Programming Languages: Expressions

Ryan Stansifer

Florida Institute of Technology
Melbourne, Florida USA 32901

http://www.cs.fit.edu/~ryan/

14 May 2023

Expressions

Overview of Topics on Expressions

- Literals
- Constants
- Operators and their operations
- Order of evaluation
  precedence, parentheses, associativity, short-circuit
- Rewriting
- Referential transparency
  - Scott, 4th ed., Section 6.2.1, page 230
  - Mitchell, Section 4.4, page 78.
  - Sebesta, 9th ed., Section 15.4, page 660. Sebesta, 11th ed., Section 7.2, page 310.

A cursory discussion of data values occurs in the discussion of type following this part.

Definition of a Literal

Definition

A *literal* is a lexical element of the program language syntax (not an identifier) standing for a specific, unchanging value.
As such the value is known at compile time.

Various Literals

`365`	typical integer
`5.11E-8`	real number in “scientific” notation
`170_234`	integer in Ada
`TRUE`	boolean value in Modula-3
`'A'`	typical character
`$a`	character in Smalltalk
`#symbol`	symbol or atom in Smalltalk
`'atom`	atom in LISP
`"string"`	typical string
`5HHELLO`	old “Hollerith” string in FORTRAN
`[]`	empty list in PROLOG and ML
`()`	`unit` in ML

String Syntax

Strings are, most commonly, sequences of characters delimited by the US-ASCII double quote character.
However, Python allows short strings (not containing newlines) and long strings. Long strings begin and end with *three* quote characters. Furthermore, either the USA-ASCII double-quote or single-quote character and be used.

Raw or Verbatim Strings

A raw string, sometimes call a verbatim string especially in .Net and C# community, has fewer meta character sequences that stand for other characters and more characters stand just for themselves.

Leaning Toothpick Syndrome

We give three examples.

- C#
- Python
- Java a raw string syntax did not make Java 12, but is expected in Java 13
- Scala

Verbatim or Raw Strings: C++

A simple example of a "raw" string in C++.

`savitch/raw.cpp`

Verbatim or Raw Strings

C#: Verbatim string literals `@"..."` (no escape sequences except the quote escape sequence—two quote marks). See Special Verbatim Character.
Python: short string literal, and long string literals (triple quoted strings). A string literal with 'f' or 'F' in its prefix is a formatted string literal; see Formatted string literals. The 'f' may be combined with 'r'.
Python raw string literal. Unless an 'r' or 'R' prefix is present, escape sequences in strings are interpreted according to rules similar to those used by Standard C.

```
r"""\d +  # the integral part
                  \.    # the decimal point
                  \d *  # some fractional digits"""
```

Verbatim or Raw Strings

In the context of strings intended to be regular expressions, Java has a unique "quote-unquote" escape sections. This should not be confused with raw strings. This Java string

```
"\n\\Q\n \Q\\\\E\t"
```

matches (but is not the same as) this:

binary	oct	dec	Latin1	Unicode	
0000 1010	0012	10	LF	U+000A	line feed
0000 1010	0012	10	LF	U+000A	line feed
0101 1100	0134	92	\	U+005C	reverse solidus
0010 0000	0040	32		U+0020	space
0101 1100	0134	92	\	U+005C	reverse solidu
0101 0001	0121	81	Q	U+0051	latin capital letter Q
0000 1001	0011	9	HT	U+0009	horizontal tabulation

Raw or Verbatim Strings

`strings/Main.kt` ↗

Verbatim or Raw Strings

Scala

```
raw"a\nb"
s"Hello, $name"
f"$name%s is $height%2.2f meters tall"
```

C++

```
R"(a\nb)"
```

String Interpolation

Python (> 3.6)

```
sys.stdout.write (f"Area is {abs((x-z)*(y-w)):.2f} square cm.\n")
```

Scala

```
println (s"Area is ${abs((x-z)*(y-w))}%.2f square cm.\n")
```

C#

```
Console.WriteLine($"Area is ${abs((x-z)*(y-w)):.2f} square cm.\n")
```

String Interpolation

```
int fib(int n) => (n>2) ? (fib(n-1) + fib(n-2)) : 1;

void main() {
  print('fib(20) = ${fib(20)}');
}
```

Constants

A *constant* is an identifier whose r-value does not change at run time. If the value of a constant can be determined at compile time, it is said to be a *static constant*, sometimes called a *compile-time constant* or *manifest constant*. Static constants are useful to the compiler for constant folding and other optimizations. An example in Ada:

```
Limit     : constant Integer := 10_000;
Low_Limit : constant Integer := Limit/10;
Tolerance : constant Float   := Dispersion(1.15);
```

Different Kinds of Constants

In C# you have both compile-time and run-time constants:

```
public const int _Year = 2008;

public static readonly DateTime _Now = new DateTime ();
```

Single Assignment Style

Single assignment style: each identifier is assigned a value only once.
An example in Java:

```
final int golden  = (year % 19) + 1;
final int century = (year / 100) + 1;
int epact = (11*golden+20+clavian-gregorian) % 30;
if (epact==24 || (epact==25&&golden==11)) epact++;
final int sunday = moon + 7 - ((extra+moon) % 7);
final String month = (sunday>31) ? "April": "March";
```

Algebra

Broadly speaking *algebra* is the study of mathematical symbols and the rules for using these symbols.

An *expression* is a syntactically meaningful notation written in linear form and defined recursively by applying symbols for operators to zero or more other expressions as operands.

Nullary operators themselves are expressions and are often called constants. It is often the case that a collection of distinct symbols used as variables are allowed as operands.

In in abstract algebra, an *algebraic structure* is a collection of finitary operations; a set with this structure is also called an algebra.

Sane people will organize an algebraic structure around one or more clearly defined domains of values with operators of fixed arities (often 2) and types.

Expressions

Expression. An *expression* is a construct representing one (integer, boolean, real, etc.) value.
Examples in Ada:

`1_234`	`Integer`
`23 + 56`	`Integer`
`34 mod 2`	`Integer`
`4*X + 92`	`Integer`
`Final='A' or else Mid='A'`	`Boolean`
`X mod 2 = 0`	`Boolean`
`2*X**3 + Y**2`	`Integer`
`X / 34.0`	`Float`
`3*(X-5)`	`Integer`
`(((A-1)*(N-1))/4) mod 2 = 0`	`Boolean`

The Typical Syntax Of Expressions

```
<expression> ::= <literal>
<expression> ::= <identifier>
<expression> ::= "(" <expression> ")"
<expression> ::= <expression> <operator> <expression>
```

This grammar has some defects, but captures the essence of expressions in most programming languages. Notice that knowing the operators in a programming language tells one pretty much all there is to know.

Ada Operators

`or`	1	disjunction
`xor`	1	exclusive or
`or else`	1	conditional or
`and`	1	conjunction
`and then`	1	conditional and
`=`	2	equality
`/=`	2	inequality
`<`	2	less than
`<=`	2	less or equal
`>`	2	greater than
`>=`	2	greater or equal
`in`	2	membership
`not in`	2	not member
`+`	3	addition
`-`	3	subtraction
`&`	3	concatenation
`*`	4	multiplication
`/`	4	division
`mod`	4	modulus
`rem`	4	remainder
`**`	5	exponentiation
`abs`	5	absolute value
`not`	5	negation

`**` is *non* associative; all others are left associative

Modula-3 Operators

`OR`	1	conditional or
`AND`	2	conditional and
`NOT`	3	(unary) negation
`=`	4	equality
`#`	4	inequality
`<`	4	less than
`<=`	4	less or equal
`>`	4	greater than
`>=`	4	greater or equal
`IN`	4	membership
`+`	5	addition
`-`	5	subtraction
`&`	5	concatenation
`*`	6	multiplication
`/`	6	real division
`DIV`	6	integer division
`MOD`	6	modulus

all infix operators are left associative

C, C++, Java, C# Operators

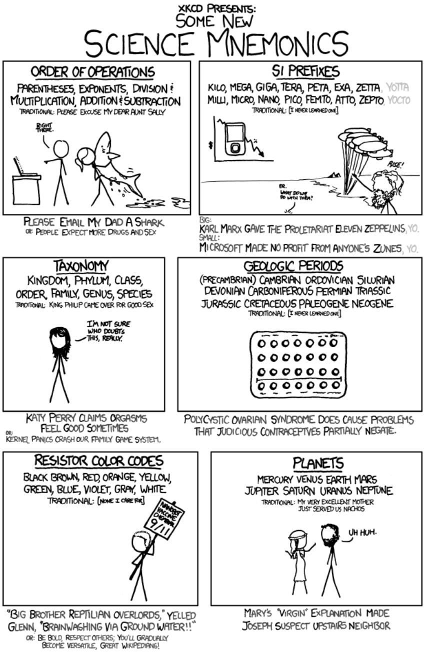
<table>
<tr><td>||</td><td>1</td><td>conditional or</td></tr>
<tr><td>&&</td><td>2</td><td>conditional and</td></tr>
<tr><td>|</td><td>3</td><td>inclusive or</td></tr>
<tr><td>^</td><td>4</td><td>exclusive or</td></tr>
<tr><td>&</td><td>5</td><td>and</td></tr>
<tr><td>==, !=</td><td>6</td><td>equality and inequality</td></tr>
<tr><td><, <=, >, >=</td><td>7</td><td>relational</td></tr>
<tr><td>instanceof</td><td>7</td><td>instance of class (Java)</td></tr>
<tr><td><<, >>, >>></td><td>8</td><td>bit shift</td></tr>
<tr><td>+</td><td>9</td><td>addition (and concatenation in Java)</td></tr>
<tr><td>-</td><td>9</td><td>subtraction</td></tr>
<tr><td>*</td><td>10</td><td>multiplication</td></tr>
<tr><td>/</td><td>10</td><td>integer/real division</td></tr>
<tr><td>%</td><td>10</td><td>remainder</td></tr>
<tr><td>!</td><td>11</td><td>(unary) negation</td></tr>
</table>

All these infix operators are left associative.

Well-ingrained notions about the syntax of expressions:

- parenthesization,
- operator infix,
- operator precedence, and
- associativity of operators

cover the unfortunate difficulties in apprehending precisely the reading of compound expressions.


XKCD PRESENTS:
SOME NEW
SCIENCE MNEMONICS
ORDER OF OPERATIONS
PARENTHESES, EXPONENTS, DIVISION & MULTIPLICATION, ADDITION & SUBTRACTION
TRADITIONAL: PLEASE EXCUSE MY DEAR AUNT SALLY
RIGHT THERE.
PLEASE EMAIL MY DAD A SHARK
OR: PEOPLE EXPECT MORE DRUGS AND SEX
SI PREFIXES
KILO, MEGA, GIGA, TERA, PETA, EXA, ZETTA, YOTTA
MILLI, MICRO, NANO, PICO, FEMTO, ATTO, ZEPTO, YOCTO
TRADITIONAL: [I NEVER LEARNED ONE]
$
RISE!
ER.
WHAT DO WE DO WITH THEM?
BIG:
KARL MARX GAVE THE PROLETARIAT ELEVEN ZEPPELINS, YO.
SMALL:
MICROSOFT MADE NO PROFIT FROM ANYONE'S ZUNES, YO.
TAXONOMY
KINGDOM, PHYLUM, CLASS, ORDER, FAMILY, GENUS, SPECIES
TRADITIONAL: KING PHILIP CAME OVER FOR GOOD SEX
I'M NOT SURE WHO DOUBTS THIS, REALLY.
KATY PERRY CLAIMS ORGASMS FEEL GOOD SOMETIMES
OR: KERNEL PANICS CRASH OUR FAMILY GAME SYSTEM.
GEOLOGIC PERIODS
(PRECAMBRIAN) CAMBRIAN ORDOVICIAN SILURIAN DEVONIAN CARBONIFEROUS PERMIAN TRIASSIC JURASSIC CRETACEOUS PALEOGENE NEOGENE
TRADITIONAL: [I NEVER LEARNED ONE]
POLYCYSTIC OVARIAN SYNDROME DOES CAUSE PROBLEMS THAT JUDICIOUS CONTRACEPTIVES PARTIALLY NEGATE.
RESISTOR COLOR CODES
BLACK BROWN, RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET, GRAY, WHITE
TRADITIONAL: [NONE I CARE FOR]
NANOBOT
VACCINE
9/11
"BIG BROTHER REPTILIAN OVERLORDS," YELLED GLENN, "BRAINWASHING VIA GROUND WATER!!"
OR: BE BOLD, RESPECT OTHERS; YOU'LL GRADUALLY BECOME VERSATILE, GREAT WIKIPEDIANS!
PLANETS
MERCURY VENUS EARTH MARS JUPITER SATURN URANUS NEPTUNE
TRADITIONAL: MY VERY EXCELLENT MOTHER JUST SERVED US NACHOS
UH HUH.
MARY'S "VIRGIN" EXPLANATION MADE JOSEPH SUSPECT UPSTAIRS NEIGHBOR

Precedence

- US. PEDMAS: parentheses, exponents, multiplication, division, addition, subtraction

  Please Excuse My Dear Aunt Sally
  Please Email My Dad A Shark
  Peolple Expect More Drugs And Sex

- Canada and NZ. BEDMAS: brackets, exponents, division, multiplication, addition, subtraction
- UK, India, and Australia. BODMAS: brackes, of order (powers/exponents), division, multiplication, addition, subtraction

Precedence

Additions and substration have *same* precedence
Multiplication and divison have *same* precedence

Parentheses

With parentheses you can express the order of evaulation unambiguously.
Too many parenthesis lead to LISP syndrom (lots of inance silly parenetheses) which why we establish rules to avoid them. These rules are invisible and so are a source of problems.
Because they are visble and common, parenthese get a lot of emphasis. Parentheses are necessary to express trees in linear form. Expresses things linearly is necessary and good, but one should focus on what things are and not how they appear.

Short-circuit Evaluation

Among the important issues relating to the runtime evaluation of expressions is *short-circuit* evaluation.
Short-circuit evaluation is possible with operators that may omit the evaluation of an operand (usually the second operand), when the final value is determined without knowing that operand's value. The name comes from the analogy with an electrical systems that finds a circuit which is shorter than the one planned by the designer by jumping a narrow gap between wires or devices.

Short-circuit Evaluation

Some expressions denote "bad values" (runtime errors or nonterminating computations) only under certain circumstances. Knowing the order of evaluation allows the programmers to write simpler and clearer code.

```
if x<>0 and y/x < 1 then ... else ...
```

In Pascal, for example, this could lead to trouble when `x` has the value zero, for the language requires that the programmer assume the expression `y/x` may be evaluated. In Pascal the order of evaluation is left unspecified. Here is a selection from the Pascal report, page 21:

> *The rules of Pascal neither require nor forbid the evaluation of the second part in such cases. This means that the programmer must assure that the second factor is well-defined, independent of the value of the first factor.*

Short-circuit Evaluation

```
if (very_unlikely() and very_costly()) then ... else ...

p = my_list;
while (p && p->key != val) p=p->next

p := my_list;
while (p <> nil) and (p^.key <> val) do p:=p^.next;

while (i<ub and A[i]<>' ') do i:=i+1;
```

Significance

The significance of expressions is that they can often be evaluated in a simple manner.

Rewriting

```
fun square x = x * x;
fun sos (x,y) = (square x) + (square y);

sos (3,4)
 ==> (square 3) + (square 4)    [Def'n of sos]
 ==> 3*3 + (square 4)           [Def'n of square]
 ==> 9 + (square 4)             [Def'n of *]
 ==> 9 + 4*4                    [Def'n of square]
 ==> 9 + 16                     [Def'n of *]
 ==> 25                         [Def'n of +]
```

Expressions in imperative languages `x` or `f(x)` hardly ever denote the same value when they appear elsewhere. Because of `:=`, side effects, aliases, global variables, static variables, etc.

Rewriting

Four rules / equations:

```
fun len x = foldr (const (+1)) 0 x

fun const x y = x

fun foldr f v []      = v
    foldr f v (x:xs) = f x (foldr f v xs)
```

Rewriting

```
len ("ab":"":"z":[])
```

Rewriting

```
len ("ab":"":"z":[])

foldr (const (+1)) 0 ("ab":"":"z":[]) -- len rule
```

Rewriting

```
len ("ab":"":"z":[])

foldr (const (+1)) 0 ("ab":"":"z":[]) -- len rule

(const (+1)) "ab" (foldr (const (+1) 0) ("":"z":[]))
```

Rewriting

```
len ("ab":"":"z":[])

foldr (const (+1)) 0 ("ab":"":"z":[]) -- len rule

(const (+1)) "ab" (foldr (const (+1) 0) ("":"z":[]))

(+1) (foldr (const (+1) 0) ("":z":[])) -- const rule
```

Rewriting

```
(+1) (((const (+1)) "") (foldr (const (+1)) 0 ("z":[])))
```

Rewriting

```
(+1) (((const (+1)) "") (foldr (const (+1)) 0 ("z":[])))

(+1) ((+1) (foldr (const (+1)) 0 ("z":[])))
```

Rewriting

```
(+1) (((const (+1)) "") (foldr (const (+1)) 0 ("z":[])))

(+1) ((+1) (foldr (const (+1)) 0 ("z":[])))

(+1) ((+1) ((const (+1) "z") (foldr (const (+1)) 0 ([]))))
```

Rewriting

```
(+1) (((const (+1)) "") (foldr (const (+1)) 0 ("z":[])))

(+1) ((+1) (foldr (const (+1)) 0 ("z":[])))

(+1) ((+1) ((const (+1) "z") (foldr (const (+1)) 0 ([]))))

(+1) ((+1) ((+1) (foldr (const (+1)) 0 ([]))))

(+1) ((+1) ((+1) 0))
(+1) ((+1) 1)
(+1) 2
3
```

Referential Transparency

The identity of indiscernibles (Leibniz's Law):

> *Eadem sunt, quorum unum potest substitui alteri salva veritate.*
>
> *Those things are identical of which one can be substituted for the other without loss of truth.*

Wilhelm Gottfried Leibniz (1646–1716), *Discourse on Metaphysics*, Section 9

Referential Transparency

Willard Quine uses the phrase to refer to the substitutivity of identities. For example, in the sentence

Tully was a Roman.

the word “Tully” may be replaced by “Cicero,” which was another name of the same man. But the sentence

William Rufus was so-called because of the colour of his hair.

becomes untrue if we replace “William Rufus” by another description of the same man, “King William II.”

Referential Transparency

We see the same idea reflected in Frege's "Über Sinn und Bedeutung", *Zeitschrift für Philosphie und philosphische Kritik*, new series, 100, 1892, pages 25–50.

> *The meaning of a sentence must remain unchanged when a part of the sentence is replaced by an expression having the same meaning.*

> *die Bedeutung eines Satzes sein Wahrheitswert ist, so muß dieser unverändert bleiben, wenn ein Satzteil durch einen Ausdruck von derselben Bedeutung, aber anderem Sinne ersetzt wird. Und das ist in der Tat der Fall. Leibniz erklärt geradezu: "Eadem sunt, quae sibi mutuo substitui possunt, salva veritate".*

Referential Transparency

The identify of indiscernibles:

> *Things are the same which can be substituted for each other without loss of truth.*
>
> *If our supposition that the reference of a sentence is its truth value is correct, the latter must remain unchanged when a part of the sentence is replaced by an expression having the same reference. And this is in fact the case. Leibniz gives the definition: 'Eadem sunt, quae sibi mutuo substitui possunt, salva veritate.' What else but the truth value could be found, that belongs quite generally to every sentence if the reference of its components is relevant, and remains unchanged by substitutions of the kind in question?*

Translation by Max Black.

Referential Transparency

A language is referentially transparent if any sub-expression can be replaced with any sub-expression of equal value anywhere in the language.

Can you find a context in a programming language in which 1+2 is not equal to three?

Can you find a context in a programming language in which 1+2 is not equal to three?

```
// Comment:  1+2 is three
"1+2"
1+2*3
```