University of Ljubljana
Faculty of Computer and Information Science

Ambrož Bizjak

The NCD Programming Language

diploma thesis

on the interdisciplinary university study program

Supervisor: prof. dr. Borut Robič

Ljubljana, 2013

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Ambrož Bizjak

Programski jezik NCD

diplomsko delo
na interdisciplinarnem univerzitetnem študiju

Mentor: prof. dr. Borut Robič

Ljubljana, 2013

This thesis was set in LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00048/2013
Datum: 04.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko ter Fakulteta za matematiko in fiziko izdaja naslednjo nalogo:

Kandidat: **AMBROŽ BIZJAK**

Naslov: **PROGRAMSKI JEZIK NCD**

THE NCD PROGRAMMING LANGUAGE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Podrobno opišite progamski jezik NCD, ki je namenjen programskemu nastavljanju omrežja v operacijskem sistemu Linux. Posebnost tega jezika je, da med izvajanjem programa vsak ukaz asinhrono povzroči sestopanje programa do tega ukaza. Pokažite, kako je z ustreznimi razširitvami tega osnovnega načina izvajanja in potrebnimi vgrajenimi ukazi jezik primeren tudi za bolj splošno uporabo.

Mentor:

prof. dr. Borut Robič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan Fakultete za računalništvo in informatiko:

prof. dr. Nikolaj Zimic

Dekan Fakultete za matematiko in fiziko:

akad. prof. dr. Franc Forstnerič

UNIVERZA V LJUBLJANI • Fakulteta za matematiko in fiziko • 1

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Ambrož Bizjak, z vpisno številko **63080085**, sem avtor diplomskega dela z naslovom:

Programski jezik NCD

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom prof. dr. Boruta Robiča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 21. junija 2013 Podpis avtorja:

Contents

List of acronyms

NCD Network Configuration Daemon

DHCP Dynamic Host Configuration Protocol

IP Internet Protocol

API Application Programming Interface

IPC Inter-Process Communication

Povzetek

V diplomskem delu opisujem programski jezik *NCD*[1]. Najpomembnejša zmogljivost tega jezika je posebna vrsta *sestopanja*, zaradi katere je v kombinaciji z asinhronim izvajanjem in dogodkovno implementacijo jezik posebej primeren za programsko nastavitev omrežja. Na začetku razvoja je bil jezik namreč namenjen izključno nastavitvi omrežja v operacijskem sistemu Linux. Vendar pa je bil od prvotne verzije jezik večkrat razširjen, tako da je v trenutnem stanju bistveno bolj uporaben, ne samo za nastavitev omrežja, temveč tudi za mnogo drugih vrst problemov.

Osnova programskega jezika NCD je nov *model izvajanja*, ki razširi model imperativnega programiranja. V enostavni varianti tega novega modela je program sestavljen iz množice tako imenovanih *procesov*. Vsak proces je definiran s seznamom *ukazov*; vsakemu ukazu je potrebno podati seznam argumenov, lahko pa mu tudi dodelimo identifikacijsko ime, s katerim se lahko nanj sklicujemo pri uporabi kasnejših ukazov. Ukazi znotraj procesa se izvajajo v zaporedju, od zgoraj navzdol. Ko se vsi ukazi znotraj procesa uspešno izvedejo, se proces ne zaključi, temveč čaka. Če je v programu prisotnih več procesov, se njihovo izvajanje prepleta. Sledi primer zelo enostavnega programa v programskem jeziku NCD, ki ob zagonu v terminal izpiše besedilo “Hello, world!”, nato pa se ne zaključi, ampak čaka. Deterministično razvrščanje procesov znotraj jezika zagotovi, da je rezultat tega programa vedno izpis “Hello, world!”, ne pa kaj drugega.

```
process world {
    println(" world!");
}
process hello {
    print("Hello, ");
```

```
}
```

Jezik NCD ne loči med deklaracijami spremenljivk in ukazi - vsak ukaz se namreč obnaša kot spremenljivka, če mu le dodelimo identifikacijsko ime. Tako lahko na primer uporabimo ukaz *var*, ki ni poseben sintaktični konstrukt, da shranimo poljubno vrednost. Naslednji program prikaže uporabo ukaza *var*. Ukazi tudi po izvedbi ostanejo aktivni v ozadju; iz tega razloga je namreč možno prebrati vrednost, ki jo hrani ukaz *var*, tudi po tem, ko se le-ta zaključi.

```
process helloworld {
    var("Hello, world!") x;
    println(x);
}
```

Vsak ukaz lahko po tem, ko se zaključi, naknadno *zahteva sestopanje* procesa. Ko ukaz zahteva sestopanje, se proces samostojno vrne v stanje preden se je ta ukaz zaključil; ko se ukaz ponovno zaključi, se izvajanje ukazov nadaljuje od tega ukaza naprej. Naslednji program prikazuje izgradnjo neskončne zanke z uporabo ukaza *backtrack_point* in sorodnega metodnega ukaza *go*, ki delujta na osnovi sestopanja. Program izvaja neskončno zanko, ker uporaba ukaza *go* povzroči, da ukaz *backtrack_point* zahteva sestopanje in se nenudoma ponovno zaključi.

```
process infinite {
    backtrack_point() point;
    println("Hello!");
    point->go();
}
```

V zgornjem programu smo sestopanje neposredno sprožili z ukazom. Vendar pa je uporabnost sestopanja v jeziku veliko večja kot morda izgleda, saj lahko ukazi sestopanje zahtevajo *asinhrono*. Implementacija jezika NCD je namreč osnovana na dogodkovni zanki; to ukazom omogoča, da se odzivajo na dogodke tudi po tem, ko so se zaključili. Na tej točki se bomo z ukazom *net.backend.waitdevice* končno približali temi nastavitve omrežja; naslednji program namreč opazuje prisotnost mrežnega vmesnika in izpiše vsako spremembo stanja.

```
process watcher {
    net.backend.waitdevice("eth0");
    println("Present.");
    rprintln("Not present.");
}
```

Ukaz *net.backend.waitdevice* čaka in se zaključi le, ko je mrežni vmesnik prisoten. Takrat se izvajanje ukazov v procesu nadaljuje, vendar *net.backend.waitdevice* v ozadju še vedno opazuje stanje mrežnega vmesnika. Če naknadno opazi, da mrežni vmesnik ni več prisoten, zahteva sestopanje procesa in ponovno čaka. V programu smo uporabili še en nov ukaz, namreč *rprintln*. Ta ukaz, prav tako kot *println*, izpiše besedilo v terminal - vendar ne takrat, ko se začne izvajati, ampak takrat, ko proces sestopa preko tega ukaza. Posledica tega obnašanja je, da program vsakič, ko preneha prisotnost mrežnega vmesnika, izpiše besedilo "Not present.". Sestopanje poteka od spodaj navzgor; če bi v program za obstoječi ukaz *rprintln* dodali še enega, bi se v primeru sestopanja najprej izpisalo besedilo spodnjega ukaza.

Z uporabo nekaj dodatnih ukazov lahko dosežemo samostojno nastavitev IP naslova žičnega mrežnega vmesnika. Potem, ko smo počakali, da je mrežni vmesnik spet prisoten, ga bomo vklopili, počakali, da je mrežni kabel priključen, ter šele nato vmesniku dodelili IP naslov.

```
process lan {
    var("eth0") dev;
    net.backend.waitdevice(dev);
    net.up(dev);
    net.backend.waitlink(dev);
    net.ipv4.addr(dev, "192.168.123.4/24");
}
```

Ukaz *net.backend.waitlink* deluje na isti način kot *net.backend.waitdevice*, le da opazuje stanje povezave mrežnega vmesnika, kar je v primeru žičnih vmesnikov ekvivalentno pravilnemu priklopu omrežnega kabla. Po drugi strani pa ukaza *net.up* in *net.ipv4.addr* ne čakata na dogodke, ampak izvajata akcije. Ukaz *net.up*

namreč ob zagonu vklopi mrežni vmesnik, ob sestopanju pa ga izklopi. Podobno ukaz *net.ipv4.addr* ob zagonu dodeli IP naslov, ob sestopanju pa ga odstrani.

Opisana semantika zagotovi, da se IP naslov samostojno odstrani, ko izključimo mrežni kabel, in se ponovno dodeli šele, ko kabel naslednjič priključimo. V primeru, da želimo celotni program zaključiti, pa bo ta, preden se zaključi, samostojno odstranil IP naslov in izklopil mrežni vmesnik, če je to potrebno. Namreč, ko program v jeziku NCD od operacijskega sistema prejme zahtevo po zaključitvi, ta začne sestopati vse procese in se zaključi šele, ko je sestopanje končano.

Jezik NCD sicer nudi tudi nekatere konstrutke iz imperativnega programiranja, vendar pa so ti implementirani na nekoliko višjem nivoju kot običajno, tako da so dostopni le preko ukazov ali morda preko sintaktičih sladkorjev. Na primer, klici funkcij potekajo preko ukaza *call*, kateremu je treba podati ime tako imenovanega *vzorca za proces*; ukaz *call* namreč kar med izvajanjem programa ustvari nov proces na osnovi vzorca. Naslednji program prikazuje, kako v jeziku NCD izgledajo klici funkcij in pogojno izvajanje; program z uporabo rekurzivnih klicev v padajočem vrstnem redu izpiše naravna števila od 10 do 1.

```
process main {
    call("count_down", {"10"});
}
template count_down {
    num_greater(_arg0, "0") greater;
    If (greater) {
        println(_arg0);
        num_subtract(_arg0, "1") new_count;
        call("count_down", {new_count});
    };
}
```

Ključne besede: programski jezik, sestopanje, vračanje, dogodkovni, asinhroni, nastavitev omrežja

Abstract

We have developed a new programming language, called NCD. The language is asynchronous and event-driven by design, meaning that multiple pieces of code can execute concurrently, while still bound to a single kernel thread. This asynchronous nature, together with the language's defining feature, *statement-triggered backtracking*, allows for elegant solutions to many programming problems which require a continuously running program that is responding to various kinds of events. It is especially suitable for problems where certain events are expected to automatically roll-back the execution to an earlier state and continue from there when appropriate. The language was originally intended to be used solely for expressing the network configuration of Linux systems in a simple and portable, but powerful and extensible, manner. However, as the potential of its features has been recognized, the language has been extended multiple times to make it more useful for solving a broader range of problems.

Keywords: programming language, backtracking, roll-back, event-driven, asynchronous, network configuration

Chapter 1

Introduction

1.1 Motivation

My original motivation for the development of the NCD programming language[1] was to describe the network configuration of a Linux system in a manner that I would find practical as a computer programmer. The primary quality I was looking for was the ability to modify the behavior of the network-configuration system with ease, especially to implement behaviors which the developers of the system in question have not foreseen or have deemed unnecessary or too specific. I have determined that modifying any existing network configuration-system to implement various features I need to be unacceptable, for multiple reasons.

- Existing network configuration systems[2, 3, 4] are not designed with the requirements which I had in mind, in particular the ease of modifying their behavior. They are designed with a certain model in mind, and new features can only be easily added if they fit into this model[5].

- Modifying an existing system would require significant effort. As an initial barrier, a lot of time would have to be spent in reading the source code to understand the system well enough be able to modify it correctly. However, a much greater problem is the likely possibility that the entire system would have to be re-engineered to provide enough flexibility to modify its behavior. In this case, little of the original system might remain intact.

- There would be an increasing disparity between my modified version of the software and the version provided by the original developers, due to changes on both sides. The developers of the original software may refuse to integrate some of the modifications, or extra development would be needed to make the modifications appeal to them, possibly reducing their quality in the process. As a result, maintaining a modified version of the likely very complex software would get more and more difficult with time.

At the very beginning when I started to develop my new network-configuration software, it was a small program that would parse a configuration file specifying the IP configuration for each network interface. It could only work with wired network interfaces, and it could only use a static IP configuration or DHCP. It was at this stage that I realized how the software could be transformed into a much more powerful and generic system, based on a model especially suitable for expressing network configurations[6].

The model will be explained using an example task of configuring a wired network interface by using the DHCP protocol to obtain an IP address. In this model, a configuration task is defined by a *sequence of steps* to be performed in order. In our case, the steps for the configuration of a network interface are these:

1. Wait until the network interface *eth0* exists.
2. Turn on the network interface *eth0*.
3. Wait until the link layer[7] has stated operating for the network interface *eth0*.
4. Use the DHCP protocol to reserve an IP address for the network interface *eth0*.
5. Assign the IP address which was *obtained by DHCP above* to the network interface *eth0*.

Emphasis is put on parts that can be considered input parameters for a step, to make it easier to see how these could be communicated to a computer. It may seem that these steps are all that a computer needs to configure a network interface. However, that would only work when *nothing goes wrong*. This means that, for example:

- After the network interface has started existing, it would never stop existing.
- After the link layer has started operating, it would never stop operating.
- After an IP address has been reserved via DHCP, the lease would never expire.
- There would never be any reason why the entire interface would have to be deconfigured and turned off.

However, in real the world these problem events need to be handled. The primary idea of the model is to allow the configuration system to automatically handle the problem events, given only an apparently imperative sequence of steps (in a machine-readable format).

1.2 Execution model

To see how the problem events could be handled automatically, we need to determine the appropriate responses, and find a common pattern. The responses to the problem events for our example could be as follows:

- If the network interface stops existing (possibly because it was pulled out of the computer), the appropriate reaction is to remove the IP addresses from the interface (if it has been assigned), turn the network interface off (if we turned it on), and then proceed to wait until the network interface begins to exist once again. Here we are ignorant of the possibility that the IP address could already have been removed by the operating system, and it may not be possible to turn off a nonexistent network interface.
- If the link layer stops operating (possibly because the network cable was pulled out), the IP address needs to be removed (if assigned) and the program has to wait until the link layer begins operating once again.
- If the DHCP lease for the reserved IP address times out, then the computer is no longer allowed to use this IP address, so the IP address needs to be removed from the interface, and the program should keep trying to get another DHCP lease (but it must not turn the interface off).

- If some event happens that requires a complete deconfiguration of the network interface (such as the action of a network administrator or the system being shut down), then all the deconfiguration actions need to be performed, like in the first case when the network interface stops existing. After that, the program does not need to wait for any events related to the interface, such as its existence or link state.

The appropriate reactions to problems described above are very similar in nature - they all involve performing, in a reverse order, *deinitialization actions* for the steps following the problematic step (or for all steps in the special last case), where the deinitialization actions depend *only* on the state of their corresponding configuration steps, and not on the state of other steps. For instance, if the IP address is assigned, and either the DHCP lease is lost or the link layer stops operating, the IP address just needs to be removed, no matter which of the two events actually happened. There may appear to be some exceptions, such as trying to remove an IP address from an interface after the interface no longer exists, but this can be solved by defining and implementing the deinitialization actions such that they will have no ill effects if a resource they are to act upon no longer exists. Put simply, our model assumes that the deinitialization actions of the steps are implicitly defined by the step itself - they need not be specified in the sequence of operations steps that define a configuration task. For example, if the step is "turn on the network interface", it is implied that its deinitialization action is to turn off the network interface, and this action will be performed automatically when something goes wrong in the configuration above this step. In the context of the NCD language, we will call this unique behavior of automatically performing deinitialization actions *backtracking*.

Chapter 2

Core language

2.1 Process syntax

If a computer is to understand the sequence of steps, a concrete syntax needs to be defined. To this end, we introduce the concept of an *NCD process*, which corresponds to a *configuration task* in the above informal description of the model. An NCD process consists of a sequence of *statements*, that we have previously called steps. With no further ado, we present our example network-interface configuration task as a complete NCD program in Listing 2.1, which will indeed run using the current NCD software.

Listing 2.1

```
# This is a comment. The code below specifies our process.
# The process needs to be given a unique name (eth0_interface).
process eth0_interface {
    net.backend.waitdevice("eth0");
    net.up("eth0");
    net.backend.waitlink("eth0");
    net.ipv4.dhcp("eth0") dhcp_id;
    net.ipv4.addr("eth0", dhcp_id.addr, dhcp_id.prefix);
}
```

The translation of the informal list of steps to an NCD process was rather straightforward. We replaced each step by specifying the appropriate statement in the process. The first part of the statement, in front of the opening parenthesis, is the *statement type*, and corresponds to the textual description of the step, excluding any specific parameters. The NCD programming language implements a fixed set of documented statement types, including the above network-configuration statements. The parts inside the parentheses are the arguments to the statements; they are passed to and interpreted by the internal implementation of the various statement types. There is one slightly problematic part in our informal list of steps; notice how the last step, which assigns an IP address, refers to the IP address that was reserved using the DHCP protocol. This address is not known in advance; it is known by, and needs to be provided by, the DHCP configuration step. To support such cases, statements in NCD are allowed to export named *variables*, which may be read from following statements in the same process. In our case, the *net.ipv4.dhcp*[9] statement exports the variables named *addr* and *prefix*, corresponding to the reserved IP address and the prefix length of the local network, respectively. To gain access to these two variables, the *net.ipv4.dhcp* statement is declared together with an arbitrary identifier, in this case *dhcp_id*, and this identifier can be used to pass the exported variables as arguments to the *net.ipv4.addr* statement[10], by adding a dot and the variable name to the statement identifier.

As a special case, statements can export a variable with an *empty name*. An empty-name variable is simply a variable whose name is an empty string; its value is passed by specifying the statement identifier directly, without a dot. The *var*[8] statement is a simple example of a statement exporting an empty-name variable. This statement accepts a single argument, and exposes the value of that argument as the empty-name variable. For instance, we can use the *var* statement to avoid the repetition of the string *"eth0"* in our program, as seen in Listing 2.2.

Listing 2.2

```
process eth0_interface {
    var("eth0") dev;
    net.backend.waitdevice(dev);
    net.up(dev);
    net.backend.waitlink(dev);
```

```
    net.ipv4.dhcp(dev) dhcp_id;
    net.ipv4.addr(dev, dhcp_id.addr, dhcp_id.prefix);
}
```

Let us emphasize that in this program, technically, it is not the *var* statement itself (as identified by *dev*) that is being passed, but rather the value of its empty-name variable. The underlying mechanism is the same when *dev* is being passed to *net.backend.waitdevice* and when *dhcp_id.addr* is being passed to *net.ipv4.addr*. Indeed, trying to pass an empty-name variable of a statement that does not expose it will result in a runtime error, the handling of which will be discussed later in the chapter about error handling. Even though the syntax has a tendency to deceive, it is naturally suited for many uses, including the *var* statement, which has just been presented.

Care should be taken not to confuse *NCD variables* with the *var* statement; they have nothing to do with each other. While the *var* statement looks like a variable declaration in a traditional sense, there is nothing special about it. It is simply one of the implemented statement types with its own defined behavior.

The separation of the NCD language into the *core language* and the various *statement implementations* is one of its defining characteristics. Considerable effort has been put into integrating as much functionality as possible into various statements and keeping the core language small and simple, and its implementation correct. In the current implementation of the NCD language, which is an interpreter written in the C programming language, these two components of NCD are in fact decoupled and communicate with each other through a strictly defined internal application programming interface (API). We will refer to the current implementation of the core language as the *interpreter core*. New statement types may be implemented without changing the definition of the core language and effectively the interpreter core. They can either be linked statically into the NCD interpreter, or loaded dynamically at runtime.

2.2 Values

In the NCD core language, *values* or *NCD-values* are the entities being passed to statement arguments - and they are in fact not used for any other purpose. As has already been demonstrated, a value can be obtained either by literal specification, or by referencing a variable within a statement. We have previously seen the passing of *strings*, but a string is just one kind of value. In general, a value can also be a list or a string. More precisely, an NCD value is defined recursively as follows:

- A *string* is a value, and is defined by an ordered sequence of bytes. A string can be represented within the NCD language by enclosing its contents in a pair of double quotes, and prefixing (*escaping*) any double quotes or backslashes in the contents with a backslash. For example, the string *hello, "world"* can be represented as *"hello, \"world\""*. There are other escape sequences, such as \n for a newline character and \xDD for a hexadecimal byte specification (e.g., \x41 for the character A), but these need not be used; the bytes they represent may be entered literally as well.

- A *list* is a value, and is defined by an ordered sequence of *values*. A list is represented by inserting commas between the representations of the values it contains and enclosing that in a pair of curly braces. For example, a list containing the strings *hello* and *world* can be represented as {*"hello", "world"*}, and an empty list can be represented as {}.

- A *map* is a value, and is defined by a set of ordered pairs of *values*, where the first components of the pairs are unique within the map. We call the components of such an ordered pair the *key* and the *value*, respectively. A map is represented by inserting commas between the representations of the individual pairs and enclosing those in a set of brackets; each pair is represented as the representation of the key, followed by a comma, and followed by the representation of the value; the order of the pairs in the map representation is insignificant. For example, a map containing a pair with the key *"hello"* and the value *"world"*, and another pair with both the key and the value equal to an empty list, can be represented as *["hello":"world",*

{}:{}]. Note the ability to use any kind of value in both the key and value parts, and to combine different types of keys in a single map.

The above definition of the concept of a *value*, together with the value representation definitions, is everything that the interpreter core operates with; even the value representation definitions are only relevant when the program is being parsed. For example, the interpreter core does not provide any way to modify a value, or to generate its representation; it just needs to know enough about values to be able to pass them as arguments to statements. The modification and representation of values is one of those features that is left up to the various statement implementations. In particular, the *value* statement[38] can be used to manipulate values; these are presented later in the chapter about imperative programming in NCD.

On the other hand, value representations can be generated using the *to_string* statement[11], as is demonstrated here. Together with the *println* statement[12], this allows printing values to the terminal for debugging purposes; see Listing 2.3.

Listing 2.3

```
process test_printing_values {
    to_string("Hello") str1;
    println(str1);
    to_string({str1, "World"}) str2;
    println(str2);
    to_string({["Hello":"World", "Goodbye":"Earth"]}) str3;
    println(str3);
}
```

Notice how this program reads the empty-name variables of the *to_string* statements, which is the string representation of the value passed as an argument. An important, though almost obvious, feature which has not been demonstrated until now is that a literal specification of a list or map can itself include variables. This can be seen in the argument to the second *to_string* statement. The output of this program is shown in Listing 2.4.

Listing 2.4

```
"Hello"
{"\"Hello\"", "World"}
{["Goodbye":"Earth", "Hello":"World"]}
```

2.3 Method-like statements

Up to this point, we have just seen statements where the only input provided to the statement implementation is a list of arguments in the form of NCD values. However, such a simple model of statement initialization proved to be insufficient for many purposes. Problems arise when we need a way to perform imperative *actions* on a statement, as opposed to just initializing the statement and possibly reading the results exposed through the statement variables. A mechanism is necessary to allow a statement to refer to an existing statement.

We will start with the example of modifying the value of a variable, in the meaning of common imperative languages. However, since variables in the classical sense do not exist in NCD (only statements do), our goal is instead to modify the value encapsulated by a *var* statement. The modification itself will, of course, take the form of a statement, but it will be a new kind of statement we have not seen before - a *method-like statement*. The input to method-like statements is not only the argument list, but also a reference to an existing statement, of which the implementation of the method-like statement has intimate knowledge. In general, method-like statements are written as *statement_id->method_name(arguments)*.

The value of a *var* statement can be changed by using the *set* method-like statement. The program shown in Listing 2.5 will print a message when the link layer of a network interface has started operating for the first time in the lifetime of the program, but will print a different message for every subsequent time this happens.

Listing 2.5

```
process main {
    var("The link layer has started operating for the first time")
        msg;
```

```
    net.backend.waitlink("eth0");
    println(msg);
    msg->set("The link layer has started operating again");
}
```

We use the form *var::set* to mean "the method-like statement *set* acting on statements of type *var*". Method-like statements are specific to and defined by the statements they act on. For example, using *set* on the identifier of a *println* statement has no meaning and results in a runtime error.

The reason our program performs as expected is that the *msg-)set* statement does not change *msg* back to its original value when the program backtracks as the link layer stops operating. When the link layer starts operating for the second time, *msg* therefore still has the new modified value which is passed to *println*. As can be seen here, there are no requirements about what the deinitialization actions of statements should be; while for many configuration commands, such as *net.ipv4.addr*, the deinitialization action has the reverse effect of the initialization action, the *msg->set* statement in our program doesn't do anything when it deinitializes.

In NCD we call an *imperative* statement one whose deinitialization action is no action at all. Frequently, the only effect of an imperative statement is the modification of the internal state of other statements. As more statements and other language features will be introduced, it will be become clear that the execution model of NCD is really an extension of the imperative programming paradigm by adding the automatic backtracking feature. To elaborate, the program can still be viewed as a sequence of statements that manipulate the program state, including conditional branching and looping. We can write completely imperative programs in NCD by only using the imperative statements, even though this is not what NCD was designed for. Many statements in NCD whose functions are conceptually imperative still leverage automatic backtracking for ease of use and integration with other parts of the language.

2.4 Multiple processes

By declaring a process in an NCD program, the interpreter will automatically begin working with it as soon as the program is loaded. We have never stated that there could not be more than one process in a program, and indeed there is no limitation on the number of processes that can be declared. However, once we're working with two or more processes, it is not immediately obvious how exactly the program would execute. For example, if one of the processes is currently initializing a statement, what is happening with the other process in the mean time? The short answer is that if an NCD process is considered a task in terms of scheduling, the processes are scheduled on to the interpreter's single execution thread in a manner called *cooperative multitasking*. This means that while a process is *busy*, its execution will never be interrupted by another process, until it voluntarily returns control[17]. Considering that the NCD interpreter is an event-driven system, a process being busy simply means that it is *processing an event*, and returning control means that it has stopped processing an event, allowing for other events to be processed[18]. For example, a process that is blocked in a *sleep* statement[13] is not considered busy while it is sleeping; this is demonstrated in Listing 2.6.

Listing 2.6

```
process p1 {
    sleep("1000"); # milliseconds
    println("p1 done");
}
process p2 {
    sleep("2000");
    println("p2 done");
}
```

When executed, this program waits one second, prints "p1 done", waits another second, prints "p2 done", and finally proceeds to do nothing. When a statement wishes to wait for something to happen, it arranges to be asynchronously notified of the event, and returns control to the interpreter's event loop, allowing the processing of other events, possibly related to other processes. So, when the one-

second sleep in the first process is over, a timer event is created and processed, resulting in "p1 done" being printed, all while the second process is waiting for its own *sleep* to complete. However, in order to understand the semantics properly, we need to describe the behavior of processes in NCD in terms of events and reactions to those events.

What we have so far called the initialization of a statement is really split into two events: the *initialization request*, which involves the interpreter core calling into the statement implementation in order to create a new instance of the statement, and the *initialization completion*, which is reported by the statement implementation to the interpreter core. For example, in our case, the initialization request for a *sleep* statement results in the creation and starting of a timer, and initialization completion is reported upon the expiration of this timer. Something similar can be said about the deinitialization of statements (which in our case only happens when the interpreter receives a termination request from the operating system). That is, the deinitialization of a statement is composed of a *deinitialization request* from the interpreter to the statement implementation, and *deinitialization completion* being reported by the statement implementation. Now that we are already discussing the behavior of NCD processes in terms of events, we should not forget about backtracking. A statement that has reported initialization completion is permitted to *trigger backtracking*, which makes the interpreter start deinitializing any statements below that statement. When a statement triggers backtracking, it effectively returns back to the same state as before it reported initialization completion; it can report initialization completion once again to permit the interpreter core to begin initializing statements below it again.

There are many statements that do not take any time to initialize, such as the *var* statement (and currently also the *println* statement because it blocks the entire interpreter). However, these really are nothing special; they simply report initialization completion immediately upon receiving an initialization request. The internal architecture of the interpreter ensures that this works correctly. In such cases, the interpreter also ensures that statement initialization is performed in an atomic manner without any other event being processed between the initialization completion of the previous statement in the process and the initialization request for the next statement in the process (assuming those statements exist). There

is, however, a single exception to this rule when an unexpected error occurs while initializing the statement. Error handling in NCD is the subject of the next section.

Finally, notice how there appears to be a small semantic ambiguity in the program in Listing 2.6 - it is not clear which of the two processes begins to execute first. In this case it is irrelevant because the program will do the same thing no matter what; however we can easily write a program where the order in which processes begin to execute affects the results, as in Listing 2.7.

Listing 2.7

```
process p1 {
    println("p1");
}
process p2 {
    println("p2");
}
```

Without any further knowledge, we can only say that the program either prints "p1" then "p2", or "p2" then "p1" - but not anything else, because the *println* statement is known to initialize immediately, which ensures that the two prints are never interleaved. The semantics, however, is defined - the result we obtain is always "p2" then "p1". This is because the interpreter, after it finishes loading the program, will process events for the last process first, until there are no more events related to the last process, then it will proceed to processing events for the second-last process, and so on. However, these events do not have an external origin; rather, they are internally generated, and are used to synchronize event processing between the various components of the interpreter. In the section about process templates, a similar question regarding the execution order in the presence of multiple processes will be raised; however even in that case, it will turn out that the execution order is well defined.

2.5 Error handling

There are different unrelated kinds of errors that can occur during the execution of an NCD program, and they will all be explained in detail here.

Program loading errors occur at the program loading stage before any part of the program is actually executed, that is, before any initialization requests are generated. These errors include syntactic errors in the program and semantic errors that can be detected at the program-loading stage. Some examples of semantic errors at the program-loading stage are the existence of two processes with the same name, and the presence of duplicate keys in the literal specification of a map value, to the extent that this can be detected. When a program-loading error occurs, the NCD interpreter prints the information about the error to standard output and terminates.

Statement errors are errors raised by specific statement implementations to the interpreter core. The most common cause of statement errors is when the argument list passed to a statement is determined to be incorrect; other causes are usually related to the inability of a statement to acquire resources or the subsequent loss of resources that are required for the correct operation of the statement; this includes memory-allocation failures. As a special case, a memory-allocation failure that occurs inside the interpreter core before generating an initialization request is also considered as a statement error. A statement error always results in the sudden deinitialization of a statement; unless the problematic statement is the last in its process, a statement error also causes backtracking of the process to that statement. After this backtracking (if any) is complete, the interpreter will wait a fixed amount of time (5 seconds by default) and then try initializing the statement again. However, if the program backtracks *past* the problematic statement before its initialization could be retried (for example, because a statement above that statement has triggered backtracking), the fact that the statement has raised an error is effectively forgotten, so there will be no additional delay before this statement is next initialized. The rationale for this behavior is that the cause of the backtracking is likely related to the very reason our statement failed in the first place. As an example, we will use the program in Listing 2.8, which observes the link status of a network interface, in addition to observing the existence of the interface itself, in case it is hot-pluggable. The program uses the *rprintln* statement[12], which prints text to the terminal when it is deinitialized, that is, when the process is backtracked over it.

Listing 2.8

```
process main {
    net.backend.waitdevice("eth0");
    net.backend.waitlink("eth0");
    println("Link up.");
    rprintln("Link down."); # prints text on deinitialization
}
```

We will assume that the *net.backend.waitlink* statement acquires some kind of resource specific to the *eth0* network interface that allows it to observe its link status, and that it raises a statement error when it loses this resource due to the removal of the network interface. Even though this is not really what happens in the current implementation (instead, when the interface is removed, it reports the link went down, but fails to report the link going up should the interface ever reappear), we insist on the assumption, since it could legitimately work that way, and there are other statements in NCD which do.

When a network interface is removed, this event is delivered by the operating system, both to the *net.backend.waitdevice*[14] and *net.backend.waitlink*[15] statements. We have no control over the order of that delivery with respect to the two statements; the events can be delivered in any order, depending on the internal workings of the operating system. We are particularly interested in the case where the *net.backend.waitlink* statement is informed of interface removal first. In this case, *net.backend.waitlink* will raise a statement error. However, soon after, the *net.backend.waitdevice* will also be informed about the interface removal, and it will trigger backtracking in response. As a result, the program will backtrack past *net.backend.waitlink*, and the occurrence of the error it has triggered will be forgotten. If the interpreter did not forget errors raised by statements it backtracks over, then there would be a problem in case the device is immediately plugged back in - it would wait some time before proceeding to initialize the *net.backend.waitlink* statement once again. But this is not desired; the initialization of *net.backend.waitlink* will likely succeed without error immediately, because when the original error was triggered, it was probably due to the network interface being removed, which has since been resolved.

The NCD language does not implement more complex error-handling mechanisms, such as throwing and catching of exception objects, which is available in many imperative programming languages[16]. The simple error-handling model described above suffices in most cases where NCD is used for network configuration. The model makes it easier to recover from transient errors, and to isolate errors to the components of the program that they occur in. On the other hand, if exceptions were used, special care would have to be taken to avoid exceptions from propagating too far, and possibly terminating the entire program. After all, the ability to write reliable systems with ease is one of the goals of the NCD language.

However, sometimes it is still desirable to handle errors in a way similar to throwing exceptions. This primarily occurs when imperative style code is written in NCD, where various kinds of errors are expected to occur and need to be dealt with immediately. We will find later, in the chapter about imperative programming, that a mechanism similar to exception throwing can be implemented inside the NCD language by making use of the backtracking feature and some built-in statements. This is one of the prime examples of how the NCD language, despite its simplicity, manages to expose the sense of seemingly complex features available in other programming languages.

2.6 Process templates

Let us try to establish a rough correspondence between the external appearance of an NCD program, and that of a simple imperative programming language, such as *C*. In NCD, all statements are contained within process definitions; in *C*, they are contained within function definitions[19]. In this regard, NCD processes and *C* functions are similar in nature. However, there is an important difference - in an NCD program, all processes start executing automatically when the program starts, but in *C*, only the *main* function is called automatically; other functions are only useful when they are explicitly called. But then if we try to complete this correspondence, a question arises: is there anything in NCD that behaves similarly to functions in *C*, other than the *main* function? The reasons why such a feature is desirable are similar to why *C* functions are useful; these include the easier reuse of commonly occurring code, the separation of responsibilities, and the definition

of abstractions, among others[20].

This feature we are seeking does in fact exist in NCD, and its underlying mechanism is called *process templates*. A process template definition looks exactly like a process definition, except that it is introduced by the *template* keyword instead of the *process* keyword. Listing 2.9 demonstrates a process-template definition.

Listing 2.9

```
template test {
    println("This statement appears in a process template.");
}
```

Similarly to functions in *C*, a process template, by itself, does not affect the runtime behavior of a program. Instead, process templates are *instantiated* by various statements specifically designed to work with process templates. The result of process-template instantiation is a *template process*. Once a template process is created, it behaves much like a normal process declared with the *process* keyword. A template process exists only as an abstract entity during program execution, which encompasses the runtime state of all the statements that appear in the corresponding process template. Of course, multiple processes can be created by instantiating a single process template and left to execute concurrently.

Like regular processes, the execution of template processes is managed by the interpreter core, the part of the NCD language implementation which implements the core language. The other side of this interface, the statement implementations, can only work with template processes to the extent that the internal API between the interpreter core and the statement implementations permits. Specifically, statement implementations have the following essential abilities related to the control of template processes:

- Statement implementations can request the interpreter core to *create* a template process. The particular process template that is to be instantiated is identified by its name, as it appears in the program; the statement requesting process creation does not need to know anything about the contents of the process template. In the case of successful process creation, the statement implementation receives an abstract handle to the process through which

the process is accessed and controlled. We will refer to the part of the statement implementation which has received the abstract process handle as the *controller* of the process.

- The controller of a template process can *request its termination*. When termination is requested, the interpreter core starts to backtrack the entire processes, deinitializing its statements from the bottom up. Once the process has finished backtracking, the controller is notified of the event, and the lifetime of the template process ends. A controller can not abandon control of its process before it has terminated. For example, if a non-method-like statement controlling a template process receives a deinitialization request, it first has to request termination of the process and wait for the process to terminate before it is permitted to report deinitialization completion.

- The controller is notified when the process has been *completely initialized*, that is, when all the statements in the process have completed initialization. After the process has completely initialized, the controller is also notified when an event occurs that causes the process to *no longer be completely initialized*; for example, when one of its statements triggers backtracking. When this happens, the controller has the ability to *delay backtracking* in the process. These abilities are necessary to allow correct implementation of statements which act as if they were replaced with the statements inside a process template. The *call* statement, which will be introduced soon, is an important statement that behaves in this way.

Now that the theory of process control in NCD has been outlined, we will introduce some of the statements which have the ability to create template processes. We will start with the *process_manager* statement[21], which is capable of controlling multiple template processes. To create a process controlled by *process_manager*, the *process_manager::start* method-like statement must be used. *process_manager::start* is an imperative statement - it does not do anything when it is deinitialized, and, in particular, it does not request termination of the process that it created. Termination of all the controlled processes is requested when the *process_manager* statement itself is requested to deinitialize; when this happens, *process_manager* waits until all the controlled processes have terminated and only

then reports deinitialization completion to the interpreter core. The program in Listing 2.10 demonstrates the basic operation of *process_manager*.

Listing 2.10

```
process main {
    process_manager() mgr;
    mgr->start("test", {"FirstInstance"});
    mgr->start("test", {"SecondInstance"});
}
template test {
    println("Starting ", _arg0);
    rprintln("Terminating ", _arg0);
}
```

This program will create two template processes based on the *test* process template. The name of the process template to use for process creation is specified by the first argument to *process_manager::start*. As the example shows, template processes can be passed a list of arguments when they are created, which themselves are NCD-values. The statement implementation that creates a template process provides the argument list for the process; in the case of *process_manager*, we pass the argument list as the second argument to *process_manager::start*. After the process is created, the arguments are available from inside the template process through the *_argN* special identifiers. Additionally, *_args* exposes all the arguments as a list value.

The ability to pass arguments is important for the usability of template processes. By passing arguments to a template process we can tell it what specifically it is supposed to do, and effectively we can differentiate between processes created from the same process template. For example, in the domain of network configuration, we can create a process template that can manage a wired network interface, then instantiate it for each of the wired network interfaces that we would like to manage, passing the name of the network interface as an argument.

Recall the problem we were facing at the end of the section about multiple processes, the one where our program, in Listing 2.7, consisted of two processes, each containing a single *println* statement. We were wondering which of these two

statements is initialized first, and the resolution was that processes declared lower in the program have priority. A very similar question arises about the behavior of the *process_manager::start* statement, as shown in Listing 2.11.

Listing 2.11

```
process main {
    process_manager() mgr;
    mgr->start("test", {});
    println("start completed");
}
template test {
    println("template process created");
}
```

So, what is printed first - "start completed" or "template process created"? The language guarantees the latter option. An informal explanation is that this is so because *start* is implemented such that the *immediate effects* of template process creation are processed before the immediate effects of the initialization completion of *start*. Notice the *immediate* in *immediate effects*. For example, if we inserted a *sleep("0");* before the *println* in our process template, printing the text is no longer considered an immediate effect of process creation (only starting the sleep timer is), so the print order would be reversed.

2.7 Identifier-resolution forwarding

The controller of a template process can also hook itself into the *identifier-resolution procedure* of the controlled process. This allows it to introduce predefined identifiers into the scope of the process and specify their semantics. In particular, *process_manager* defines a special identifier *_caller* inside its controlled processes which serves as a gateway into the scope of the *process_manager* statement itself. The program in Listing 2.12 shows a template process controlled by *process_manager* accessing a *var* statement using this mechanism, both by reading its empty-name variable (remember, the one which exposes its stored value), as well as by invoking a method-like statement (*var::set*) to change its stored value.

Listing 2.12

```
process main {
    var("Hello!") msg;
    process_manager() mgr;
    mgr->start("test", {});
}
template test {
    println(_caller.msg);
    _caller.msg->set("Changed!");
}
```

When the controller of a template process hooks into the process's identifier-resolution procedure, the identifiers that it introduces into the process, with regard to their scope, behave as if they were declared at the very top of the process. If a playful programmer were to declare a statement with the identifier *_caller* inside a process controlled by *process_manager*, that statement would indeed take precedence over the special *_caller* identifier introduced by *process_manager*, for anything that appears below the offending statement.

The ability of process controllers to forward identifier resolutions through statements should seem a bit magical at this point. After all, the controller would need some kind of access to the scope of the statement through which identifier resolutions are to be forwarded. But the answer is simply that it has that access - the internal API for communication between statement implementations and the interpreter core allows any statement to access identifiers in its scope, that is, anything that appears above it.

Another statement which exercises the ability to access identifiers in its scope is the *alias* statement[22]. The sole purpose of the *alias* statement is to forward identifier-resolution requests to an identifier in its scope, which is specified by the single string argument to *alias*. This effectively makes the *alias* statement an alias for its target identifier. Listing 2.13 demonstrates the behavior of the *alias* statement and its possible uses.

Listing 2.13

```
process main {
```

```
    var("Hello") msg;
    alias("msg") msg_alias;
    println(msg_alias);
    msg_alias->set("Good");
    process_manager() mgr;
    mgr->start("test", {"_caller.msg", "World"});
}
template test {
    alias(_arg0) msg;
    alias("_arg1") new_text;
    println(msg);
    msg->set(new_text);
    println(msg);
}
```

A common use for the *alias* statement is to give descriptive identifiers to template-process arguments in order to make the code easier to write and understand; in the example, the second argument to the *test* template is accessed via the *new_text* alias.

However, notice the way in which *alias* is used for the first argument to the *test* template. Unlike the second argument, it is not an alias for an argument; instead, the argument is directly passed to the *alias* statement, determining its target. By exploiting the special *_caller* identifier provided by *process_manager*, this allows us to effectively pass a reference to a statement, and allows the template process to not only read variables exposed by the referenced statement, but also invoke method-like statements on it.

Finally. because the argument to the *alias* statement can be any string, it can be used for runtime indirection in general, even though most such uses would be considered bad programming style. See Listing 2.14, where a process template receives a digit in the form of a string and prints the textual form of the digit. There, the *concat* statement[23] is used to construct the target of the alias.

Listing 2.14

```
template print_digit_text {
```

```
    alias("_arg0") digit;
    var("zero") str0;
    var("one") str1;
    var("two") str2;
    var("three") str3;
    # and so on...
    concat("str", digit) target;
    alias(target) str;
    println(str);
}
```

Again, this is just a demonstration of the theoretical abilities of the *alias* statement and the NCD core language. No programmer in his right mind would produce such code for production purposes; NCD provides better ways to map values to other values, but these will be explained in due time.

The ability to build these *identifier expressions* at runtime and pass them to *alias* is not unlike the so-called *eval* functions that are present in some programming languages, such as *JavaScript*[24]. The *eval* functions most commonly accept a string argument, which they parse and evaluate as an expression in the programming language concerned and return the result of the evaluation. However, *alias* is very limited compared to general *eval* functions, as it can only express a list of identifiers, separated by dots.

2.8 Including files

Definitions of processes and templates can be separated into multiple files, and later inserted into a specific place in a program using the *include* directive. This makes it possible to keep logically contained components in their own files in order to ease the development and maintenance of a program. The *include* directive may only be used at the top level and not within a process or a process template. The path given to *include* is interpreted relative to the directory where the including file resides, unless the path is absolute. Listing 2.15 shows how the *include* directive is used.

Listing 2.15

```
# File program.ncd
include "helper.ncdi"
process main {
    process_manager() mgr;
    mgr->start("helper", {});
}
# File helper.ncdi
template helper {
    println("Hello from helper!");
}
```

Yet this does not quite work when two included files both end up including the same file, because any processes or templates in that file will appear multiple times in the composed program, resulting in a program-loading error. To solve that, the *include_guard* directive was added. This directive must be provided with a special identifier; when a file containing an *include_guard* is included, but an *include_guard* with the same identifier has previously been encountered in another file, the inclusion is automatically ignored. If the identifier uniquely identifies a file, this mechanism will prevent multiple inclusions of the same file. Listing 2.16 demonstrates how a file can be fixed to avoid errors when it is included multiple times.

Listing 2.16

```
# File helper.ncdi
include_guard "helper"
template helper {
    println("Hello from helper!");
}
```

Chapter 3

Other basic features

3.1 Calling templates

Remember how we introduced process templates by looking for an equivalent to functions in the *C* language? Well, using process templates with *process_manager* is still not quite like *C* functions. In particular, calling the *process_manager::start* statement only creates a process; it does not wait for the process to completely initialize, or perhaps even return some kind of data to the code starting the process. But *process_manager* is just one of many statements that can create template processes. We will find later in the section on dependencies that it is still possible to effectively wait for a process created by *process_manager::start* to completely initialize and to return data from it; however, here we introduce the *call* statement[25], which solves this problem much more directly.

Like *process_manager::start*, the *call* statement accepts two arguments - the name of a process template and an argument list. Upon the reception of an initialization request, it creates a template process based on this information. Identifier resolutions inside the called process are hooked so that the special *_caller* identifier provides a gateway into the scope of the *call* statement. However, unlike *process_manager::start*, the *call* statement will only report its own initialization completion once the called process has completely initialized. After this occurs, the identifier of the *call* statement serves as a gateway into the scope of the called process. That is, both the called process can access identifiers in the calling process,

and the calling process can access identifiers inside the called process. The program in Listing 3.1 shows the basic functionality provided by the *call* statement. Notice how the returning of a value is simulated by accessing a statement within the called process (*old_x*).

Listing 3.1

```
process main {
    var("hello") x;
    call("change_x", {}) c;
    println(c.old_x, x);
}
template change_x {
    var(_caller.x) old_x;
    _caller.x->set("world");
}
```

As far as this example is concerned, it appears that the *call* statement really behaves as if it was literally replaced by the statements inside the process template, with some differences, such as the introduction of a new scope, the ability to use *call* recursively, as well as the ability to specify the called process template at runtime. However, so far we have insufficient information to say that *call* really behaves like that, because we do not know how it responds to various events related to backtracking. The program in Listing 3.2 exposes this issue.

Listing 3.2

```
process main {
    call("wait_for_device_and_link", {"eth0"});
    println("Link up.");
    rprintln("Link down.");
}
template wait_for_device_and_link {
    net.backend.waitdevice(_arg0);
    net.backend.waitlink(_arg0);
}
```

The question is whether this program is functionally equivalent to the program in Listing 3.3, which does not call a process template. We are concerned about what happens in the first program, the one using *call*, when the link goes down - does it result in the program printing "Link down.", like in the second program?

Listing 3.3

```
process main {
    net.backend.waitdevice("eth0");
    net.backend.waitlink("eth0");
    println("Link up.");
    rprintln("Link down.");
}
```

It is indeed the case generally that *call* behaves as if it was replaced by the statements inside the called process template, backtracking behavior included. However, before we can be sure, we need to examine how the *call* statement reacts to various events, and confirm that the resulting behavior is equivalent to replacing *call* with the statements inside the called process template.

When *call* receives a *deinitialization request* from the interpreter core, it requests termination of its process, waits for the process to terminate and finally reports its own deinitialization completion. The reception of a deinitialization request indicates that the calling process is being backtracked over the *call* statement. The end result of *call*'s reaction is that backtracking proceeds as follows: first, statements in the calling process below the *call* are deinitialized; then statements inside the called process are deinitialized, and finally backtracking continues by deinitializing statements above the *call*. This really is exactly what would have happened if *call* was replaced with the statements in the process template.

The second case to be examined, and the most complex one, is when the called process is completely initialized and one of the statements inside it *triggers backtracking* or experiences a statement error, which implicitly results in backtracking. The *call* statement is informed of this event; in response it delays backtracking within the called process and instead itself triggers backtracking, in the calling process. It then waits until the calling process finishes backtracking to *call*, that is, after all statements following *call* have been deinitialized (the interpreter core API

makes it possible for statements to be informed upon completion of backtracking). After this happens, *call* finally resumes backtracking inside the called process. The resulting behavior of the program in case one of the statements within the called process triggers backtracking is that first statements in the calling process are deinitialized from the bottom up, up to but not including the *call* statement, then statements within the called process are deinitialized until the statement which has triggered backtracking is reached. Again, this behavior is exactly the same as if *call* was replaced by the statements within the called process template.

This is not a formal proof of the stated equivalence; it is even a bit incomplete, because we have assumed that individual backtracking operations happen in isolation. In reality, after one statement triggers backtracking, another statement above it can trigger backtracking as well, before the first backtracking is complete; similarly, termination of the entire process can be requested while backtracking is in progress. Nevertheless, these cases can be analyzed with regard to the implementation of the *call* statement, and the equivalence stands.

3.2 If clause

A feature of essential importance in imperative programming languages may appear to be missing from the NCD language: the conditional branch, or simply *if*[26]. In NCD, how can we initialize a set of statements only if a given runtime condition is satisfied? Actually, this can already be done with what has been presented so far. The code in Listing 3.4 demonstrates how the *call* statement can be used (or perhaps abused) to simulate a conditional branch. It uses the *val_different* statement[28] to check whether two values are different.

Listing 3.4

```
template print_if_not_hello {
    alias("_arg0") x;
    val_different(x, "Hello") different;
    concat("print_different_", different) template_name;
    call(template_name, {});
}
```

```
template print_different_false {
    var(""); # no-op statement, templates cannot be empty
}
template print_different_true {
    println(_caller.x);
}
```

This is one of those cases where we construct identifier expressions at runtime in order to achieve some kind of indirection. It is similar to when we abused *alias* in Listing 2.14 in order to select one of a set of statements based on a runtime condition, except that here we're selecting between process templates.

To avoid the abuse and allow more elegant and efficient branching, a statement was created specifically for this purpose: *embcall2_multif*[25]. As its cryptic name might suggest, this statement is not intended to be used directly by programmers, but through *syntactic sugar*[27] which is implemented in NCD as a translation phase. Regardless, we first explain this internally used statement, to allow for a deeper understanding of the NCD language and its implementation.

The *embcall2_multif* statement takes a variable number of arguments. The first two arguments are the *condition*, which is a string, and a template name. The statement will call the specified template if the condition is equal to the string "true", in a manner similar to the *call* statement. Any extra number of argument pairs consisting of a condition and a template name may be specified; these will be examined if the first condition is not true, in order. At last, a template name may be specified which is used if none of the conditions are true. However, there are important differences in the manner of process creation between *embcall2_multif* and *call*. First, *embcall2_multif* does not pass any arguments to the called process template; even the special identifier *_args* does not exist, as opposed to being an empty list. More importantly, the process created by *embcall2_multif* has *direct* access to the scope of the *embcall2_multif* statement, and not through the *_caller* gateway that is provided by *call*. Listing 3.5 shows how *embcall2_multif* can be used directly to implement the same semantics as the previous example that abuses the *call* statement.

Listing 3.5

```
template print_if_not_hello {
    alias("_arg0") x;
    val_different(x, "Hello") different;
    embcall2_multif(different, "print_different");
}
template print_different {
    println(x); # note the absence of _caller
}
```

This does look much better than before, when we were using *call*. However, the need to specify conditionally initialized statements externally inside a separate process template is still bothering us. To this end, the NCD language provides syntactic sugar around the *embcall2_multif* statement, in the form of the *If* clause. The *If* clause allows conditionally initialized statements to be specified directly in the process or process template where branching is desired. The code sample in Listing 3.6 demonstrates the full form of the *If* clause, including the optional *Elif* parts and the optional *Else* part. The *val_equal* statement[28] is used to compute conditions for the *If* clause. Additionally, it is shown how statements within the conditional blocks can be accessed by statements outside the *If* clause, as long as the *If* clause is given an identifier. We have in fact already seen a similar ability in the *call* statement, but with the branching behavior here it may appear unusual.

Listing 3.6

```
template test {
    alias("_arg0") x;
    val_equal(x, "one") is_one;
    val_equal(x, "two") is_two;
    If (is_one) {
        var("It's a One") msg;
    } Elif (is_two) {
        var("It's a Two") msg;
    } Else {
        var("It's something else.") msg;
```

```
    } branch;
    println(branch.msg);
}
```

It is easy to see how the NCD interpreter would translate this program into a form without syntactic sugar, that is, *desugar* it. Each of the three conditional blocks would be turned into a process template with an automatically generated name, and the entire *If* clause would be replaced by the *embcall2_multif* statement, as is shown in Listing 3.7. Hovever, the names of the generated templates would be chosen differently, in a way which minimizes any interference with user-defined processes and process templates.

Listing 3.7

```
template test {
    alias("_arg0") x;
    val_equal(x, "one") is_one;
    val_equal(x, "two") is_two;
    embcall2_multif(is_one, "case1", is_two, "case2", "case3")
        branch;
    println(branch.msg);
}
template case1 {
    var("It's a One") msg;
}
template case2 {
    var("It's a Two") msg;
}
template case3 {
    var("It's something else.") msg;
}
```

3.3 Foreach clause

We have already used list values in order to pass arguments to template processes, where the lists ended up being interpreted internally by the implementations of the statements we were using, such as *call*. But is there a way to interpret list values manually from inside the NCD language? For example, given a list of network interface names, we would like to wait for all of them to exist, as well as trigger backtracking appropriately when any one of them ceases to exist. In NCD, element-wise operations on lists are implemented by the *Foreach* clause. Listing 3.8 demonstrates how the *Foreach* clause can be used to solve the problem of observing the existence of multiple network interfaces.

Listing 3.8

```
template wait_for_interfaces {
    alias("_arg0") interfaces;
    Foreach (interfaces As one_interface) {
        net.backend.waitdevice(one_interface);
    };
}
process main {
    call("wait_for_interfaces", {{"eth0", "eth1"}});
    println("All interfaces exist.");
    rprintln("Some interfaces don't exist.");
}
```

The argument to *Foreach* immediately after the opening parenthesis is the list being operated upon. This list is followed by the keyword *As*, and then by an arbitrary identifier. Inside the block of statements that *Foreach* is supplied with, this identifier refers to a particular element of the list.

It should come as no surprise at this point that the *Foreach* clause internally works by creating template processes. The clause attempts to create a template process for every element of the list - first, it creates the template process for the first element; once this one has completely initialized, it creates a template process for the second element, and so on. Backtracking is also handled as would

be expected; put simply, *Foreach* manages its template processes in a manner very similar to how the interpreter core manages statements within a process. Fortunately, we have an easy to understand equivalence, like we did when we were discussing the *call* statement. The operation of *Foreach* is the same as if it was *replaced with multiple call statements*, one for each element of the list being passed to it, with minor exceptions, such as the introduction of new scopes and the special element identifier provided by *Foreach*.

Like the *If* clause, *Foreach* is implemented as syntactic sugar, where the block of statements is translated into an automatically generated process template, and the clause itself into a statement. In the case of *Foreach*, this statement is called *foreach_emb*[29]. However, *foreach_emb* is not intended to be used directly by programmers; instead the *foreach* statement is available, in case the programmer prefers to explicitly declare a process template. The use of the *foreach* statement[29] is demonstrated in Listing 3.9.

Listing 3.9

```
template wait_for_interfaces {
    alias("_arg0") interfaces;
    foreach(interfaces, "one_interface", {});
}
template one_interface {
    net.backend.waitdevice(_elem);
}
```

In processes created by the *foreach* statement, the respective list elements are always available as *_elem*. Like *call*, *foreach* allows passing arguments to the template process and provides the *_caller* gateway.

Both the *Foreach* clause and the *foreach* statement can also operate on map values, in addition to list values. In this case, the *Foreach* clause must be provided with two identifiers, separated by a colon, representing the key and the value of a map entry, respectively. On the other hand, the *foreach* statement exposes the key as *_key* and the value as *_val*.

3.4 Dependencies

If we look at the NCD language from the right perspective, we can see that its entire execution model is based on dependencies between statements. Let us start by saying that each statement depends on the ones above it, and ceases to operate correctly when the dependency is broken; and that each statement can either be in a state where other statements can rely on it, or in a state where they cannot. This almost completely defines the execution model of NCD; just substitute "statement triggers backtracking" for "statement becomes unreliable" and "statement reports initialization completion" for "statement becomes reliable". Backtracking, the way we have described it, is implied, because the only sensible action to do when a statement has one of its dependencies broken is to deinitialize it and attempt to initialize it again after its dependencies have been satisfied.

However, the dependency mechanism built into the language may not be sufficient when more complex dependencies need to exist between the components of a program. Suppose we have some kind of service which depends on two network interfaces being online - perhaps another network interface which we want to disable unless both of these two network interfaces are online. Our starting point is the code in Listing 3.10, which consists of two processes, each managing a network interface of its own. These two processes exist as template processes, not normal processes, because this will turn out to be easier to work with when we try to implement dependencies. The code is short for clarity; in practice, the interface processes would be doing much more than merely using *net.backend.waitdevice*, such as assigning IP addresses.

Listing 3.10

```
process main {
    process_manager() mgr;
    mgr->start("interface1", {});
    mgr->start("interface2", {});
}
template interface1 {
    var("eth0") dev;
    net.backend.waitdevice(dev);
```

```
}
template interface2 {
    var("eth1") dev;
    net.backend.waitdevice(dev);
}
```

We could try to use the *call* statement, instead of *process_manager*, to implement the dependency on both network interfaces, as shown in Listing 3.11.

Listing 3.11

```
process main {
    call("interface1", {});
    call("interface2", {});
    call("depending_interface", {});
}
template depending_interface { ... }
```

This program does achieve the goal that *depending_interface* effectively depends on both *interface1* and *interface2*. However, the behavior of the program has changed with regard to the these two interfaces: *interface2* now depends on *interface1*, while originally there was no such dependency. In general, this dependency is undesirable, since *interface2* can likely operate in a useful manner without *interface1* being online.

This problem can be properly solved by using the *depend_scope* statement[30] and the associated method-like statements *provide* and *depend*. The program in Listing 3.12 demonstrates the usage of *depend_scope* to properly implement dependencies in our problem.

Listing 3.12

```
process main {
    depend_scope() depsc;
    process_manager() mgr;
    mgr->start("interface1", {});
    mgr->start("interface2", {});
```

```
    mgr->start("depending_interface", {});
}
template interface1 {
    var("eth0") dev;
    net.backend.waitdevice(dev);
    _caller.depsc->provide("interface1-dep");
}
template interface2 {
    var("eth1") dev;
    net.backend.waitdevice(dev);
    _caller.depsc->provide("interface2-dep");
}
template depending_interface {
    _caller.depsc->depend({"interface1-dep"});
    _caller.depsc->depend({"interface2-dep"});
    # configure this network interface here
}
```

So, how does this work? Just like we did for some things before, we will start the explanation with an equivalence. The equivalence will only work for a specific way of using *depend_scope*, but it will be enough to make the general case easier to understand. So, as promised, the programs in Listing 3.13 and Listing 3.14 are equivalent, except for obvious differences such as the presence of multiple scopes in the second program.

Listing 3.13

```
process main {
    process_manager() mgr;
    mgr->start("test", {});
}
template test {
    # some statements here (A)
    # some statements here (B)
}
```

Listing 3.14

```
process main {
    depend_scope() depsc;
    process_manager() mgr;
    mgr->start("test1", {});
    mgr->start("test2", {});
}
template test1 {
    # some statements here (A)
    _caller.depsc->provide("dummy");
}
template test2 {
    _caller.depsc->depend({"dummy"});
    # some statements here (B)
}
```

In the second program, the one which uses dependencies, *depend* is used to delay the progression of the second process (*test2*) until the first process (*test1*) reaches its *provide* statement. This works as would be expected in the other direction too - if something causes the first process to backtrack over the *provide* statement, the *depend* in the second process will trigger backtracking, and only once this backtracking is complete will the *provide* statement report deinitialization completion, allowing backtracking in the first process to continue. In general, since multiple *depend* instances can be *bound* to a *provide*, *provide* will delay reporting deinitialization completion until backtracking is finished for *all* of those *depend* instances.

While we have cleared up the basic mechanics of dependencies, one mystery remains: what are those "dummy" strings being passed to *provide* and *depend*, and why did we pass a list to *depend*? We call these *dependency identifiers*; they define which bindings between *provide* and *depend* statements are permitted. In general, the argument to the *depend* statement is a list of zero of more dependency identi-

fiers; *depend* continuously attempts to bind itself to a *provide* whose dependency identifier is one of the identifiers in the list. Here, the order of identifiers in the list matters - *depend* always prefers *provide* statements whose dependency identifiers appear earlier in the list. Even when *depend* is already bound to a *provide*, if a more desirable *provide* becomes available, *depend* will trigger backtracking in the hope that it will be able to bind itself to the more desirable *provide*.

Finally, the *depend* statement behaves as a gateway into the scope of the *provide* it is bound to; see Listing 3.15. This ability is essential to most practical uses; without it, there would be no way to pass any kind of information through the dependency.

Listing 3.15

```
process main {
    depend_scope() depsc;
    process_manager() mgr;
    mgr->start("test1", {});
    mgr->start("test2", {});
}
template test1 {
    var("hello") x;
    _caller.depsc->provide("dummy");
}
template test2 {
    _caller.depsc->depend({"dummy"}) dep;
    println(dep.x);
}
```

Chapter 4

Imperative programming

4.1 Imperative event handling

While we have seen many statements that report various kinds of events, the events were always integrated with the NCD execution model. But that is all right, since it allowed us to handle the events in an elegant way. For example, using *net.ipv4.addr* some time after *net.backend.waitlink* automatically makes sure that when the link goes down on a network interface, the assigned IP address is removed, which was the intended behavior. But there are still some kinds of events that need to be handled more explicitly, in particular ones where there is no meaningful *opposite* event that would cause a statement to trigger backtracking. For example, suppose we would like to observe keyboard events, that is, pressing and releasing of keys. On first look it may seem simple - after all, the opposite event to pressing a key is releasing that key. Therefore we may be tempted to implement keyboard observation as a statement which observes the state of a particular key. However, such an implementation would be extremely inefficient in general, since we would have to create one instance of this hypothetical key observation statement for each of all possible keys.

However, statements in NCD which explicitly report events still employ the backtracking mechanism. They generally operate like this: as soon as an event can be reported, they report initialization completion, and expose details about the event through variables. Below the event-reporting statement, the program-

mer writes imperative code which handles events; at the end of this code, the programmer invokes the *nextevent* method-like statement on the event-reporting statement to indicate that the event has been handled. This invocation causes the event-reporting statement to trigger backtracking; the sequence then repeats, as the statement waits for another event.

Our demonstration code is based on the *sys.evdev* statement[32], which can report events from any *event device* on Linux, including keyboards[31]. This statement needs to be provided with a device node corresponding to the event device which is to be observed. To keep the code simple, we will just assume that a valid keyboard device has been selected before the program is started. However, NCD does in fact provide a facility for observing the set of all event devices in a system, in the form of the *sys.watch_input*[34] statement; proper application of this statement, which itself is also a generic event-reporting statement, would allow us to receive events from all keyboards, including ones that have been subsequently plugged in.

The program in Listing 4.1 will print out every event produced by the selected event device. The *type*, *value* and *code* variables precisely describe an input event; their exact meaning corresponds to definitions in the Linux kernel source code [33]. Keep in mind the importance of the *nextevent* invocation; without it, the program would only ever print one event.

Listing 4.1

```
process main {
    sys.evdev("/dev/input/your_keyboard") evdev;
    println(evdev.type, " ", evdev.value, " ", evdev.code);
    evdev->nextevent();
}
```

Before we conclude with event handling, we have yet to explain how in particular events can be handled. To that end, Listing 4.2 shows a program which enables a network interface when the key *F12* is pressed, and disables it when *F11* is pressed. It uses *process_manager* to manage a process dealing with the network interface, together with a feature of *process_manager* which allows us to give special identifiers to managed processes and subsequently request their termination.

Listing 4.2

```
process main {
    process_manager() mgr;
    var({"EV_KEY", "1", "KEY_F12"}) f12_pressed_event;
    var({"EV_KEY", "1", "KEY_F11"}) f11_pressed_event;
    sys.evdev("/dev/input/your_keyboard") evdev;
    var({evdev.type, evdev.value, evdev.code}) event;
    val_equal(event, f12_pressed_event) is_f12;
    val_equal(event, f11_pressed_event) is_f11;
    If (is_f12) {
        # Three-argument start() form; first argument is identifier
        # as understood by stop() below.
        mgr->start("interface-proc-id", "interface", {});
    } Elif (is_f11) {
        # Request termination of process.
        mgr->stop("interface-proc-id");
    };
    evdev->nextevent();
}
template interface {
    println("Interface enabled.");
    rprintln("Interface disabled.");
    # network interface configuration goes here
}
```

The program works correctly, though we would have to clear up the precise semantics of *start* and *stop* to be sure about that. For example, invoking *start* for a process identifier that already exists has no effect, and so has invoking *stop* for a process identifier which does not exist.

4.2 Imperative loops

There is still a piece missing before we can rightfully claim that NCD implements an extension of imperative programming: *iteration*, that is, the ability to execute a list of statements until some boolean condition is satisfied[26]. Technically, we can already do that with a combination of the *If* clause and recursive use of the *call* statement. For example, in Listing 4.3, *call* recursion is used to count from zero to nine, in combination with the *If* clause, as well as the *num_lesser* and *num_add* statements[28].

Listing 4.3

```
template count_up {
    alias("_arg0") current;
    alias("_arg1") end;
    num_lesser(current, end) not_yet_done;
    If (not_yet_done) {
        println(current);
        num_add(current, "1") next;
        call("count_up", {next, end});
    };
}
process main {
    call("count_up", {"0", "10"});
}
```

But this program is inefficient in terms of memory use, which is proportional to the number it is counting up to. This inefficiency is present because each process that has been created but has not yet terminated occupies some memory in the interpreter. When a *count_up* process is about to be created with some specific value for the *current* argument, *current-1* such processes have already been created without having terminated. Therefore, we are looking for an alternative way of performing iteration, one where memory usage does not grow with an increasing number of iterations.

The solution will, of course, involve backtracking. In fact, we have already seen the solution, in the previous section about event-reporting statements. There, a sequence of statements was being repeatedly initialized and deinitialized, once for each event being reported. This is what we care for, except for the part about reporting events; in order to perform unlimited iteration, our "event-reporting statement" would have to act like a source of unlimited quantities of events. The *backtrack_point* statement[37] does precisely that, as demonstrated in Listing 4.4. There is a minor difference though; instead of the *nextevent* method-like statement, *go* needs to be used in order to proceed to the next iteration.

Listing 4.4

```
process main {
    var("0") current;
    backtrack_point() point;
    num_lesser(current, "10") not_yet_done;
    If (not_yet_done) {
        println(current);
        num_add(current, "1") next;
        current->set(next);
        point->go();
    };
}
```

The behavior of *backtrack_point* is easy to define. When an initialization request is received for *backtrack_point*, initialization completion is immediately reported, allowing the program to proceed. On the other hand, when an initialization request is received for *backtrack_point::go*, the corresponding *backtrack_point* statement triggers backtracking and reports initialization completion once again, at the same time. This effectively prevents the program from proceeding over the *go* statement, causes it to backtrack to *backtrack_point* and resume from there. Notice how this mechanism works across the *If* clause; this is because the *If* clause (and the closely related *call* statement) is specifically engineered to interoperate with backtracking, as we have already discussed in the section about calling templates.

On the surface, the functionality offered by *backtrack_point* in NCD is not unlike the famous *goto* statement that is present in many imperative languages[35]. However, especially if the implementation of *backtrack_point* is taken into account, it can be seen as a higher-level and more structured feature than *goto*. For example, it can only be used to jump *back* to an earlier point in the program, and not without the automatic deinitialization of statements that are being jumped over.

4.3 Manual error handling

When writing code in imperative style, various error conditions can arise which are to be expected and handled in a defined manner. For these kinds of errors, the error-handling mechanism built into NCD, the one which retries statements after a timeout, may not suffice. Let us say we wish to read a file, while explicitly handling any kind of error that can occur with regard to that, such as the file not existing, an access restriction or hardware failure. The code is in Listing 4.5; there, the *file_open* statement[36] is used, which implements file input/output and allows for a certain degree of manual error handling.

Listing 4.5

```
process main {
    file_open("a.txt", "r") file;
    If (file.is_error) {
        println("Error reading file!");
    } Else {
        backtrack_point() read_point;
        file->read() data;
        If (data.not_eof) {
            println("Got some data: >", data, "<");
            read_point->go();
        };
        file->close();
        println("Finished reading.");
    };
```

```
}
```

The program is not complex; it tries to open a file for reading, checks whether there was an error opening it, and if not, it proceeds to read the file in pieces until there is no more data to be read, and finally closes the file. It appears to be completely imperative. But notice the apparent lack of handling read errors; if we are not handling read errors, how does this program achieve the goal of manual error handling? Well, it turns out that we actually are handling read errors - in the same place where we're handling errors related to opening the file. That is, if an error occurs during *read* or *close*, *file_open* triggers backtracking and modifies its *is_error* variable. This causes the *If* clause to be re-evaluated, and the error is handled by printing an error message. Even though we did not directly handle every error in the place where it occurs, this kind of *coarse-grained* error reporting as implemented by *file_open* and some other statements is sufficient for most practical purposes, where it significantly simplifies the handling of errors. To some degree, this mechanism allows the programmer to *assume* that once the file has successfully been opened, there will not be an error reading it; though some consideration is still necessary with respect to external side effects of the code reading and processing the data.

4.4 Manipulation of values

We have already seen how the contents of a list or map value can be examined or processed individually using the *Foreach* clause. Here, we will go further; we will show how to construct, examine and modify values imperatively. In NCD, such complex operations on values have been delegated to the *value* statement[38] and related method-like statements, see Listing 4.6. The *value* statement itself is initialized exactly like the *var* statement, that is, it accepts a single argument which specifies the initial stored value. The *value* statement exposes its stored value via the empty-name variable; it also exposes the additional variables *type*, *length* and *keys*. The *type* variable is a string describing the type of the value ("string", "list" or "map"); *length* is the number of elements in a list, the number of entries in a map, or the number of bytes in string, encoded as a decimal string;

keys is a list of all the keys in a map, when the value is a map.

Listing 4.6

```
process main {
    value({"Hello", "Values"}) v;
    to_string(v) str;
    println(str, v.type, v.length);
}
```

Unlike *var*, *value* deconstructs its argument into an internal reference-counted tree structure. The *value::get* method-like statement is used to retrieve the element of a list based on an index, as well as the value corresponding to a key in a map; this is shown in Listing 4.7. *get* itself acts as the result of the retrieval - it exposes the same variables as *value* and permits invocation of the same method-like statements. However, *get* does not copy the result; rather, it creates an internal reference to the result within the reference-counted tree structure. As such, its performance does not depend on the size of the result.

Listing 4.7

```
process main {
    value({"a", "b"}) v;
    v->get("1") w; # retrieves the "b"
    println(w, w.type, w.length);
    value(["h":"hello"]) q;
    q->get("h") r; # retrieves the "hello"
}
```

If the list index or map key passed to *get* does not exist, a statement error is raised. If this is not desired, the *value::try_get* statement should be used instead, which behaves much like *get*, except that it does not trigger a statement error if the requested element does not exist. Instead, existence is reported via the *exists* variable, which is either “true” or “false”. In case the element does not exist, any other variables will not be available and must not be read, including the empty-name variable which would otherwise expose the result. Obviously, the *If* clause

can be used in combination with the *exists* variable to perform different actions based on the existence of the element.

Values referenced by *value* and related statements can also be modified; this is demonstrated in Listing 4.8. For example, *value::replace* and *value::insert* add new list elements or map entries. The only difference between the two is that when operating on a list, *replace* will replace the element at the given index, but *insert* will insert another element in front of that index. Both of these statements serve as references to the inserted value, similarly to how *get* serves as a reference to the retrieved value. Additionally, *value::append* can be used to append elements to a list, or to append a string to a string.

Listing 4.8

```
process main {
    value(["h":"hello", "w":"world"]) v;
    v->get("h") old_h;
    v->replace("h", "good") new_h;
    new_h->append("bye");
    println(old_h, new_h); # hellogoodbye
    v->get("h") h;
    println(h); # goodbye
}
```

Keep in mind that while *insert* and *replace* preserve the identity of the value they are modifying, the new value that they insert has its own new identity; if an existing value was replaced in the process, it will continue to exist unmodified as long as any references to it remain. In the above program, *old_h* is an example of such a reference; it continues to reference the string "hello" even after the value corresponding to the key "h" in the map *v* was replaced.

4.5 Blocker statement

The *blocker* statement[39] makes it possible to imperatively control process execution more precisely than *backtrack_point*. The *blocker* statement itself initializes immediately and remains passive; it only serves as a kind of scope for its

method-like statements operate in, similarly to *process_manager* and *depend_scope*. The *blocker::use* method-like statement is the one which will change its state based on imperative commands. On the other end, *blocker::up*, *blocker::down* and *blocker::downup* can be used to control *blocker::use*. *up* will make *use* report initialization completion; *down* will make it trigger backtracking; *downup* will make it trigger backtracking and immediately report initialization completion. All these control statements act in an idempotent manner; for example, invoking *up* twice will result in at most one initialization completion being reported. The program in Listing 4.9 demonstrates the usage of *blocker* in order to implement a network interface that is automatically disabled once per hour for ten seconds. That is generally not a useful feature, but enough to explain *blocker* and give some inspiration for practical applications.

Listing 4.9

```
process main {
    blocker() blk;
    process_manager() mgr;
    mgr->start("interface", {});
    mgr->start("controller", {});
}
template interface {
    _caller.blk->use();
    println("Enabled.");
    rprintln("Disabled.");
    # network interface code comes here
}
template controller {
    backtrack_point() loop;
    _caller.blk->up();
    sleep("3590000");
    _caller.blk->down();
    sleep("10000");
    loop->go();
}
```

Chapter 5

Conclusion

In its current state, the NCD programming language can be used to manage wired and wireless[40] network interfaces on Linux, and possibly other kinds of interfaces, if the programmer writes the appropriate support code within the NCD language. NCD also provides statements which observe the set of all network interfaces present on a system[41]; these can be exploited in combination with the process-template feature to automatically configure any network interface, including those seen for the first time. Input devices, including keyboards and mice, can be observed as well, though this capability may be more useful for purposes unrelated to network configuration[42].

Imperative programming features in NCD allow a programmer to manually implement special-purpose features, including those which were never considered by the developer of NCD. A built-in inter-process communication (IPC) mechanism makes it possible for an external program to connect to an NCD program and communicate with it, using a custom protocol based on NCD-values[43]. This IPC mechanism greatly increases the general usability of the language; for example, it enables the implementation of a graphical user interface communicating with a backend written in NCD[44].

If NCD is directly compared to various network-configuration tools for Linux, such as *NetworkManager*[2], *Wicd*[3] or *OpenRC*[4], it appears to lack many features, even some essential ones. However, because NCD is not a network-configuration tool, but a programming language designed for implementing such tools, the comparison is generally unfair. NCD is designed to provide a sufficient set of primitive

features which programmers can use to implement their requirements upon, without providing complex features itself.

When compared to general-purpose programming languages, NCD is, disregarding the backtracking feature, a primitive and verbose language. For example, it is impossible to directly use a statement within a statement argument, like "*If (val_equal(a, b))*". Instead, we have to explicitly declare the statement whose result we need, and use its identifier when passing the result. This problem alone makes a lot of imperative code considerably larger than it could be, as well as harder to read. A possible solution is the implementation of syntactic sugar which would allow using statements within statement arguments, and the required declarations would be automatically generated. However, such a feature would introduce some problems, including the difficulty of implementing short-circuit evaluation, in case logical operators are ever added to the language.

A slightly less severe problem with programming in NCD concerns the use of the *call* statement, which many would consider bad style, especially due to the use of a string argument to specify the process template being called. Some kind of syntactic sugar could be implemented in order to make calling process templates look better and closer to how functions are called in various imperative languages. However, if the syntax was the same as for statement declarations, care would need to be taken to avoid a process-template call from being mistaken for a statement declaration, and the other way, in cases where the name of a process template is also the name of a statement.

More invasive changes may also be implemented in the language in order to bring it closer to common imperative languages, or a new language may be built based on the variant of backtracking that is present in NCD. This may involve redefining the concept of a *value*, possibly introducing special reference values. Reference values could simplify the calling of process templates by avoiding the need to simulate references through the *_caller* gateway. Another possible change is the implementation of variables as part of the core language.

Generally, the seeming inferiority of the NCD language when directly compared to existing languages is to some degree intentional, in order to keep the interpreter simple and correct. New features are only added after careful analysys of their effects on the implementation of the interpreter as well as on programs written in

the language.

Last but not least, the author of the NCD language, as well as this thesis, considers the language design solid and its implementation of high quality. In fact, he has been using NCD programs to manage the network configuration on his personal computer for more than two years, and on his home router for about six months, and he continues to be impressed with the expressive power of the language and the robustness of its implementation.

Bibliography

[1] "NCD - scripting language for network configuration and much more", BadVPN, Google Code project hosting. https://code.google.com/p/badvpn/wiki/NCD

[2] "NetworManager - Linux Networking made Easy", Gnome Projects. http://projects.gnome.org/NetworkManager/

[3] "wicd - home", Wicd, SourceForge. http://wicd.sourceforge.net/

[4] "OpenRC", Gentoo Linux Projects. http://www.gentoo.org/proj/en/base/openrc/

[5] "Re: new ip settings method acceptibility", NetworkManager mailing list. https://mail.gnome.org/archives/networkmanager-list/2012-August/msg00080.html

[6] "ncd: major rework, make NCD into a much more general system", BadVPN, Google Code project hosting. https://code.google.com/p/badvpn/source/detail?r=213

[7] "Link layer", Wikipedia. http://en.wikipedia.org/wiki/Link_layer

[8] "var.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/var.c

[9] "net_ipv4_dhcp.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/net_ipv4_dhcp.c

[10] "net_ipv4_addr.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/net_ipv4_addr.c

[11] "to_string.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/to_string.c

[12] "print.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/print.c

[13] "sleep.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/sleep.c

[14] "net_backend_waitdevice.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/net_backend_waitdevice.c

[15] "net_backend_waitlink.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/net_backend_waitlink.c

[16] "Exception handling syntax", Wikipedia. http://en.wikipedia.org/wiki/Exception_handling_syntax

[17] "Computer multitasking", Wikipedia. http://en.wikipedia.org/wiki/Cooperative_multitasking

[18] "Event-driven programming", Wikipedia. http://en.wikipedia.org/wiki/Event_driven_programming

[19] "C language", Wikipedia. http://en.wikipedia.org/wiki/C_language

[20] "Function (computer science)", Wikipedia. http://en.wikipedia.org/wiki/Function_(computer_science)

[21] "process_manager.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/process_manager.c

[22] "alias.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/alias.c

[23] "concat.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/concat.c

[24] "JavaScript eval() Function", W3Schools.. http://www.w3schools.com/jsref/jsref_eval.asp

[25] "call2.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/call2.c

[26] "Structured program theorem", Wikipedia. http://en.wikipedia.org/wiki/Structured_program_theorem

[27] "Syntactic sugar". Wikipedia. http://en.wikipedia.org/wiki/Syntactic_sugar

[28] "valuemetic.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/valuemetic.c

[29] "foreach.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/foreach.c

[30] "depend_scope.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/depend_scope.c

[31] "Linux Input drivers v1.0", Linux kernel documentation. https://www.kernel.org/doc/Documentation/input/input.txt

[32] "sys_evdev.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/sys_evdev.c

[33] "include/uapi/linux/input.h", Linux kernel source code. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/input.h

[34] "sys_watch_input.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/sys_watch_input.c

[35] "goto", Wikipedia. http://en.wikipedia.org/wiki/Goto

[36] "file_open.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/file_open.c

[37] "backtrack.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/backtrack.c

[38] "value.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/value.c

[39] "blocker.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/blocker.c

[40] "NCD_examples", NCD online documentation. https://code.google.com/p/badvpn/wiki/NCD_examples

[41] "net_watch_interfaces.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/net_watch_interfaces.c

[42] "NCD_events_example", NCD online documentation. https://code.google.com/p/badvpn/wiki/NCD_events_example

[43] "sys_request_server.c", NCD implementation source code. https://code.google.com/p/badvpn/source/browse/trunk/ncd/modules/sys_request_server.c

[44] "ncdgui - User-friendly network configuration for Linux", Google Code project hosting. https://code.google.com/p/ncdgui/