
scalc

cmput415

Jun 11, 2025

CONTENTS

1	Reserved Keywords	3
2	Booleans	5
3	Integers	7
4	Identifiers	9
5	Expressions	11
5.1	Operators	11
5.2	Valid Expressions	11
5.3	Precedence	12
6	Statements	13
6.1	Declaration	13
6.2	Assignment	14
6.3	Conditional	14
6.4	Loop	14
6.5	Print	15
7	Comments	17
8	Backends	19
8.1	RISC-V	19
8.2	x86	20
8.3	ARM	21
8.4	Interpreter	22
9	Input	23
10	Output	25
11	Assertions	27
12	Clarifications	29
13	Deliverables	31
14	Tips and Hints	33
15	Using Inja	35

The goal of this assignment is to implement a compiler for a simple imperative language called *SCalc*. This compiler will directly generate code for the following three backends:

- *x86* assembly
- *RISC-V* assembly
- *ARM* assembly

You must also create an *interpreter* for *SCalc*

For the interpreter you will be computing the value of the expressions as you traverse the tree. However, when generating assembly code for each of the backends **you are not allowed to perform any computations in the compiler**. You must create assembly code to perform all of the computations that appear in the input file.

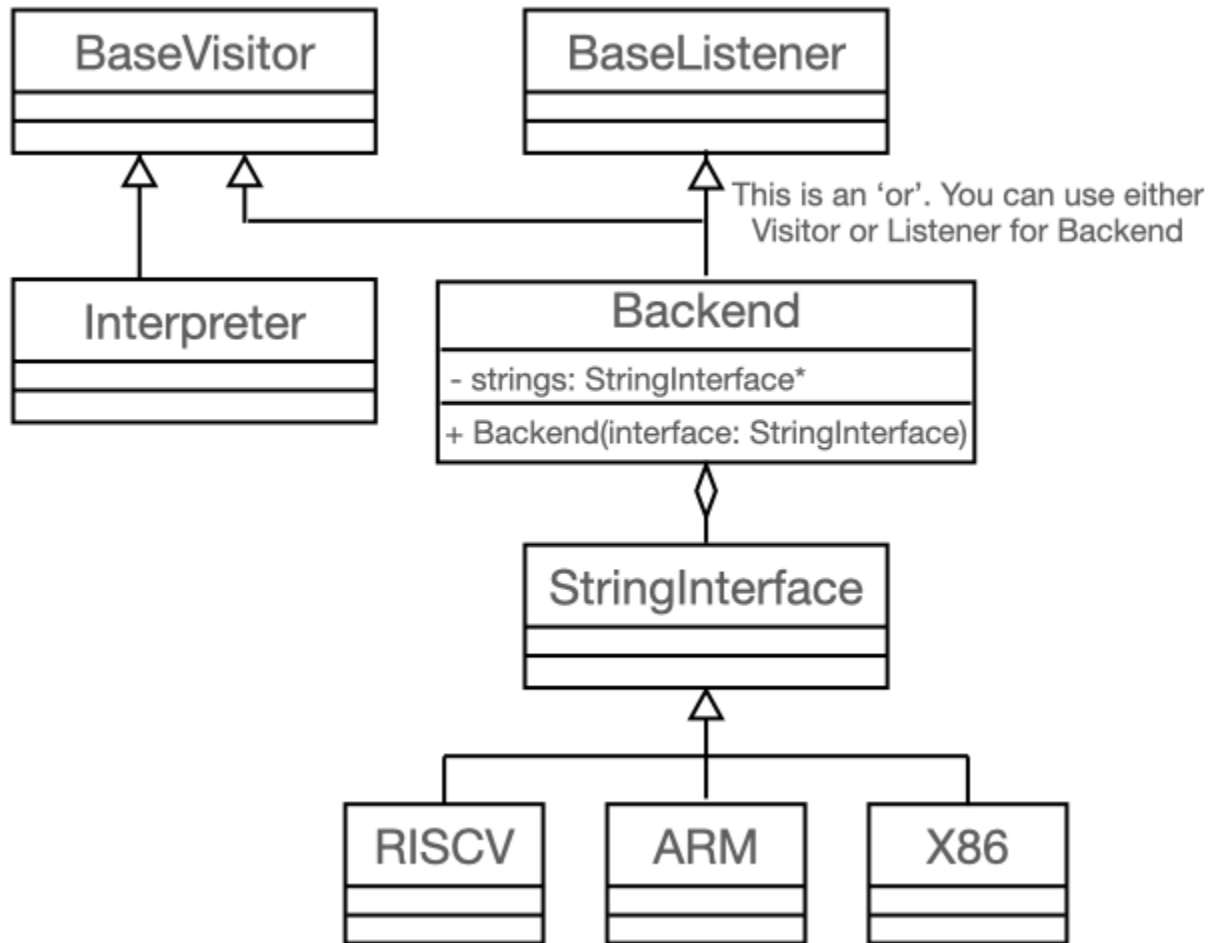
RISC-V assembly will be run using the RISC-V Assembler and Runtime Simulator, *RARS*

x86 assembly will be assembled using the nasm assembler and run natively.

ARM assembly will be compiled using the *ARM* toolchain (assembler `arm-none-eabi-as`, linker `arm-none-eabi-gcc`) and then run using an *ARM* simulator (*qemu-arm*).

Your compiler will produce the assembly text files, not binaries. **You must generate your assembly using [inja](#)**. Inja is a string template engine which allows you to make use of formatted strings. Its syntax is similar to *Python*'s *jinja*. You won't need any advanced features to complete this assignment.

It is recommended to follow a pattern of classes matching the following UML diagram (structure only, names should be reasonably changed):



Note that you must use the visitor pattern for the interpreter as it is impossible to revisit nodes during a loop when using the listener pattern.

Beyond these constraints you are allowed to use any internal representation that you wish, as well as emit any code that you wish, as long as the output is correct.

SCalc has integer variables, conditionals, loops, prints, and various integer expressions.

RESERVED KEYWORDS

The following Keywords are reserved in *SCalc*:

- `if`
- `fi`
- `loop`
- `pool`
- `int`
- `print`

BOOLEANS

In this assignment, booleans can only be produced using comparison operators, there is no literal to express them. As well, they are ephemeral: there is no way to store them. They can only exist when created using one of the comparison operators.

When printed, booleans take on a value of 1 (true) or 0 (false). For example:

```
print(1 == 1);  
print(1 == 0);
```

produces the following output:

```
1  
0
```

As well booleans *are* usable in expressions and must be *upcast* to an integer. This means if a boolean is used in an arithmetic expression it takes on the integer value described above. For example:

```
print(1 + (1 == 1));  
print(1 + (1 == 0));
```

produces the following output:

```
2  
1
```


INTEGERS

In this assignment, integers are the *only* numerical type (there is no floating point type). As well, they are the only type that you can store.

In this assignment integer literals are defined as being a string that contains only the numerals 0-9 with no spaces.

Assertion: All integer literals will be ≥ 0 . (*nonnegative-literals*)

Examples of valid integers:

```
1
123
5234
01
10
```

Examples of invalid integers:

```
1.0
one
1_1
1o
```

As well, integers *are* usable in conditions and must be *downcast* to booleans. This means in a conditional, an integer that *is not* zero will be considered true and an integer that *is* zero will be considered false. For example:

```
if (999)
    print(999);
fi;
if (0)
    print(0);
fi;
```

produces the following output:

```
999
```


IDENTIFIERS

For the purpose of this assignment, identifiers are simple. They must start with an alphabetical character. This character may be followed by numbers or alphabetical characters. A keyword cannot be used as an identifier.

Examples of valid identifiers:

```
hello  
h  
h3llo  
Hi  
h3
```

Examples of invalid identifier:

```
3d  
a-bad-variable-name  
no@twitter  
we.don't.like.punctuation  
or_spelling
```


EXPRESSIONS

An expression is composed of integers, identifiers, and operators.

5.1 Operators

Operators are listed in descending precedence order. Operators without a horizontal line dividing them have equal precedence. For example, addition and subtraction have an equal level of precedence.

Class	Operation	Symbol	Usage	Associativity
Arithmetic	multiplication	*	<code>expr * expr</code>	left
	division	/	<code>expr / expr</code>	left
	addition	+	<code>expr + expr</code>	left
	subtraction	-	<code>expr - expr</code>	left
Comparison	less than	<	<code>expr < expr</code>	left
	greater than	>	<code>expr > expr</code>	left
	is equal	==	<code>expr == expr</code>	left
	is not equal	!=	<code>expr != expr</code>	left

Clarification: There is no remainder operator in SCalc. (*no-rem*)

Clarification: There is no exponentiation operator in SCalc. (*no-pow*)

Clarification: Division is integer division. (*int-div*)

5.2 Valid Expressions

Valid formats for expressions are

```
(<expr>
<expr> <op> <expr>
<int>
<id>
```

- `expr` is an expression.
- `int` is an integer.
- `id` is the identifier of a variable.

Assertion: All expressions will result in a value that fits in a 32 bit signed integer. (*expression-size*)

Assertion: No expression will contain a division by 0. (*zero-divide*)

Examples of valid expressions are

```
i * 2 * 10 + 4
2 - 4 * 5
```

5.3 Precedence

Precedence determines what order operations are evaluated in. Precedence works as defined in the following table:

Precedence	Operations
HIGHER	* /
	+ -
	< >
LOWER	== !=

STATEMENTS

In *SCalc* there are five types of statements:

- *Declaration*
- *Assignment*
- *Conditional*
- *Loop*
- *Print*

Each statement ends with a semicolon. White space is not important in *SCalc*.

Assertion: Whitespace is guaranteed to be a space, a tab, a carriage return, or a new line. (*simple-whitespace*)

6.1 Declaration

A variable declaration in *SCalc* has the following form:

```
int <id> = <expr>;
```

- `id` is the identifier of a variable.
- `expr` is an expression.

Variables have a few properties:

- cannot be used before being declared.
- cannot be declared without initialisation.
- cannot be declared more than once in an *SCalc* program.

Examples of valid declarations are:

```
int i = 9;  
int j = 9 * 4 + 10;  
int k = i * j;
```

Examples of invalid declarations are:

```
int i;  
int j =;
```

6.2 Assignment

Variable assignment is similar to variable declaration but it allows variables to be assigned new values. An assignment in *SExpr* has the following form:

```
<id> = <expr>;
```

- `id` is the identifier of an already declared variable.
- `expr` is an expression.

6.3 Conditional

A conditional in *SExpr* has the following form:

```
if (<expr>)  
  <statement-1>  
  <statement-2>  
  ...  
  <statement-n>  
fi;
```

- `expr` is an expression. The body of the `if` statement is executed if and only if this expression evaluates to a non-zero value.
- `statement-*` is any type of statement *except* a declaration. This means there can be assignments, nested loops, nested conditionals, and prints. There does not have to be any statements in the conditional.

Clarification: Declarations in conditionals can lead to undefined values due to global scoping. (*no-decl-cond*)

6.4 Loop

A loop in *SExpr* has the following form:

```
loop (expr)  
  <statement-1>  
  <statement-2>  
  ...  
  <statement-n>  
pool;
```

- `expr` is an expression. The body of the `loop` statement is repeatedly evaluated as long as this expression is non-zero. The expression is evaluated prior to running the body similar to a *C* `while` loop.
- `statement-*` is any type of statement *except* a declaration. This means there can be assignments, nested loops, nested conditionals, and prints. There does not have to be any statements in the loop, but without side effects a loop will be infinite (unless it is never entered).

Clarification: Declarations in loops can lead to undefined or repeatedly defined values due to global scoping. (*no-decl-loop*)

6.5 Print

Print statements print the integer value of an expression followed by a newline. A print statement in *S*Calc has the following form:

```
print(<expr>);
```

- `expr` is an expression.

For example, the input:

```
int i = 0;  
loop (i < 5)  
  print(i);  
  i = i + 1;  
pool;
```

should print:

```
0  
1  
2  
3  
4
```


COMMENTS

SCalc supports a subset of *C99* comments.

Single line comments are made using `//`. Anything on the line after the two adjacent forward slashes is ignored. For example:

```
// A comment on its own line  
print(7); // This is ignored
```


BACKENDS

SCalc. This compiler will directly generate code for the following three backends:

- *x86* assembly
- *RISC-V* assembly
- *ARM* assembly

You must also create an *Interpreter* for *SCalc*

8.1 RISC-V

We recommend that you implement variables in the `.data` segment of your assembly code using `.word` entries. The general syntax for RISC-V assembly is as follows:

```
.data
_newline: .asciz "\n"
var1: .word 0
var2: .word 0
...

.text
main:
    <your generated code goes here>
    li    a7, 10
    ecall
```

This reference is the *RISC-V Assembler and Runtime Simulator (RARS)* <<https://github.com/TheThirdOne/rars>>, which includes a list of available syscalls. To print integers you should use the `print int` `ecall` (1). To print strings, you should use the `print string` `ecall` (4) in combination with the address of a string you've defined in the `.data` section containing only a new line character (see above).

If you save the *RISC-V* output as `program.s` then on the CS undergrad machines you can run it with the command:

```
/usr/local/bin/rars program.s
```

If you wish to debug you may also launch RARS with the command:

```
/usr/local/bin/rars
```

which opens a graphical IDE that you can load your assembler code into.

8.2 x86

You should use the Intel syntax for *x86*, and your compiler's output must work with the *nasm* assembler. Your *x86* output should look something like this:

```
global main
extern printf

section .data
var1: DD 0
var2: DD 0
...

section .text
main:
    <your generated code goes here>
    mov eax, 0 ; Set return value
    ret      ; Return
```

Print will use `printf`, which we will link in later to make it easier to implement the `print` statement. Printing consists of three steps:

1. Push the arguments onto the stack in reverse order.
2. Call `printf`.
3. Clean up the stack.

For instance, the following segment of code contains a call to `printf`:

```
global main
extern printf
section .data
_format_string: DB "Hello! Here is a number: %d",0xA,0x0

section .text
main:
    push dword 7      ; Second argument
    push _format_string ; First argument (remember stacks are FILO)
    call printf      ; Make the call to printf
    add esp, 8        ; Pop the stack, we are done!

    mov eax, 0
    ret
```

`_format_string` deserves a quick explanation. The `DB` declares a datatype of bytes, which is appropriate for characters. The string is converted to its character components and placed at the label. The values in commas after it are appended to the array. `0xA` is the [ASCII value of a newline in hexadecimal](#). The `0x0` is the [null terminator](#) for the string.

You won't need to know more of the *x86* calling conventions than what was demonstrated above.

If you save the *x86* output as `program.s` you can assemble an executable and run it by executing the following commands:

```
nasm -felf -o program.o program.s
gcc -m32 -o program program.o
./program
```

Try this on the `printf` example and make sure that it works!

8.3 ARM

The *ARM* assembly output should look something like this:

```
.arch armv7-a
.data
_format_string: .asciz "%d\n"
var1: .word 0
var2: .word 0
...

.text
.globl main
main:
    push {ip, lr}      // Save link and scratch registers.

    <your generated code goes here>

    pop {ip, lr}       // Load link and scratch registers.
    mov r0, #0         // Set return value.
    bx lr              // Return.
```

We will also be using `printf` with *ARM*. The *ARM* calling convention is different from *x86*: the first argument is passed in `r0`, and the second argument is passed in `r1`. The following code demonstrates a call to `printf` in *ARM* assembly:

```
.arch armv7-a
.data
_format_string: .asciz "Hello! Here is a number: %d\n"

.text
.globl main
main:
    push {ip, lr}      // Save link and scratch registers.

    ldr r0, =_format_string // Load the address of the format string into the first
    ↪ argument.
    mov r1, #7         // Place the literal 7 into the second argument.
    bl printf          // Call printf.

    pop {ip, lr}       // Load link and scratch registers.
    mov r0, #0         // Set return value.
    bx lr              // Return.
```

Aside from the difference in calling convention, this code is very similar to the *x86* example. As well, declaring the `_format_string` is a lot easier because it has a null-terminated string directive and can parse `\n` like *RISC-V*.

ARMv7-A lacks a division instruction. Therefore, we have to call the subroutine `__aeabi_idiv` to perform integer division. The following code demonstrates a call to `__aeabi_idiv` in *ARM* assembly:

```
.arch armv7-a
.data

.text
.globl main
main:
    push {ip, lr}      // Save link and scratch registers.

    mov r0, #5          // Move the literal 5 into the first argument (the dividend).
    mov r1, #3          // Move the literal 3 into the second argument (the divisor).
    bl __aeabi_idiv(PLT) // Divide 5 by 3, return the result in r0.

    pop {ip, lr}       // Load link and scratch registers.
    mov r0, #0         // Set return value.
    bx  lr             // Return.
```

In order to assemble and run an executable you may run the following commands:

```
arm-none-eabi-as -o program.o program.s
arm-none-eabi-gcc -specs=rdimon.specs -o program program.o
qemu-arm ./program
```

Try this on the `printf` example and make sure that it works!

8.4 Interpreter

You should be able to execute a program without compiling by implementing an interpreter. This should work similarly to the generator assignment.

INPUT

The input processed by your compiler will be in a file specified on the command line. You will also receive a value signifying which mode your compiler should be run in. The command to run your compiler will take this form:

```
scalc <mode> <input_file_path> <output_file_path>
```

Mode can be one of these four values:

- riscv
- x86
- arm
- interpreter

You should open the file `input_file_path` and parse it. The input file will be a valid `scalc` file.

OUTPUT

Output is to be written to a file specified on the command line. Your compiler will be invoked with the following command:

```
scalc <mode> <input_file_path> <output_file_path>
```

You should open the file `output_file_path` and write to it. The output file should be overwritten if it already exists.

Output content is standardized to ensure everyone can pass everyone's tests. Follow these specifications:

- Each number printed should be on its own line followed by a new line.
- There *must* be a new line after each `print` statement's printed value.
- There *must not* be any trailing space after the final number and before the newline.
- There *must* be an empty line at the end of your output.

Clarification: Empty input should result in empty output. (*empty-input*)

ASSERTIONS

ALL input test cases will be valid. It can be a good idea to do error checking for your own testing and debugging, but it is *not necessary*. If you encounter what you think is undefined behaviour or think something is ambiguous then *do* make a forum post about it to clarify.

What does it mean to be valid input? The input must adhere to the specification. The rules below give more in-depth explanation of specification particulars.

1. **undef-behaviour:**

A test case *will not* take advantage of undefined behaviour. Undefined behaviour is functionality that does not have an outcome described explicitly by this specification.

2. **nonnegative-literals:**

All integer literals will be ≥ 0 . For example, the following tests would be considered invalid:

```
int i = -1;
```

3. **literal-size:**

All integer literals will fit in 31 unsigned bits. This means an integer literal can be anywhere in the range $[0, 2^{31} - 1]$ or $[0, 2147483647]$. For example, the following tests would be considered invalid:

```
int i = 2147483648;
```

4. **expression-size:**

All expressions will result in a value that will fit in 32 signed bits. This means the result of an expression can be anywhere in the range $[-2^{31}, 2^{31} - 1]$ or $[-2147483648, 2147483647]$. Any operation that results in underflow or overflow will render the input invalid. For example, the following tests would be considered invalid:

```
int i = 2147483647 + 1;
int j = 0 - 2147483647 - 2;
```

5. **zero-divide:**

No expression will contain a division by 0. The result of a division by zero is indeterminate so we will not handle it. For example, the following tests would be considered invalid:

```
int i = 1 / 0;
int j = 1 / (1 - 1);
```

6. **simple-whitespace:**

Whitespace is guaranteed to be a space, a tab, a carriage return, or a new line. Any other whitespace characters will render the input invalid. The following ANTLR rule will ensure you adhere to this:

```
WS: [ \t\r\n]+ -> skip;
```

CLARIFICATIONS

These clarifications are meant to add more information to the specification without cluttering it.

1. **no-rem:**

There is no remainder operator in SCalc. For example, the following tests would be considered invalid:

```
int i = 5 % 2;
```

2. **no-pow:**

There is no exponentiation operator in SCalc. For example, the following tests would be considered invalid:

```
int i = 2 ^ 2;
```

3. **int-div:**

Division is integer division. This means that any decimal portion of a division operation result is truncated (not rounded). No extra work is required: this is the default in C++, RISC-V, ARM, and X86. For example:

```
print(5 / 3);  
print((-5) / 3);
```

produces the following output:

```
1  
-1
```

4. **no-decl-cond:**

Declarations in conditionals can lead to undefined values due to global scoping. Because of the potentially conditional nature of the execution, it is possible to violate the property of variables stating that *variables must be defined before being used* (not just declared) by never executing the definition. For example, the following test would break this property and is therefore invalid:

```
if (1 < 0)  
    int i = 0;  
fi  
int j = i;
```

5. **no-decl-loop:**

Declarations in loops can lead to undefined or repeatedly defined values due to global scoping. Because of the potentially conditional nature of the execution, it is possible to violate the property of variables stating that *variables must be defined before being used* (not just declared) by never executing the definition. For example, the following test would break this property and is therefore invalid:

```
loop (1 < 0)
  int i = 0;
pool
int j = i;
```

As well, because of the potentially repeated nature of the execution, it is possible to violate the property of variables stating that *variables can only be defined once* by repeating the declaration. For example, the following test would break this property and is therefore invalid:

```
int i = 0;
loop (i < 2)
  int j = 0;
  i = i + 1;
pool;
```

6. **empty-input:**

Empty input should result in empty output. This is in keeping with all of the output rules defined. There are no `print` statements so there would be no numbers, newlines or output of any kind. All that you are left with is a single empty line, which matches “*should* be an empty line at the end of your output”.

DELIVERABLES

Your submission will be **the latest commit before the deadline** to your github repository. Your submission will be automatically snapshotted by the GitHub classroom at the submission time.

Do not submit your binaries, they will be built just before being tested. The solutions will be built using the lab machines. You should make sure your solution builds in a lab environment prior to the submission time.

Your tests also should be committed to your github repository. We will pull both your submission and tests directly from your repository.

You do not need to submit anything on eclass or anywhere else.

TIPS AND HINTS

- Review the [Tips and Hints](#) from the generator assignment: much of it applies to this assignment as well. In particular, the style and design sections are necessary.
- Write tests *before* you implement the things they will test. The testing script provided is designed to handle failed test cases. You can reduce output in the testing tool by passing the `-q` flag.
- As with the generator assignment, the ANTLR visitor pattern is best for implementing the interpreter.
- The RISC-V, ARM, and x86 compilers can be built using either the visitor or the listener pattern. The listener may be more appropriate so it is the suggested method.
- A single listener or visitor should be able to handle the x86, RISC-V, and ARM code generation. All that should change is the templated strings.
- In addition to the [previous tips](#), here's how to stay stylish in this assignment:
 - There is no specified syntax guide for use with *inja*, thus you **could** write your strings in any style that you choose. I suggest you use [implicit string concatenation](#) to delineate your strings. For example, this may be a format string for literal division:

```
std::string literalDivision =
    "\n # Div.\n"
    "  addi  t0, zero, {{ dividend }} # Set dividend.\n"
    "  addi  t1, zero, {{ divisor }}  # Set divisor.\n"
    "  div   t2, t0, t1               # Divide.\n"
```

- This assignment will be extended to build a calculator that handles vectors in the next assignment. For that assignment you will need to do type checking. Therefore you are advised to:
 - Read that assignment now.
 - Include type checking in this solution even though you are only required to handle integers in this assignment.
- You are allowed to use RISC-V pseudo-instructions.

USING INJA

Inja is the string templating library you'll be using in this assignment. You'll only need the most basic string replacement functionality, even though it can [be much more advanced](#).

String templating looks like this:

```
#include "nlohmann/json.hpp"
#include "inja.hpp"

#include <string>
#include <iostream>

using JSON = nlohmann::json;

int main() {
    std::string name;
    int age;
    std::cout << "What's your name?\n";
    std::cin >> name;
    std::cout << "How old are you?\n";
    std::cin >> age;

    JSON data;
    data["name"] = name;
    data["age"] = age;

    std::string strTemplate =
        "Hi, {{ name }}!\n"
        "You are {{ age }} years old!\n";

    std::cout << inja::render(strTemplate, data);

    return 0;
}
```

Let's deconstruct the example. First we have the includes that give us access to the `json` class that *inja* uses to store data as well as the *inja* tools.

```
#include "nlohmann/json.hpp"
#include "inja.hpp"
```

Next we have a statement that's purely for convenience purposes. It aliases the `json` class to `JSON`. If we didn't use this we could refer to the `json` class using the fully qualified name `nlohmann::json`.

```
using JSON = nlohmann::json;
```

Now we create our data object and fill it. If you have keys `key1` and `key2` that map to values `value1` and `value2` respectively, the general syntax is:

```
JSON data;  
data["key1"] = value1;  
data["key2"] = value2;  
...
```

Almost done. You need to define your format string using the inja syntax. To have the value replace the key in the string you need to surround the key double braces like so: `{{ keyname }}`.

```
std::string strTemplate = "{{ key1 }} {{ key2 }}";
```

Finally, you need to render the data into the template:

```
std::string result = inja::render(strTemplate, data);
```