, **%**, and the ***2nd* ++**.

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e, **and**

(b) **op** is ***not*** a prefix unary operator that
is immediately after another operator in e, **and**

(c) **op** is ***not*** a postfix unary operator that
is immediately before another operator in e.

Example: The top-level operators in the C++ expression

– x / – (y + (z – 3) – 2) * w ++ % v ++

are the *1st* unary –, /, *, %, and the *2nd* ++.

The three operators in (y + (z – 3) – 2) are
not top-level: They violate (a).

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e, **and**

(b) **op** is ***not*** a prefix unary operator that is immediately after another operator in e, **and**

(c) **op** is ***not*** a postfix unary operator that is immediately before another operator in e.

Example: The top-level operators in the C++ expression

```
- x / - (y + (z - 3) - 2) * w ++ % v ++
```

are the *1st* unary -, /, *, %, and the *2nd* ++.

The three operators in (y + (z - 3) - 2) are ***not*** top-level: They violate (a).

The ***2nd* unary -** is ***not*** top-level: It violates (b).

How to Determine Which Operator of an Infix Expression e Should be Applied Last

We'll say an operator **op** is ***top-level*** in e if

(a) **op** is ***not*** surrounded by parentheses in e, **and**

(b) **op** is ***not*** a prefix unary operator that
    is immediately after another operator in e, **and**

(c) **op** is ***not*** a postfix unary operator that
    is immediately before another operator in e.

Example: The top-level operators in the C++ expression

```
– x / – (y + (z – 3) – 2) * w ++ % v ++
```

are the *1st* unary –, /, *, %, and the *2nd* ++.

The three operators in (y + (z – 3) – 2) are ***not*** top-level: They violate (a).

The 2nd unary – is ***not*** top-level: It violates (b).

The *1st* ++ is ***not*** top-level: It violates (c).

How to Determine Which Operator of an Infix Expression e Should be Applied Last (Continued)

How to Determine Which Operator of an Infix Expression e Should be Applied Last (Continued)

If no precedence and associativity rules for the operators have been given, then when an expression has more than one top level operator it will ***not*** be possible to uniquely determine which operator should be applied last!

How to Determine Which Operator of an Infix Expression e Should be Applied Last (Continued)

If no precedence and associativity rules for the operators have been given, then when an expression has more than one top level operator it will ***not*** be possible to uniquely determine which operator should be applied last!

However, the designer of an infix notation will usually give ***precedence and associativity rules*** for the operators that:

(i)

(ii)

How to Determine Which Operator of an Infix Expression e Should be Applied Last (Continued)

If no precedence and associativity rules for the operators have been given, then when an expression has more than one top level operator it will ***not*** be possible to uniquely determine which operator should be applied last!

However, the designer of an infix notation will usually give ***precedence and associativity rules*** for the operators that:

(i) partition the operators into ***ranked precedence classes***, and

(ii)

How to Determine Which Operator of an Infix Expression e Should be Applied Last (Continued)

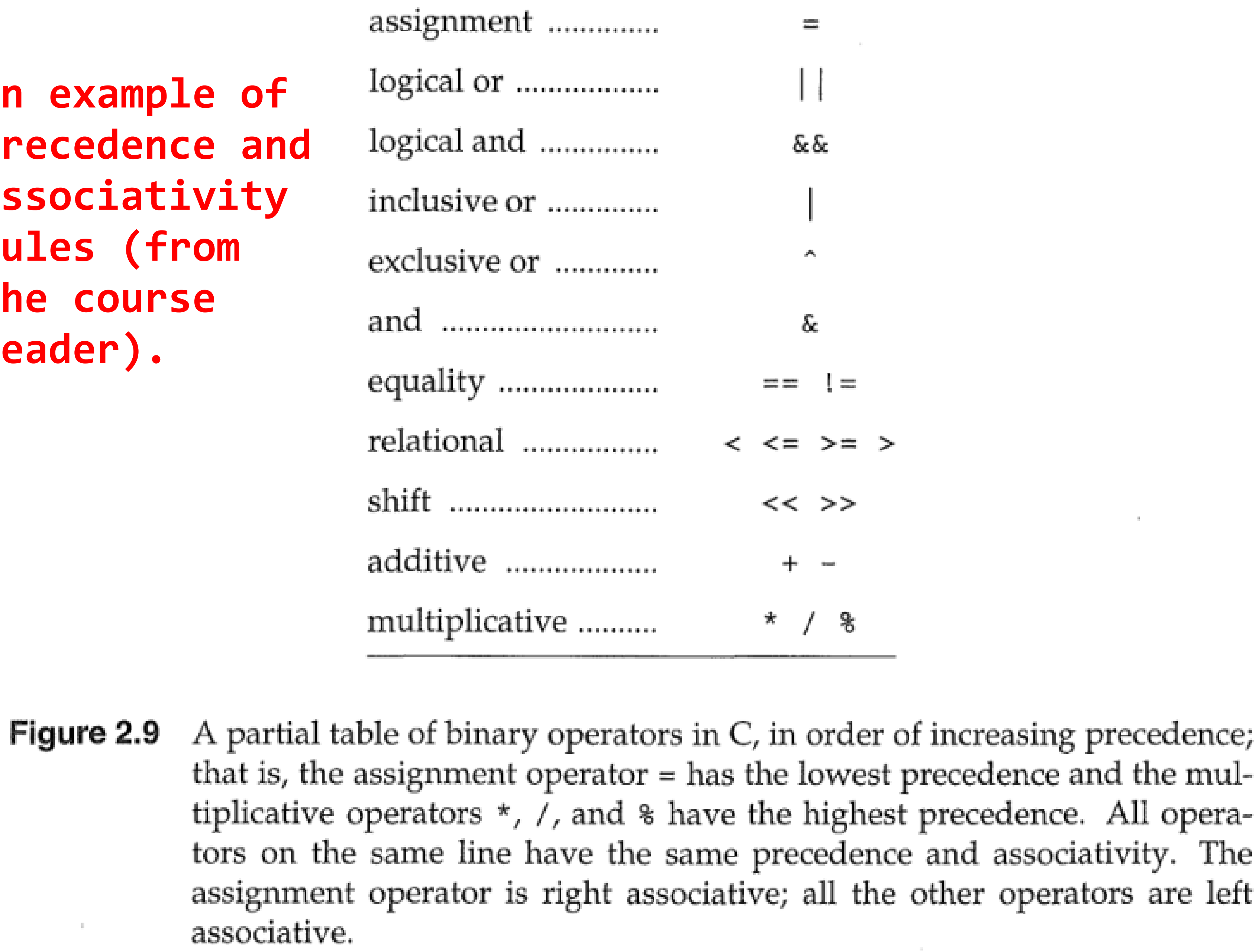
If no precedence and associativity rules for the operators have been given, then when an expression has more than one top level operator it will ***not*** be possible to uniquely determine which operator should be applied last!

However, the designer of an infix notation will usually give ***precedence and associativity rules*** for the operators that:

(i) partition the operators into ***ranked precedence classes,*** and

(ii) specify, for each class, whether the class is ***left***-associative (= ***left-to-right*** associative) or ***right***-associative (= ***right-to-left*** associative).

An example of precedence and associativity rules (from the course reader).



assignment	=
logical or	||
logical and	&&
inclusive or	|
exclusive or	^
and	&
equality	== !=
relational	< <= >= >
shift	<< >>
additive	+ -
multiplicative	* / %

Figure 2.9 A partial table of binary operators in C, in order of increasing precedence; that is, the assignment operator = has the lowest precedence and the multiplicative operators *, /, and % have the highest precedence. All operators on the same line have the same precedence and associativity. The assignment operator is right associative; all the other operators are left associative.

How to Determine Which Operator of an Infix Expression e Should be Applied Last (Continued)

If no precedence and associativity rules for the operators have been given, then when an expression has more than one top level operator it will ***not*** be possible to uniquely determine which operator should be applied last!

However, the designer of an infix notation will usually give ***precedence and associativity rules*** for the operators that:

(i) partition the operators into ***ranked precedence classes***, and

(ii) specify, for each class, whether the class is ***left***-associative (= ***left-to-right*** associative) or ***right***-associative (= ***right-to-left*** associative).

How to Determine Which Operator of an Infix Expression e Should be Applied Last (Continued)

If no precedence and associativity rules for the operators have been given, then when an expression has more than one top level operator it will ***not*** be possible to uniquely determine which operator should be applied last!

However, the designer of an infix notation will usually give ***precedence and associativity rules*** for the operators that:

(i) partition the operators into ***ranked precedence classes***, and

(ii) specify, for each class, whether the class is ***left***-associative (= ***left-to-right*** associative) or ***right***-associative (= ***right-to-left*** associative).

Using these rules, ***we can find the operator of*** e ***that should be applied last as follows***:

Follow These Steps to Determine Which Operator of an Infix Expression e Should be Applied Last:

1.

2.

2.1

2.2

2.3

Follow These Steps to Determine Which Operator of an Infix Expression e Should be Applied Last:

1. If e is of the form (e′), then the operator that should be applied last in e′ is also the operator that should be applied last in e.

2.

2.1

2.2

2.3

Follow These Steps to Determine Which Operator of an Infix Expression e Should be Applied Last:

1. If e is of the form (e′), then the operator that should be applied last in e′ is also the operator that should be applied last in e.

2. Otherwise, the operator that should be applied last in e can be found by doing 2.1 – 2.3 below.

2.1

2.2

2.3

Follow These Steps to Determine Which Operator of an Infix Expression e Should be Applied Last:

1. If e is of the form (e′), then the operator that should be applied last in e′ is also the operator that should be applied last in e.
2. Otherwise, the operator that should be applied last in e can be found by doing 2.1 – 2.3 below.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2

2.3

Follow These Steps to Determine Which Operator of an Infix Expression e Should be Applied Last:

1. If e is of the form (e′), then the operator that should be applied last in e′ is also the operator that should be applied last in e.
2. Otherwise, the operator that should be applied last in e can be found by doing 2.1 – 2.3 below.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3

Follow These Steps to Determine Which Operator of an Infix Expression e Should be Applied Last:

1. If e is of the form (e′), then the operator that should be applied last in e′ is also the operator that should be applied last in e.

2. Otherwise, the operator that should be applied last in e can be found by doing 2.1 – 2.3 below.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but***

Follow These Steps to Determine Which Operator of an Infix Expression e Should be Applied Last:

1. If e is of the form (e′), then the operator that should be applied last in e′ is also the operator that should be applied last in e.

2. Otherwise, the operator that should be applied last in e can be found by doing 2.1 – 2.3 below.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

Example: Find the operator that should be applied last in this C expression: x * (y + (z + 3) – 2) + w – u / t

Solution:

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

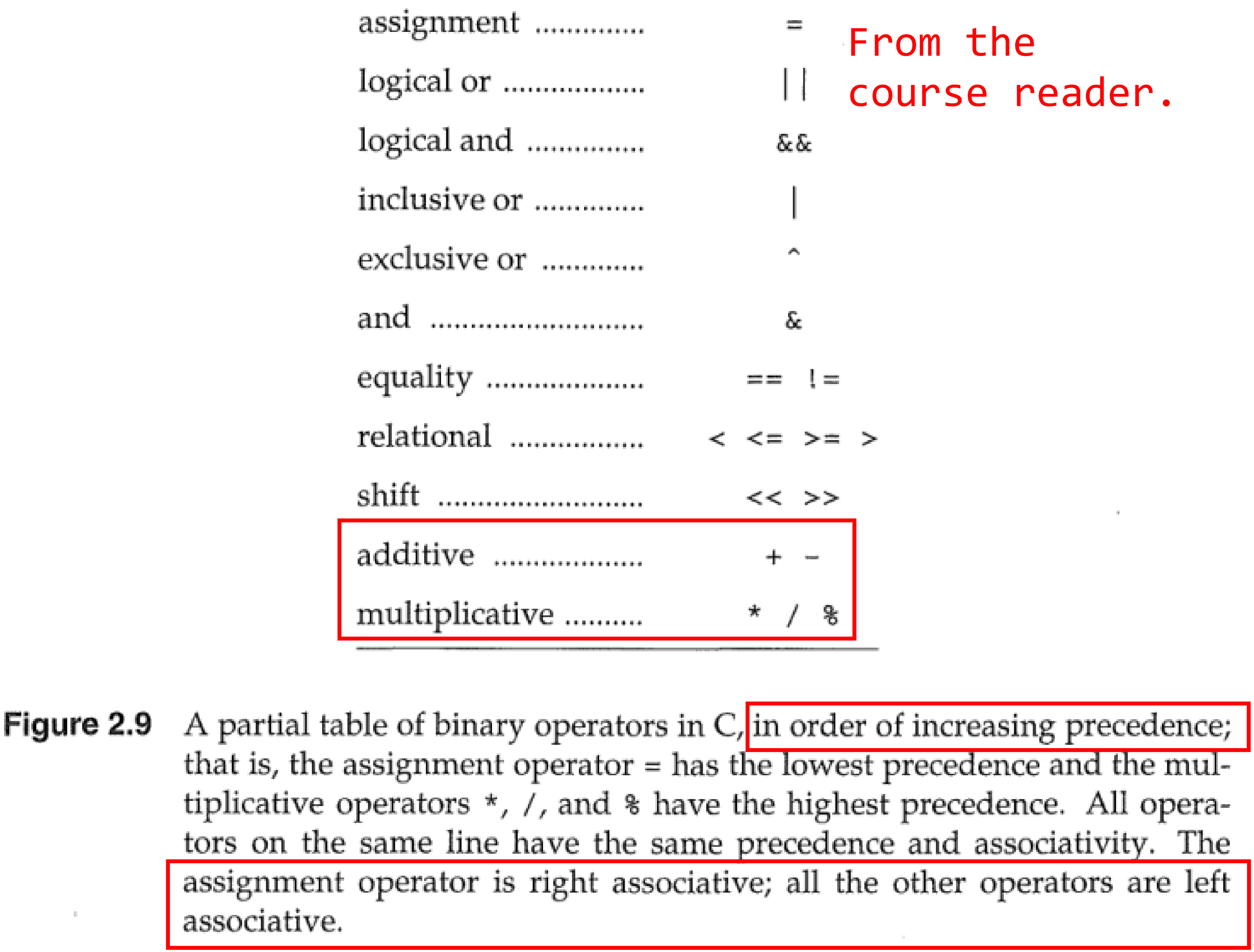
2.3 If more than one operator is found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

Example: Find the operator that should be applied last in this C expression: x * (y + (z + 3) – 2) + w – u / t

Solution: There are 4 top-level operators, namely the 4 **black** operators in: x ***** (y + (z + 3) – 2) **+** w **–** u **/** t

Step 2.1:

Step 2.3:



assignment	=
logical or	||
logical and	&&
inclusive or	|
exclusive or	^
and	&
equality	== !=
relational	< <= >= >
shift	<< >>
additive	+ -
multiplicative	* / %

Figure 2.9 A partial table of binary operators in C, in order of increasing precedence; that is, the assignment operator = has the lowest precedence and the multiplicative operators *, /, and % have the highest precedence. All operators on the same line have the same precedence and associativity. The assignment operator is right associative; all the other operators are left associative.

From the course reader.

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If two or more operators are found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

Example: Find the operator that should be applied last in this C expression: x * (y + (z + 3) – 2) + w – u / t

Solution: There are 4 top-level operators, namely the 4 **black** operators in: x ***** (y + (z + 3) – 2) **+** w **–** u **/** t

Step 2.1:

Step 2.3:

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If two or more operators are found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

Example: Find the operator that should be applied last in this C expression: x * (y + (z + 3) - 2) + w - u / t

Solution: There are 4 top-level operators, namely the 4 **black** operators in: x ***** (y + (z + 3) - 2) **+** w **-** u **/** t

Step 2.1: The top-level operators of ***lowest precedence*** are the 2 blue operators in: x * (y + (z + 3) - 2) + w - u / t

Step 2.3:

2.1 Find the ***top-level*** operators of ***lowest*** precedence rank in e.

2.2 If just one operator is found by step 2.1, then that is the operator that should be applied last.

2.3 If two or more operators are found by step 2.1, then the operator that should be applied last is the ***rightmost*** of the operators found by 2.1 if their precedence class is ***left***-associative, ***but*** is the ***leftmost*** of the operators found by 2.1 if their precedence class is ***right***-associative.

Example: Find the operator that should be applied last in this C expression: x * (y + (z + 3) - 2) + w - u / t

Solution: There are 4 top-level operators, namely the 4 **black** operators in: x ***** (y + (z + 3) - 2) **+** w **-** u **/** t

Step 2.1: The top-level operators of ***lowest precedence*** are the 2 blue operators in: x * (y + (z + 3) - 2) **+** w **-** u / t

Step 2.3: These 2 operators **+** and **-** belong to a ***left-associative*** precedence class, so the rightmost of them (i.e., **-**) must be applied last.

Precedence & Associativity Rules Might ***Not*** Uniquely Determine Which Operator Should be Applied ***First***

Precedence & Associativity Rules Might ***Not*** Uniquely Determine Which Operator Should be Applied ***First***

Consider this C/C++ expression: **y / 2 * --y**

/ and ***** belong to a ***left-associative*** precedence class, so the operator that should be applied ***last*** is: *****

Precedence & Associativity Rules Might *Not* Uniquely Determine Which Operator Should be Applied *First*

Consider this C/C++ expression: **y / 2 * --y**

/ and ***** belong to a ***left-associative*** precedence class, so the operator that should be applied ***last*** is: *****

But a C or C++ compiler is free to generate code that applies **--** first or applies **/** first!

Precedence & Associativity Rules Might ***Not*** Uniquely Determine Which Operator Should be Applied ***First***

Consider this C/C++ expression: **y / 2 * --y**

/ and ***** belong to a ***left-associative*** precedence class, so the operator that should be applied ***last*** is: *****

But a C or C++ compiler is free to generate code that applies **--** first or applies **/** first!

Note: In addition to precedence & associativity rules, a language may have *other* rules that govern the order in which operators in infix expressions are applied!

Example:

Precedence & Associativity Rules Might ***Not*** Uniquely Determine Which Operator Should be Applied ***First***

Consider this C/C++ expression: **y / 2 * --y**

/ and ***** belong to a ***left-associative*** precedence class, so the operator that should be applied ***last*** is: *****

But a C or C++ compiler is free to generate code that applies **--** first or applies **/** first!

Note: In addition to precedence & associativity rules, a language may have *other* rules that govern the order in which operators in infix expressions are applied!

Example: In **Java**, arguments of functions or operators are evaluated in ***left-to-right*** order, so **/** is applied ***before*** **--** when evaluating the above expression. Thus

Precedence & Associativity Rules Might *Not* Uniquely Determine Which Operator Should be Applied *First*

Consider this C/C++ expression: `y / 2 * --y`

`/` and `*` belong to a ***left-associative*** precedence class, so the operator that should be applied ***last*** is: `*`

But a C or C++ compiler is free to generate code that applies `--` first or applies `/` first!

Note: In addition to precedence & associativity rules, a language may have *other* rules that govern the order in which operators in infix expressions are applied!

Example: In **Java**, arguments of functions or operators are evaluated in ***left-to-right*** order, so `/` is applied ***before*** `--` when evaluating the above expression. Thus

```
     int y = 4; System.out.println(y / 2 * --y);
```

prints 6 (not 3).

Precedence & Associativity Rules Might *Not* Uniquely Determine Which Operator Should be Applied *First*

Consider this C/C++ expression: **y / 2 * --y**

/ and ***** belong to a ***left-associative*** precedence class, so the operator that should be applied ***last*** is: *****

But a C or C++ compiler is free to generate code that applies **--** first or applies **/** first!

Note: In addition to precedence & associativity rules, a language may have *other* rules that govern the order in which operators in infix expressions are applied!

Example: In **Java**, arguments of functions or operators are evaluated in ***left-to-right*** order, so **/** is applied ***before*** **--** when evaluating the above expression. Thus

```
     int y = 4; System.out.println(y / 2 * --y);
```

prints 6 (not 3). In C++ there's no left-to-right rule, so **int y = 4; cout << y / 2 * --y;** may print 3 or 6.