CSC347 Concepts of Programming Languages

Lecture Notes: Expressions and Statements

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Language Development: How Do We Support Computations?

The most basic function of computer programs is to aid humans in automating tedious computational tasks, like adding two numbers. In this lecture, we explore the following questions:

- ❓ What building blocks should a programming language provide to let programmers express such computations in a convenient way?
- ❓ How should we integrate auxiliary functionality, such as printing the result of a computation?
- ❓ Which expressions must be fully evaluated to produce a result, and which ones can be partially evaluated?
- ❓ What corner cases of expression evaluation exist and how should we treat them?

Learning Goals

- 🎓 Understand different ways of expressing computations
- 🎓 Understand the importance of sequencing in computations
- 🎓 Understand the difference between statements and expressions
- 🎓 Understand strict vs. non-strict expression evaluation
- 🎓 Understand how C treats corner cases by leaving them "undefined" and how this leads to *unsafe* programs

Expressions and Statements

Assembly language consists of *statements* that are direct step-by-step instructions for a machine to execute. Let us start our journey of designing a programming language by asking ourselves which higher-level constructs (but still basic building blocks) than direct machine instructions are useful to express computations? Answering this question in an entirely abstract way is difficult, which is why often programming language designers have some application domain in mind. Still, there are some general-purpose mechanisms that we may find useful across many application domains: one such example is that users may want to compute the values of arithmetic expressions, or perform (limited) logic reasoning.

Expressions, in contrast to statements, are descriptive in nature; they describe a computation (e.g., the addition $1 + 2$) and were added later to higher-level languages. *Pure expressions* describe functions in the mathematical sense, they take arguments and return a result. Anything else, such as assigning to a variable, changing the control flow (e.g., goto) or IO operations (e..g, printing to the console, writing to a file or network), is a *side effect*. As a rough distinction, statements change state, expressions should not.

Example 1 (Is this C?). Since C is a prerequisite to this course, let's build on our shared understanding to inspect expressions and statements. Below is a short code snippet of a C function that returns 1 if its input is non-zero (remember that in a conditional statement, 0 means false, any non-zero value means true), else it returns 2. We can now ask whether this code snippet is a well-formed program in C.

```
1 int f (int x) {
2     int y;
3     if (x) y=1; else y=2;
4     return y;
5 }
6 int main() { printf ("%d\n", f(5)); return 0; }
```

This question is about the syntax of the programming language, which means that we can ask the C compiler whether the code snippet is a syntactically well-formed program in C.

```
1 gcc example1.c
```

In this case the compiler reassures us that it is. Let's take a closer look at the code snippet and we might notice that there both branches of the conditional statement share an assignment to the variable y. What if we pull out that assignment to remove this "code duplication", is the result still C?

```
1 int f (int x) {
2     int y;
3     y = if (x) 1; else 2;
4     return y;
5 }
6 int main() { printf ("%d\n", f(5)); return 0; }
```

Now the answer is no. The reason is that the code snippet above (in terms of C) uses a statement (the conditional statement `if (x) 1; else 2;`) on the right-hand side of an assignment, where an expression is expected.

Expressions evaluate to values and do not have side effects (do not change program state, such as write to variables, write to the console or a file), whereas statements may have side effects.

There are now two interesting aspects that allow us to inspect expressions vs. statements in C. Let's first dissect the conditional statement `if (x) 1; else 2;`, which you may remember executes the statement on the "if-branch" when x is true, else it executes the statement on the "else-branch". You may now ask: how are 1 and 2 statements? After all, they do not have side effects and are just numbers that can equally well appear in expressions like $x + 1$. In C, any expression can be turned into a statement using the ";" operator, so `1` is an expression, but `1;` is a statement.

Returning to the example, in order to get well-formed C we need to turn the right-hand side of the assignment to y into an expression. Since conditional evaluation of expressions is such a common use case, C provides dedicated expression syntax in terms of a (C's only) ternary operator `x ? 1 : 2`. Using this operator, we can now rephrase our example to again obtain well-formed C.

```
1 int f (int x) {
2     int y;
```

```
3       y = x ? 1 : 2;
4       return y;
5   }
6   int main() { printf("%d\n", f(5)); return 0; }
```

What if our conditional expression is more complicated and requires an algorithm to compute, could we still express that inline like below? Not in C, but some languages support such syntax.

```
1   int f (int x) {
2       int y;
3       y = {int z=0; while (x>0) {x--; z++;} z}
4       return y;
5   }
6   int main() { printf("%d\n", f(5)); return 0; }
```

Expressions

Expressions in C are constructed from the following components:

- Literals (boolean, character, integer, string)
- Operators (arithmetic, bitwise, logical)
- Function calls, e.g., f(1+(2* strlen("hello")))

C knows *side-effecting expressions*, such as x++, x+=2, x=(y=5), x-=(y+=5), and any function that changes (global) memory or writes to some resource.

Example 2 (Side-effecting Expressions in C). The example below is bad programming style, but should give you an idea of how side-effecting expressions operate.

```
1   int global = 0;
2
3   int post_inc () {
4     return global++;
5   }
6
7   int main () {
8     printf("%d\n", post_inc () + post_inc ());
9   }
```

The effect of this program is printing 1; if we change line 4 to **return** ++global; the output would change to 3.

Note 1. The code above is clearly an example of bad programming style, which reminds us that we must be thoughtful in how we use programming language; often, programming languages provide concepts that are useful for same use case or help in phrasing arguments in a crisp and clear way, but may result in code that is hard to understand or maintain when misused. As designers of programming languages, we should strive for a language design that makes it difficult or impossible to write incorrect programs (here, with incorrect we mean programs that violate functional or non-functional properties of programs).

A useful way of using side-effecting expressions is in combination with IO operations, whose return value in C usually indicates success or failure, while the result is passed back using a pointer argument.

Example 3 (A Useful Example of Side-effecting Expressions).

```
string s;
while (read_string(s), s.len() > 5) {
    // do something
}
```

This examples uses the operator "," of the shape $(e_1, e_2, \ldots, e_n)$ for sequential execution of expressions. This operator executes expressions $e_1 \ldots e_{n-1}$ sequentially for their side effect, followed by e_n whose result is the result of the operator. In the example above, the function `read_string` is called repeatedly to store the result in the string s; the loop keeps repeating as long as the length of the string read exceeds 5.

Sequencing can also be used to write programs more concisely, as illustrated with the following two examples.

```
int main () {
    int x = 5;
    x *= 2;
    printf ("%d\n", x);
}
```

```
int main () {
    int x = 5;
    printf ("%d\n", (x *= 2, x));
}
```

In summary, besides atomic expressions, C provides compound expressions for sequencing (`e1 ,..., en`) and conditional evaluation of expressions (`e1 ? e2 : e3`).

Statements

Statements in many imperative programming languages, like C, are the most common way of sequencing operations and deciding between different computation options. They typically operate with mutable data (i.e., change memory locations by changing the values of variables) and do not produce themselves a result.

- Return statements
- Selection, such as if-then-else, switch-case
- Iteration, such as while, do-while, for
- Expression statements (expressions terminated in ";") including assignment

Strict and Nonstrict Expression Evaluation

Expressions are evaluated for their value, but do we need to evaluate all operands of an expression in order to determine the value of a compound expression? You may already guess that

the answer is no: strict expression evaluation evaluates all operands, whereas nonstrict expression evaluation evaluates operands only until the value of the result is determined regardless of the values of the remaining operands.

Exercise 1 (Name some strict constructs in C). Before you read on, take some time to name strict constructs in C.

- Arithmetic operators +, −, *, … (interestingly, multiplication could be non-strict but only for the concrete argument 0)
- Comparison operators <, …
- Bitwise operators |, …
- Function calls
- …

Exercise 2 (Name some nonstrict constructs in C). Before you read on, take some time to name strict constructs in C.

- Boolean operators &&, … (e1 && e2 is strict in e1 but non-strict in e2; another example is p && p->f() to check a pointer before dereferencing it)
- Conditional operator e1 ? e2 : e3 is strict in e1 but not in e2 or e3
- Macro expansion

Nonstrict evaluation is important because it allows us to define recursive algorithms. Let us inspect an example that illustrates nonstrict evaluation by conditionally modifying one of two memory locations.

Example 4 (Non-strict evaluation of operands of C's ternary operator).

```
1  int main () {
2      for (int i=0; i<10; i++) {
3          int x = 0;
4          int y = 0;
5          int z = (rand()%2) ? (x=1,111) : (y=2,222);
6          printf ("x=%d, y=%d, z=%d\n", x, y, z);
7      }
8  }
```

Save this code snippet as `statements-01.c`. We compile and execute the programming as below.

```
1  $ gcc statements-01.c && ./a.out
2  x=1, y=0, z=111
3  x=0, y=2, z=222
4  ...
```

Programming languages often have more than one way of expressing the same computation in several different ways. Below we compare an implementation of computing the factorial of a number using conditional statements vs. conditional expressions. Nonstrict (partial) expression evaluation is vital in such a recursive algorithm, because it allows us to terminate with the base case; with strict expression evaluation, we would continue evaluating the recursive case, even when we arrive at the base case.

Exercise 3 (Conditional statements vs. conditional expressions).

```
int fact(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

```
int fact(int n) {
    return (n <= 1) ? 1 : n * fact(n - 1);
}
```

Execute both programs to see whether there is a difference in how they execute. What happens?

```
int f(int c, int t, int f) { return c ? t : f; }
int main() {
    for (int i=0; i<10; i++) {
        int x = 0;
        int y = 0;
        int z = f(rand()%2, (x=1,111), (y=2,222));
        printf("x=%d, y=%d, z=%d\n", x, y, z);
    }
}
```

Remember that function calls are strict! As a result, the code above evaluates all operands to f, which undoes the non-strict evaluation of the conditional expression that is used to implement the function. Macro calls, in contrast, are non-strict.

```
#define m(b, t, f) (b)?(t):(f)
int main () {
    for (int i=0; i<10; i++) {
        int x = 0;
        int y = 0;
        int z = m(rand()%2, (x=1,111), (y=2,222));
        printf ("x=%d, y=%d, z=%d\n", x, y, z);
    }
}
```

The reason is that macro calls are evaluated in the preprocessor stage during compilation, by textual substitution, and our specific textual substitution introduces the nonstrict ternary conditional operator. This means, that the compiler does not "see" the code that may look to a programmer like a function call; the non-strict ternary operator is substituted directly into main.

```
int ftriple(int i) { return i + i + i; }
#define mtriple(i) (i)+(i)+(i)
int main () {
  int x = 10;
  int rx = ftriple(x=x+1);
```

```
6     int y = 10;
7     int ry = mtriple(y=y+1);
8     printf ("x=%d, rx=%d, y=%d, ry=%d\n", x, rx, y, ry);
9  }
```

Exercise 4 (Another way of expressing factorial). Before you read on, can you think of another way of expressing this program?

```
1      int fact(int n) {
2          switch (n) {
3              case 1: return 1;
4              default: return n * fact(n-1);
5          }
6      }
```

Undefined Behavior

Many mathematical functions $f : A \to B$ are not defined over their entire domain (remember that we call the values of the set A a function's domain, and the values of the set B its range or co-domain). For example, division $\text{div} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a function that takes two real values and returns one real value, but division by zero is undefined. When we specify division as an operation in a computer, however, we need to specify how it behaves on all inputs from the domain. There are several ways of achieving this.

Exercise 5 (Language constructs for undefined behavior). Before you read on, can you think of some ways of specifying a function such that it avoids undefined behavior?

- Define a type $\mathbb{R}_{\neq 0} \equiv \mathbb{R} \setminus \{0\}$ and define division as $\text{div} : \mathbb{R} \times \mathbb{R}_{\neq 0} \to \mathbb{R}$
- Check arguments in the function definition and use exceptions to report violation
- Make it the responsibility of the user to not misuse the function (i.e., the C approach, just return any value on division by zero)
- Define division by zero to be zero (the approach in the theorem prover Isabelle)

This example is also a good reminder that numerical computations with mathematical sets of infinite size, such as $\mathbb{R}$, are challenging to be represented in a computer with finite memory and finite bitsize representations (symbolically, we can represent and reason about the properties of infinite sets and operations on infinite sets, with finite memory we just cannot represent every element of an infinite set with its own symbol). Specific programming languages may have other sources of undefined behavior. For example, in C, over- and underflow of variables, evaluation sequence in expressions, and dangling pointers can result in undefined behavior.