Preview of Expressions

- Literals
- Constants
- Operators and their operations
- Order of evaluation
  precedence, parentheses, associativity, short-circuit
- Referential transparency (Sebesta, 9th ed., Section 15.4, page 660.)
**Literal.** A *literal* is a lexical element of the program (not an identifier) standing for a specific, unchanging value known by the compiler.
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
Strings

C#: Verbatim string literals @"..." (no escape sequences except the quote escape sequence—two quote marks).
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"
"

Scala

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

Python (3.6)
```
sys.stdout.write (f"Area is {abs((x-z)*(y-w)):.2f} square cm.
"
```

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

C#
```
Console.WriteLine($"Area is ${abs((x-z)*(y-w)):.2f} square cm.
")
```
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);
```
in C# you have both compile-time and run-time constants:

```csharp
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:

```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";
```
Expressions

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

Examples in Ada

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>1</td>
</tr>
<tr>
<td>xor</td>
<td>1</td>
</tr>
<tr>
<td>or else</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
</tr>
<tr>
<td>and then</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
</tr>
<tr>
<td>/=</td>
<td>2</td>
</tr>
<tr>
<td>&lt;</td>
<td>2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>2</td>
</tr>
<tr>
<td>&gt;</td>
<td>2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>2</td>
</tr>
<tr>
<td>in</td>
<td>2</td>
</tr>
<tr>
<td>not in</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>&amp;</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>/</td>
<td>4</td>
</tr>
<tr>
<td>mod</td>
<td>4</td>
</tr>
<tr>
<td>rem</td>
<td>4</td>
</tr>
<tr>
<td>**</td>
<td>5</td>
</tr>
<tr>
<td>abs</td>
<td>5</td>
</tr>
<tr>
<td>not</td>
<td>5</td>
</tr>
</tbody>
</table>

**is non associative; all others are left associative**
<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR</td>
<td>1</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
</tr>
<tr>
<td>NOT</td>
<td>3</td>
</tr>
<tr>
<td>=</td>
<td>4</td>
</tr>
<tr>
<td>#</td>
<td>4</td>
</tr>
<tr>
<td>&lt;</td>
<td>4</td>
</tr>
<tr>
<td>&lt;=</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>4</td>
</tr>
<tr>
<td>&gt;=</td>
<td>4</td>
</tr>
<tr>
<td>IN</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>5</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>&amp;</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>6</td>
</tr>
<tr>
<td>/</td>
<td>6</td>
</tr>
<tr>
<td>DIV</td>
<td>6</td>
</tr>
<tr>
<td>MOD</td>
<td>6</td>
</tr>
</tbody>
</table>

All infix operators are left associative.
C, C++, Java, C# Operators

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

all infix operators are left associative
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. 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;
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, . . .
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

\[
\text{len } ("ab":"":"z":[])\]

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

\[
(+1) \text{ (foldr (const } (+1) \text{) } 0 \text{) } ("":z":[])\] -- \text{const rule}

Rewriting

\[
\text{len } ("\text{ab}"":"":"\text{z}" : [])
\]

\[
\text{foldr } (\text{const } (+1)) \ 0 \ ("\text{ab}"":"":"\text{z}" : []) \ -- \ \text{len rule}
\]
Rewriting

\[
\text{len ("ab":"":"z":[])}
\]

\[
\text{foldr (const (+1)) 0 ("ab":"":"z":[])} \quad \text{-- len rule}
\]

\[
(\text{const (+1)}) \text{ "ab" (foldr (const (+1) 0) ("":"z":[]))}
\]
Rewriting

\[ \text{len ("ab":"":"z":[])} \]

\[ \text{foldr (const (+1)) 0 ("ab":"":"z":[])} \quad -- \quad \text{len rule} \]

\[ (\text{const (+1)}) \ "ab" \ (\text{foldr (const (+1) 0)} ("":"z":[])) \]

\[ (+1) \ (\text{foldr (const (+1) 0)} ("":"z":[])) \quad -- \quad \text{const rule} \]
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 Leibnizean principle:

*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*
Referential Transparency

Willard Quine uses the phrase to refer to the substitutivity of identities. E.g., 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 phrase

*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 Philosophie und Philosophische 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”.

The identity 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, then the truth value 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 subexpression can be replaced with any subexpression 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