Language Design

The language will be defined in a bottom-up approach, starting with data types, going over how operators and values can be used in expressions, to finally defining procedures and the global campaign structure.

There are four data types in HCCDL. Two of which are elementary, the other two are recursive data types. There are numerical values, which can be expressed in the decimal format by writing a sequence of digits (`12`, `0`, `25746523`, `01`). Leading zeros are allowed, and there is no limit on the size. The binary representation of numerical values starts with `0b`, followed by a sequence of zeros and ones (`0b0010`). Similarly, a hexadecimal number is defined by `0x`, followed by a sequence of hexadecimal digits (`0xff00f8`). The other elementary type is string: A possibly empty sequence of non-quotation mark-characters, enclosed by quotation marks (`"this is a string"`, `"[{}+"`, `""`). Recursive data types contain other elementary or recursive data types. Key-value pairs consist of a string as the key and a value of any other data type. Key-value pairs are created using the `->` operator. On the left side, there has to be a string, and on the right side, a value (`"key1" -> 2`, `"key2" -> "a value"`, `"key3" -> ("nested" -> 7)`). Finally, the last data type is the list. Lists contain zero or more values of any data type and do not have to be homogeneous. Brackets are used to describe lists (`[]`, `[1, 2, 3]`, `["mix", 0b10, "k" -> "v", ["nested", "list"]]`).

Variables can be used to store values. The language is loosely typed, so no data type has to be defined for variables. In fact, a single variable can store, e.g., a string at first, and later get assigned a numeric value. Assignments are performed using the `=` operator (`var1 = 1`, `var2 = [1, 2]`). When a variable name is used without being on the left side of `=`, it evaluates to its stored value.

There are several more operators, which can be used to modify and combine values. A numbers sign can be changed using the unary `-` (`-12`, `-0xf8`). It is possible to use the unary `+`, which has no effect, though. It can be used for explicitness. The binary operators `+`, `-`, `*`, `/`, and `%` can be used with numbers, performing their conventional operation. `+` is also applicable to merge lists (`[1, 2] + [3, 4]`), and to add elements to lists (`[1, 2] + 3`, `0 + [1, 2]`). With two strings as parameters, `+` will evaluate to their concatenation. The key, as well as the value, can be extracted from a key-value pair using `.key` and `.val`, respectively (`("a" -> 1).key`, `("a" -> 1).val`). List elements can be retrieved from a list by denoting the index in brackets after the list (`["one", "two", "there"][1]`). The first element has index 0, so the example would evaluate to `"two"`. Operators differ in their precedence, as defined in Table 4.1.

Precedence	Operators
1	`[]` (list access)
2	`.key`, `.val`
3	`+`, `-` (unary)
4	`*`, `/`, `%`
5	`+`, `-` (binary)
6	`->`
7	`=`
8	`[]` (list creation)

Table 4.1.: HCCDL Operator Precedence

All of the elementary values and operators mentioned up until now are considered expressions. Even a variable assignment is an expression; it evaluates to the value assigned.

Expressions can be nested arbitrarily deeply into each other, as long as the types match the operators' constraints. To fulfill the requirement of imperativeness, statements are introduced. An expression followed by a semicolon is a statement (`3;` (useless statement), `x = "k" -> ([1,2,3] + [4, 5] + 6)[4 + (-2)];`). Another statement is the for loop. It takes a list of values, a variable name, and a statement. For every element in the list, the element is assigned to the variable name, and the statement is executed. To not only be able to execute a single expression in a loop, a sequence of statements enclosed by braces is also a statement. An example is given in Listing 4.4.

Listing 4.4: HCCDL For Loop Example

```
list = [1, 2, 3, 4, 5, 6]
doubled = []

for (i : list) {
    double_i = 2 * list[i];
    doubled = doubled + double_i;
}
```

The requirements demand the existence of "user-defined instructions", similar to functions in the C language. Thus, in HCCDL, the concept of procedures exists. They can take a number of parameters, and consist of a sequence of statements. The parameters basically act like pre-initialized variables. For uniformity reasons, HCCDL follows the procedural programming paradigm and requires that statements can exist only inside of procedures. Invoking other procedures is an expression, as well. In the case of user-defined procedures, a procedure call will evaluate to the evaluation result of the last executed expression in the called procedure. Listing 4.5 shows how procedures are defined and called.

Listing 4.5: HCCDL Procedure Example

```
proc do_something(parameter1, parameter2) {
    ["para1" -> parameter1, "para2" -> parameter2]
}

proc caller() {
    first = do_something(1, 2);
    second = do_something("hello", "world");
}
```

The `do_something` procedure wraps its parameters in a list of key-value pairs. Because this creation of the list is the last expression in this procedure, it is also what a call of this procedure will evaluate to. Thus, at the end of the `caller` procedure, `first` will have the value `[1, 2]`, and `second` will have the value `["hello", "world"]`. The compiler will start the campaign by manually calling the procedure with the name `main`, which has to exist.

As of now, we have defined a language capable of performing calculations and creating data structures, but there has been no mention of how to issue hypercalls and delays. Those are provided as built-in procedures. They are called the same way as user-defined ones but are handled differently by the compiler. Following built-ins are available:

- `delay(duration)`: Requests a delay of length duration. Duration has to be a numeric value. Because procedure calls are expressions, a delay call has to evaluate to some value. This call is only relevant due to its side-effect of requesting a delay; thus, it returns a special value `None`, with which nothing can be done.

- `hcall(params)`: Requests a hypercall. `params` has to be a list of key-value pairs, which describe the hypercall to be performed. Which entries are expected depends on the active campaign listener implementation, and is not part of the generic language. `hcall` also evaluates to `None`.
- `integerBounds(bitcount)`: Evaluates to a list containing the values 0, 1, the signed maximum, as well as the unsigned maximum for integers of the size `bitcount`.
- `randExp(scale)`: Evaluates to a random number, following a negative-exponential distribution with scaling factor `scale`. Beware that such random values are usually floating-point values. This procedure returns random samples rounded down to the nearest integer. `scale` has to be a numerical value.
- `randomUniform(bitcount)`: Evaluates to a random integer lying in the interval [0, $2^{bitcount}-1$]. According to the uniform distribution, all possible outcomes are equally likely. `bitcount` has to be a numerical value.
- `range(lower, upper)`: Evaluates to a list of integers of the format [`lower`, `lower`+1, `lower`+2 ..., `upper-1`]. Thus, the lower bound is inclusive, and the upper bound is exclusive. Both parameters have to be numeric values.
- `rangeStep(lower, step, upper)`: Evaluates to a list of integers of the format [`lower`, `lower` + `step`, `lower` + 2*`step`, ..., `upper`-1], if `lower` + n*`step` = `upper`-1, otherwise the next smaller number in the sequence is the last element of the list. All three parameters have to be numeric values.
- `signedMax(bitcount)`: Evaluates to the signed maximum of an integer with size `bitcount`, using the two's complement representation. This equals $2^{bitcount-1}-1$. `bitcount` has to be a numerical value.
- `unsignedMax(bitcount)`: Evaluates to the unsigned maximum of an integer with size `bitcount`. This is $2^{bitcount}-1$. `bitcount` has to be a numerical value.

While procedures can exchange values by passing parameters, this might cause textual overhead for values that are needed in many places. Therefore, global variables are introduced. They can be either uninitialized or initialized with a numeric value. To allow for more complex initializations, the user has to possibility to write an `init` procedure, which, if it exists, is executed by the compiler before the main procedure. Listing 4.6 demonstrates the usage of global variables.

Listing 4.6: HCCDL Global Variable Example

```
uninit1, uninit2;
init_jm = 396;

proc init() {
    uninit1 = [1, 2, 3];
    uninit2 = ["a" -> init_jm, "b" -> uninit1];
}

proc main() {
    x = uninit2;
    delay(init_jm);
}
```

A hypercall campaign can be composed of uninitialized and initialized global variables, an `init` and a `main` procedure, and other user-defined procedures. The `main` procedure is the only mandatory component.

There is an additional remark to specify, which did not fit into the flow of the bottom-up explanation. Identifiers, which are the names of variables, parameters, and procedures, are allowed to consist of lower- and upper-case letters, digits, and underscores. However, the first character cannot be a number.

This concludes the definition of the HCCDL - Hypercall Campaign Description Language. The implementation of a compiler that is able to parse the language, evaluate the campaign, and deliver hypercall and delay requests to a hypervisor-specific listener is described in Section 5.1. The rest of this chapter is dedicated to explaining how Hyper-V-specific components for the framework are designed.