TED Language Reference Manual

Theodore Ahlfeld(twa2108), Konstantin Itskov(koi2104)

Matthew Haigh(mlh2196), Gideon Mendels(gm2597)

Preface

This manual describes the language TED as of its initial conception. The language TED (Token Expression Determiner) is a text parser specialized for web page parsing though It can still be used to parse any text.

1. Lexical Elements

This chapter describes the lexical elements that make up a TED source code file. Lexical elements will be referred to as tokens. These tokens will be categorized as: Identifiers, Keywords, Constants, Operators, Separators, and Whitespace.

1.1 Identifiers

Identifiers are sequences of characters used for naming variables, functions, or data types. These identifiers can include letters, digits, and underscores '_', but must begin with a letter. Identifiers are case sensitive.

1.2 Keywords

Keywords are special identifiers reserved for the compiler and the language itself. They cannot be redefined or reused for any other purpose.

```
if, then, else, for, while, int, float, str, list, FILE, Page, Element, return
```

1.3 Constants

A constant is a literal value. All constants belong to a language defined data types.

1.3.1 Integer Constants

All integer constants must be in the base ten number system. Any integer value prefix with a hyphen '-' is treated as a negative integer.

1.3.2 String Constants

TED does not allow character manipulation explicitly. To prevent unsafe string manipulation that has plagued both C and C++, strings are a primitive type. String constants are a double quoted sequence of characters.

```
"Example"
"String Constants"
```

1.3.2.1 String Escape Constants

Despite the lack of characters, many of C's escape characters are recycled into TED's string escape constants. They are two character sequences with a backslash followed by a character. Escape sequences are as follows

```
Backslash:                  "\\"
Single quotation mark:      "\'"
Double quotation mark:      "\""
Backspace:                  "\b"
Newline:                    "\n"
Carriage return:            "\r"
Horizontal Tab:             "\t"
```

1.3.3 Real Number Constants

A real number constant is a value that represents a floating point number. The are represented as a sequence of digits, which contains an integer part, a decimal, and another sequence of numbers consisting of the fractional portion. For example:

```
1.234
0.1234
.1234
1324.
1234.0
```

1.4 Operators

An operator is a special token that performs either a unary or binary operation. They can perform operations such as addition, complements, etc. Full coverage of operation is provided in chapter 3, “Expressions and Operators”.

1.5 Separators

Separators are single characters which separate tokens such as:

```
( ) { } ; , . :
```

In addition white spaces, which are still considered separators, and are not considered tokens and are ignored during parsing.

1.6 Whitespace

Whitespace is any permutation of either spaces, tabs, newline, carriage return, or form feeds. White space is ignored, outside of string constants, and is therefore considered as an option with an exception of separating tokens. White space is not required to separate operators from their operand, and any whitespacing between an operand and operators are ignored.

1.7 Comments

Comments are indicated by the symbol “/*”. Any input read after a comment symbol will be ignored until the end symbol “*/” is encountered. Nested comments are not allowed.

2. Data Types

2.1 Primitive Data Types

2.1.1 Integer Data Types

int : The 32-bit int data type can hold integer values in the range of −2,147,483,648 to 2,147,483,647

char: char data type is a single 16-bit Unicode character. Minimum value is '\u0000' (or 0). Maximum value is '\uffff' (or 65,535 inclusive).

2.1.2 Float Data Types

float: Float data type is a single-precision 32-bit IEEE 754 floating point.

2.1.3 String Data Types

str: a sequence of 16 bit Unicode characters

2.1.4 List Data Types

list: a linked list of memory references to objects. Can refer to any type but all objects in a list must be of the same type.

2.2 Object Data Types

2.2.1 Element Data Types

Contains seven member values: id, class, html, text, type, attr and children. id is the html id attribute, class is the html class attribute, html is the entire raw html data associated with the element, and text is the plain-text content of the element. Type is the html tag, and attr is a list of all html attributes associated with the element, including class and id. Children is a list of all child elements of the HTML DOM element. All member values are represented as str values internally.

2.2.2 Page Data Types

Contains four member values: response, url, children and html. response is the HTTP response string that was returned when the page was fetched, url is a str value which corresponds to the base url of the website being parsed. Children refer to the top level DOM elements and HTML is the raw html content.

3. Expressions and Operators

3.1 Expressions

Expressions consist of at least one operand and zero or more operators. Operands are typed objects, such as constants, variables, and function calls that must return values (non void functions). Examples:

```
42
40 + 2
f(42)  /* This assumes function f returns a non void value */
```

Parentheses group subexpressions in order to override default precedence.

```
(2 + 3) * (5*(2+2))/4 - 0
```

Innermost expressions are evaluated first. In the above example, 2+2 is first evaluated to be 4 before being multiplied by 5. Next, 2 + 3 = 5 is calculated, followed by the multiplication of the already calculated 5 * 20 to be 100, which is then divided by 4, ended with a minus 0.

3.2 Assignment Operators

Assignment operators store values in variables.

The standard assignment operator = simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand cannot be a literal or constant value. For example:

```
int x = 10;
float y = 45.12 + 2.0;
int z = (2 * (3 + function () ));
str a = "ted";
```

Additionally, expressions can be assigned, from left to right, in a sequence. For example, x = y = 2, evaluates 2, assigns it to y, which returns a value of 2, which is then assigned to x.

3.3 Arithmetic Operators

TED provides operators for standard arithmetic operations: addition, subtraction, multiplication, and division. Usage of these operators is straightforward; here are some examples:

```
/* Addition. */
x = 5 + 3;
y = 10.23 + 37.332;

/* Subtraction. */
x = 5 - 3;
y = 57.223 - 10.903;

/* Multiplication. */
x = 5 * 3;
y = 47.4 * 1.001;

/* Division. */
x = 5 / 3;
y = 940.0 / 20.2;
```

Integer division of positive values truncates towards zero, so 5/3 is 1.

3.4 Comparison Operators

The equal-to operator == tests its two operands for equality. The result is true if the operands are equal, and false if the operands are not equal.

```
if (x == y)
    print (stdout, "%s", "x is equal to y");
else
    print (stdout, "%s", "x is not equal to y");
```

The not-equal-to operator != tests its two operands for inequality. The result is true if the operands are not equal, and false if the operands *are* equal.

```
if (x != y)
    print (stdout, "%s", "x is not equal to y");
else
    puts (stdout, "%s", "x is equal to y");
```

Beyond equality and inequality, there are operators you can use to test if one value is less than, greater than, less-than-or-equal-to, or greater-than-or-equal-to another value. Here are some code samples that exemplify usage of these operators:

```
if (x < y)
    print (stdout, "%s", "x is less than y");

if (x <= y)
    print (stdout, "%s", "x is less than or equal to y");

if (x > y)
    print (stdout, "%s", "x is greater than y");

if (x >= y)
    print (stdout, "%s", "x is greater than or equal to y");
```

3.5 Conditional Operators

Logical operators test the truth value of a pair of operands. Every logical comparison must evaluate to true or false. The logical conjunction operator && tests if two expressions are both true. If the first expression is false, then the second expression is not evaluated.

```
if ((x == 5) && (y == 10))
    print (stdout, "%s", "x is 5 and y is 10");
```

The logical conjunction operator || tests if at least one of two expressions it true. If the first expression is true, then the second expression is not evaluated.

```
if ((x == 5) || (y == 10))
    print (stdout, "%s", "x is 5 or y is 10");
```

You can prepend a logical expression with a negation operator ! to flip the truth value:

```
if (!(x == 5))
    print (stdout, "%s", "x is not 5);
```

3.6 Bit Shifting

The left-shift operator << is used to shift its first operand's bits to the left. The second operand denotes the number of bit places to shift. Bits shifted off the left side of the value are discarded; new bits added on the right side will all be 0.

```
x = 47;    /* 47 is 00101111 in binary. */
x << 1;    /* 00101111 << 1 is 01011110. */
```

Similarly, you use the right-shift operator >> to shift its first operand's bits to the right. Bits shifted off the right side are discarded; new bits added on the left side are *usually* 0, but if the first operand is a signed negative value, then the added bits will be either 0 *or* whatever value was previously in the leftmost bit position.

```
x = 47;   /* 47 is 00101111 in binary. */
x >> 1;   /* 00101111 >> 1 is 00010111. */
```

For both << and >>, if the second operand is greater than the bit-width of the first operand, or the second operand is negative, the behavior is undefined.
The shift operators can be used to perform a variety of interesting hacks. For example, given a date with the day of the month numbered as d, the month numbered as m, and the year as y, the entire date can be stored in a single number x:

```
int d = 12;
int m = 6;
int y = 1983;
int x = ((y << 4) + m) << 5) +  d;
```

The original day, month, and year can then be extracted out of x using a combination of shift operators and modular division:

```
d = x % 32;
m = (x >> 5) % 16;
y = x >> 9;
```

3.7 Bitwise/Logical Operators

TED provides operators for performing bitwise conjunction, inclusive disjunction, exclusive disjunction, and negation (complement).

Bitwise conjunction examines each bit in its two operands, and when two corresponding bits are both 1, the resulting bit is 1. All other resulting bits are 0. Here is an example of how this works, using binary numbers:

```
11001001 & 10011011 = 10001001
```

Bitwise inclusive disjunction examines each bit in its two operands, and when two corresponding bits are both 0, the resulting bit is 0. All other resulting bits are 1.

```
11001001 | 10011011 = 11011011
```

Bitwise exclusive disjunction examines each bit in its two operands, and when two corresponding bits are different, the resulting bit is 1. All other resulting bits are 0.

```
11001001 ^ 10011011 = 01010010
```

Bitwise negation reverses each bit in its operand:

```
~11001001 = 00110110
```

In TED, you can only use these operators with operands of an integer type, and for maximum portability, you should only use the bitwise negation operator with unsigned integer types. Below are some examples of using these operators in TED code:

```
int foo = 42;
int bar = 57;
int bistromath;

bistromath  = foo & bar;
bistromath = foo | bar;
bistromath = foo ^ bar;
bistromath = ~foo;
```

3.8 Function Calls as Expressions

A call to any function which returns a value is an expression.

```
int function(do something);
...
a = 10 + function();
```

3.9 The Comma Operator

The comma operator is used in for statements as follows:

```
/* Using the comma operator in a for statement. */
for (x = 1, y = 10;  x <=10 && y >=1;  x++, y--) {
    ...
}
```

This allows the user to conveniently set, monitor, and modify multiple control expressions for the for statement.

A comma is also used to separate function parameters. Thus,

```
foo (a,  b,  c,  d);
```

is interpreted as a function call with four arguments.

3.10 Member Access Expressions

The member access operator . can be used to access the members of an Element or Page. The name of the object variable is put on the left side of the operator, and the name of the member on the right side.

```
Page x = get("http://mylittlepony.hasbro.com/en-us",NULL,NULL,NULL);
Element y = x.children.head;
Element z;
z.children[0] = y;
```

3.11 Statements and Declarations in Expressions

Variables must be declared outside of expressions, including for for loops. The incrementer variable must be declared outside of the conditional expression.

```
int i;
for (i = 0; i < 10; i = i +1) {
    do something ...
}

not allowed: b = (int a = 3) // must define a in a separate expression
int add (x,y) { return x + y }
int c = add(a = 3, b  = 2); //c can be declared here but not a or b
```

3.12 Operator Precedence

When an expression contains multiple operators, such as a + b * f(), the operators are grouped based on rules of precedence. For instance, the meaning of that expression is to call the function f with no arguments, multiply the result by b, then add that result to a. That is what the TED rules of operator precedence determine for this expression.

The following is a list of types of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right unless stated otherwise.

1. Function calls, array subscripting, and membership access operator expressions.
2. Unary operators, including logical negation, bitwise complement, and unary negative. When several unary operators are consecutive, the later ones are nested within the earlier ones: !-x means !(-x).
3. Multiplication, division, and modular division expressions.
4. Addition and subtraction expressions.
5. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
6. Equal-to and not-equal-to expressions.
7. Bitwise AND expressions.
8. Bitwise exclusive OR expressions.
9. Bitwise inclusive OR expressions.
10. Conditional AND expressions.
11. Conditional OR expressions.
12. All assignment expressions, including compound assignment. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.

4. Expressions and Operators

4.1 Expression Statements

You can turn any expression into a statement by adding a semicolon to the end of the expression. Here are some examples:

```
5;
2 + 2;
10 >= 9;
```

In each of the examples above, all that happens is that each expression is evaluated. However, they are useless because they do not store a value anywhere, nor do they actually do anything, other than the evaluation itself. The compiler is free to ignore such statements.

Expression statements are only useful when they have some kind of side effect, such as storing a value, calling a function, or (this is esoteric) causing a fault in the program. Here are some more useful examples:

```
x = x + 1;
y = x + 25;
print (stdout, "%s" "Hello, user!");
```

4.2 The if Statement

You can use the if statement to conditionally execute part of your program, based on the truth value of a given expression. Here is the general form of an if statement:

```
if (test) {
    then-statement
} else {
    else-statement
}
```

If test evaluates to true, then the then-statement is executed and else-statement is not. If test evaluates to false, then the else-statement is executed and then-statement is not. The else clause is optional. The execution code must be enclosed in braces.
Here is an actual example:

```
if (x == 10) {
    print (stdout, "%s", "x is 10");
}
```

If x == 10 evaluates to true, then the statement print (stdout, “%s”, "x is 10"); is executed. If x == 10 evaluates to false, then the statement print (stdout, “%s”, "x is 10"); is not executed. Here is an example using else:

```
if (x == 10) {
    print (stdout, "%s", "x is 10");
} else {
    print (stdout, "%s", "x is not 10");
}
```

You can use a series of if statements to test for multiple conditions:

```
if (x == 1) {
    print (stdout, "%s", "x is 1");
} else if (x == 2) {
    print (stdout, "%s", "x is 2");
} else if (x == 3) {
    print (stdout, "%s", "x is 13");
} else {
    print (stdout, "%s", "x is something else");
}
```

4.3 The while Statement

The while statement is a loop statement with an exit test at the beginning of the loop. Here is the general form of the while statement:

```
while (test) {
    statement
}
```

The while statement first evaluates test. If test evaluates to true, statement is executed, and then test is evaluated again. statement continues to execute repeatedly as long as test is true after each execution of statement. The execution code must be enclosed in brackets. The example below shows a while statement, which prints the integers from zero through nine:

```
int counter = 0;
while (counter < 10) {
    print (stdout, "%d ", counter);
    counter = counter + 1;
}
```

4.4 The for Statement

The for statement is a loop statement whose structure allows for easy expression testing, and variable modification. It is very convenient for making counter controlled loops. Here is the general form of the for statement:

```
for (initialize; test; step)
    statement
```

The for statement first evaluates the expression initialize and then it evaluates the expression test. If test is false, then the loop ends and program control resumes after statement. Otherwise, if test is true, then statement is executed. Finally, step is evaluated, and the next iteration of the loop begins with evaluating test again. Most often, initialize assigns values to one or more variables, which are generally used as counters, test compares those variables to a predefined expression, and step modifies those variables' values. Below is another example that prints the integers from zero through nine:

```
int x;
for (x = 0; x < 10; x++)
   printf ("%d ", x);
```

First, the variable x is declared outside the loop, then it evaluates initialize, which assigns x the value 0. Then, as long as x is less than 10, the value of x is printed (in the body of the loop). Then x is incremented in the step clause and the test re-evaluated. All three of the expressions in a for statement are optional, and any combination of the

three is valid. Since the first expression is evaluated only once, it is perhaps the most commonly omitted expression.

4.5 Blocks

A block is a set of zero or more statements enclosed in braces. Blocks are also known as compound statements. Often, a block is used as the body of an if statement or a loop statement, to group statements together.

```
for (x = 1; x <= 10; x++) {
    printf ("x is %d\n", x);
    if ((x % 2) == 0)
        printf ("%d is even\n", x);
    else
        printf ("%d is odd\n", x);
}
```

You can also put blocks inside other blocks:

```
for (x = 1; x <= 10; x++) {
    if ((x % 2) == 0) {
        printf ("x is %d\n", x);
        printf ("%d is even\n", x);
    } else {
        printf ("x is %d\n", x);
        printf ("%d is odd\n", x);
    }
}
```

Blocks have their own variable scope.

4.6 The break Statement

The break statement can be used to terminate a while, do, for, or switch statement. Below is an example:

```
int x;
```

```
for (x = 1; x <= 10; x = x + 1) {
    if (x == 8)
        break;
    else
        print (stderr, "%d ", x);
}
```

The above example prints numbers from 1 to 7. When x is incremented to 8, x == 8 is true, so the break statement is executed, terminating the for loop prematurely. If you put a break statement inside of a loop or a switch statement, which is inself inside of a loop or switch statement, the break only terminates the innermost loop or switch statement.

4.7 The goto Statement

The goto statement can be used to unconditionally jump to a different place in the program. Here is the general form of a goto statement:

```
goto label;
```

A label to jump to must be specified, and when the goto statement is executed, program control jumps to that label.

```
goto end_of_program;
...
end_of_program:
```

The label can be anywhere in the same function as the goto statement that jumps to it, but a goto statement cannot jump to a label in a different function.
You *can* use goto statements to simulate loop statements, but it is not recommended, because it makes the program harder to read, and GCC cannot optimize it as well. Instead it is preferable to use for, while, and do statements instead of goto statements, when possible.

4.8 The Label Statement

Labels can be used to identify a section of source code for use with a later goto (see The goto Statement). A label consists of an identifier (such as those used for variable names) followed by a colon. Below is an example: treet: You should be aware that label names do not interfere with other identifier names:

```
int treet = 5; /* treet the variable. */
treet: /* treet the label. */
```

4.9 The return Statement

The return statement can be used to end the execution of a function and return program control to the function that called it. Here is the general form of the return statement is as follows:

```
return return-value;
```

Functions must specify return type and must return that type under all conditions.

5. Functions

5.1 Function Declarations

A function declaration should be written to specify the name of a function, a list of parameters, and the function's return type. A function declaration ends with a semicolon. Here is the general form:

```
return-type function-name (parameter-list);
```

return-type indicates the datatype of the value returned by the function. function-name can be any valid identifier. parameter-list consists of zero or more parameters, separated by commas. A typical parameter consists of a data type and a name for the parameter. Here is an example of a function declaration with two parameters:

```
int zaphod (int x, double y);
```

The parameter names can be any identifier, and if there is more than one parameter, the same name cannot be used more than once within a single declaration. The parameter names in the declaration need not match the names in the definition.
The function declaration should be written above the first use of the function.

5.2 Function Definitions

You write a function definition to specify what a function actually does. A function definition consists of information regarding the function's name, return type, and types and names of parameters, along with the body of the function. The function body is a series of statements enclosed in braces; in fact it is simply a block. Here is the general form of a function definition:

```
return-type function-name (parameter-list) {
    function-body
}
```

return-type and function-name are the same as what is used in the function declaration. parameter-list is the same as the parameter list used in the function declaration. Names for the parameters must be included in a function definition. Here is an simple example of a function definition, which take two integers as its parameters and returns the sum of them as its return value:

```
int add_values (int x, int y) {
    return x + y;
}
```

5.3 Calling Functions

A function can be called by using its name and supplying any needed parameters. Here is the general form of a function is:

```
function-name (parameters);
```

A function call can make up an entire statement, or it can be used as a subexpression. Here is an example of a standalone function call:

```
foo (5);
```

In the above example, the function ‘foo’ is called with the parameter 5.
Here is an example of a function call used as a subexpression:

```
a = square (5);
```

Supposing that the function ‘square’ squares its parameter, the above example assigns the value 25 to a.

If a parameter takes more than one argument, the parameters as separated with commas:

```
a = vogon (5, 10);
```

5.4 Function Parameters

Function parameters can be any expression—a literal value, a value stored in variable, or a more complex expression built by combining the two. Within the function body, the parameter is a local copy of the value passed into the function; you cannot change the value passed in cannot be changed by changing the local copy.

```
int x = 23;
foo (x);
...
/* Definition for function foo. */
int foo (int a) {
    a = 2 * a;
    return a;
}
```

In the above example, even though the parameter a is modified in the function ‘foo’, the variable x that is passed to the function does not change. In order to use the function to change the original value of x, the function call would need to be incorporated into an assignment statement:

```
x = foo (x);
```

5.5 Recursive Functions

It is possible to write a function that is recursive—a function that calls itself. Here is an example that computes the factorial of an integer:

```
int factorial (int x) {
    if (x < 1) {
        return 1;
    }
    else {
        return (x * factorial (x - 1));
    }
}
```

Caution must be taken to ensure a function is not written to be infinitely recursive. In the above example, once x is 1, the recursion stops. However, in the following example, the recursion does not stop until the program is interrupted or runs out of memory:

```
int watermelon (int x) {
    return (watermelon (x));
}
```

Functions can also be indirectly recursive, of course.

5.6 Built in Functions

TED comes with a small set of built in functions for basic functionality.

5.6.1 The print Function

TED offers limited output. The only output function is print, which behaves very similarly to the write system call for linux and prints a string to specified file.

```
print(stdout, "This will output to standard out");
```

To print to a specified file would be similar to

```
FILE ted_fp = fopen("C:\temp\tedoutput.txt");
print(ted_fp, "Prints this string to file");
```

5.6.2 The get Function

TED offers only one way to connect over the network and retrieve pages. The get function is the only way to create a page. It connects to the specified web address and loads the url into the page element along with a list of top level DOM elements (as children). The get function will create a new HTTP GET request to the specified URL, download the response, and save it to memory. The response header that contains the http code returned from the server will be saved into the Page element response member element.

The get function has a few variables to control the underlying network socket:

- timeout = default 15 seconds
- maxsize = default 1mb
- user agent = default “TED Scraper / 1.0”

There get function is called by

get(“http address”, “useragent”, timeout_int, maxsize_int);

The useragent, timout_int, and maxsize_int, are optional. The use their default value, they must have NULL in their place.

5.6.3 The find Function

Both Page and Elm come with a find function. They find function parses through the children list of their data looking for CSS matching. find will pass its parent's find function unless specified differently

```
Page page = get("www.ted-lang.com",NULL,NULL,NULL);
Elm elm = page.find("class","classname");
elm.find("id","id-name");
```

5.6.4 List Functions

List functions in TED are iterative as opposed to functional.

5.6.4.1 The head Function

The head function returns the head of the list.

```
Elm elm_node = elm_lst.head();
```

5.6.4.2 The next Function

Returns the next element in the list, and will return NULL if it does not exist.

```
Elm elm_node = elm_lst.head();
elm_node = elm_node.next();
```

5.6.4.3 The addafter Function

Adds a new element into the list after the specified node.

```
Elm elm_node = elm_lst.head();
Elm elm_new_node;
elm_node.addafter(elm_new_node);
```

5.6.4.3 The addbefore Function

Adds a new element into the list after the specified node.

```
Elm elm_node = elm_lst.head()
Elm elm_new_node;
elm_node.addbefore(elm_new_node);
```

5.6.4.3 The remove Function

Removes an element from the list.

```
Elm elm_node = elm_lst.head()
Elm elm_new_node = elm_node.next();
elm_lst.remove(elm_new_node);
```

6. Program Structure and Scope

6.1 Program Structure

TED can be compiled from a source file with the extension .ted or directly from the command line. In both cases, it will produce an output file with the .tedp extension. However, in the case of command line invocation, the compiled program will also be executed. Note that the invocation of the compilation process will require an input string that identifies the base web address that the program must crawl. The input string can contain more than one base address, in which case the compiled program will be executed on each of the strings sequentially. To do so, each base address may be separated by a newline character and any white spaces will be ignored. Thus, the list of base strings may be written into a text file with one string per line and provided to the program's input as input.

Multiple .ted files can be compiled together via the import command. Import statements connect modules or libraries to other files. Example:

```
import myModule.ted /*appends myModule.ted to the current position of the current file */
```

The code will be executed in the order that it appears.

```
import first.ted
print (stderr, "%s%, "foo");
import second.ted
/* will print "foo" before executing any code from second.ted */
```

It is recommended to include all import statements at the top of the program that will invoke functions from the imported files, and to limit execution to that main file (after all imports and function declarations).

6.2 Scope

Scope refers to what parts of the program can “see” a declared object. A declared object can be visible only within a particular function, or within a particular file, or may be visible to an entire set of files by way of import statements. Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions.

Declarations made within functions are visible only within those functions.
A declaration is not visible to declarations that came before it; for example:

```
int x = 5;
int y = x + 10;
```

will work, but:

```
int x = y + 10;
int y = 5;
```

will not.