1

Introduction

This book is all about a computer programming language called AgentSpeak, and a particular implementation of AgentSpeak called *Jason*. The AgentSpeak language is intended for developing *multi-agent* systems. Before we start to investigate how to program with AgentSpeak, it seems appropriate to try to understand in more detail what multi-agent systems are, and some of the ideas that underpin the language.

1.1 Autonomous Agents

To better understand what we mean by the terms 'agent' and 'multi-agent systems', let us consider how agents relate to other types of software. Start by considering *functional* programs, which are possibly the simplest type of software from the point of view of software development and software engineering. A functional program takes some input, chews over this input, and then on the basis of this, produces some output and halts. A compiler is an example of such a program: the input is some source code (e.g. a `.java` file), and the output is bytecode (`.class` files), object code or machine code. When we learn how to program, the types of program we typically construct are of this type: the sorts of exercises we set to programmers in an introductory Java class are things like 'read a list of numbers and print the average'. Functional programs are so called because, mathematically, we can think of them as functions $f : I \rightarrow O$ from some domain I of possible inputs (source code programs, in our 'compiler' example) to some range O of possible outputs (bytecode, object code, etc). We have a range of well-established techniques for developing such programs; the point is that, from the standpoint of software development, they are typically straightforward to engineer.

Programming Multi-Agent Systems in AgentSpeak using Jason R.H. Bordini, J.F. Hübner, M. Wooldridge

Unfortunately, many programs do not have this simple input – compute – output operational structure. In particular, many of the systems we need to build in practice have a 'reactive' flavour, in the sense that they have to *maintain a long-term, ongoing interaction with their environment*; they do not simply compute some function of an input and then terminate:

> Reactive systems are systems that cannot adequately be described by the *relational* or *functional* view. The relational view regards programs as functions . . . from an initial state to a terminal state. Typically, the main role of reactive systems is to maintain an interaction with their environment, and therefore must be described (and specified) in terms of their on-going behavior . . . [E]very concurrent system . . . must be studied by behavioral means. This is because each individual module in a concurrent system is a reactive subsystem, interacting with its own environment which consists of the other modules. [77]

Examples of such programs include computer operating systems, process control systems, online banking systems, web servers, and the like. It is, sadly, a well-known fact that, from the software development point of view, such *reactive* systems are *much* harder to correctly and efficiently engineer than functional systems.

A still more complex class of systems is a subset of reactive systems that we will call *agents*. An agent is a reactive system that exhibits some degree of *autonomy* in the sense that we delegate some task to it, and the system itself determines how best to achieve this task. We call such systems 'agents' because we think of them as being active, purposeful producers of actions: they are sent out into their environment to achieve goals for us, and we want them to actively pursue these goals, figuring out for themselves how best to accomplish these goals, rather than having to be told in low-level detail how to do it. We can imagine such agents being delegated a task like booking a holiday for us, or bidding on our behalf in an online auction, or cleaning our office space for us, if they are robotic agents.

1.2 Characteristics of Agents

Let us try to be a little more precise about what sorts of properties we are thinking of when we talk about agents. We consider agents to be systems that are *situated* in some *environment*. By this, we mean that agents are capable of *sensing* their environment (via *sensors*), and have a repertoire of possible *actions* that they can perform (via *effectors* or *actuators*) in order to modify their environment. The key question facing the agent is how to go from sensor input to action output: how to *decide what to do* based on the information obtained via sensors. This leads to the

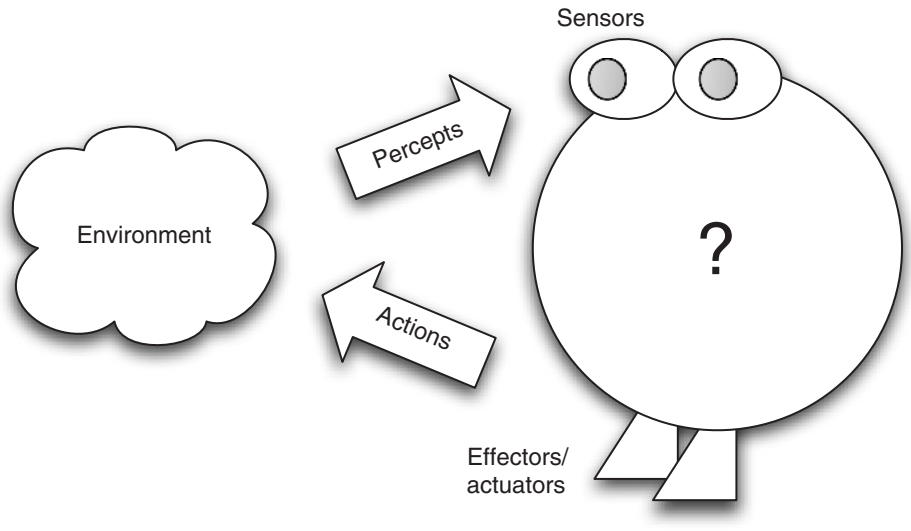

Figure 1.1 Agent and environment (after [82, p. 32]).

view of an agent as shown in Figure 1.1. As we will see, in AgentSpeak, deciding what to do is achieved by manipulating *plans*.

The environment that an agent occupies may be physical (in the case of robots inhabiting the physical world) or a software environment (in the case of a software agent inhabiting a computer operating system or network). We think of decisions about what action to perform being translated into actual actions via some mechanism external to the agent; usually, this is achieved via some sort of API. In almost all realistic applications, agents have at best *partial* control over their environment. Thus, while they can perform actions that change their environment, they cannot in general completely control it. Very often this is because there will be other agents in the environment, who exhibit control over their part of the environment.

Apart from being situated in an environment, what other properties do we expect a rational agent to have? Wooldridge and Jennings [104] argued that agents should have the following properties:

- autonomy;
- proactiveness;
- reactivity; and
- social ability.

Autonomy

It is important to realise that autonomy is a very broad spectrum. At one end of the spectrum, we have computer programs such as conventional word processors and spreadsheets, which exhibit little or no autonomy. Everything that happens with such an application happens because you make it happen – you select a menu item, or click on an icon, for example. Such programs, by and large, do not *take the initiative* in any sense. At the other end of the autonomy spectrum are you and us. You are completely autonomous. You can ultimately choose to believe what you want, and do what you want – although society typically constrains your autonomy in various ways, preventing you from doing certain things, for the sake of you and your peers. You have your own goals, your own agenda, and autonomy means they really are yours: nobody and nothing explicitly dictates them to you. (Of course, you might argue that society tries to shape our beliefs and goals, but that is another story.) In this book, we are interested in computer programs that lie somewhere between these two extremes. Roughly speaking, we want to be able to *delegate* goals to agents, which then decide how best to act in order to achieve these goals. Thus, our agent's ability to construct goals is ultimately bounded by the goals that we delegate. Moreover, the way in which our agents will act to accomplish their goals will be bounded by the *plans* which we give to an agent, which define the ways in which an agent can act to achieve goals and sub-goals. One of the key ideas in AgentSpeak is that of an agent putting together these plans on the fly in order to construct more complex overall plans to achieve our goals.

At its simplest, then, autonomy means nothing more than being able to operate independently in order to achieve the goals we delegate to an agent. Thus, at the very least, an autonomous agent makes independent decisions about how to achieve its delegated goals – its decisions (and hence its actions) are under its own control, and are not driven by others.

Proactiveness

Proactiveness means being able to exhibit *goal-directed* behaviour. If an agent has been delegated a particular goal, then we expect the agent to try to *achieve* this goal. Proactiveness rules out entirely passive agents, who never try to do anything. Thus, we do not usually think of an object, in the sense of Java, as being an agent: such an object is essentially passive until something invokes a method on it, i.e. tells it what to do. Similar comments apply to web services.

Reactiveness

Being *reactive* means being *responsive* to changes in the environment. In everyday life, plans rarely run smoothly. They are frequently thwarted, accidentally or

deliberately. When we become aware that our plans have gone wrong, we respond, choosing an alternative course of action. Some of these responses are at the level of 'reflexes' – you feel your hand burning, so you pull it away from the fire. However, some responses require more deliberation – the bus has not turned up, so how am I going to get to the airport? Designing a system which simply responds to environmental stimuli in a reflexive way is not hard – we can implement such a system as a lookup table, which simply maps environment states directly to actions. Similarly, developing a purely goal-driven system is not hard. (After all, this is ultimately what conventional computer programs are: they are just pieces of code designed to achieve certain goals.) However, implementing a system that achieves an effective *balance* between goal-directed and reactive behaviour turns out to be hard. This is one of the key design objectives of AgentSpeak.

Social Ability

Every day, millions of computers across the world routinely exchange information with humans and other computers. In this sense, building computer systems that have some kind of social ability is not hard. However, the ability to exchange bytes is not social ability in the sense that we mean it. We are talking about the ability of agents to *cooperate* and *coordinate* activities with other agents, in order to accomplish our goals. As we will see later, in order to realise this kind of social ability, it is useful to have agents that can communicate not just in terms of exchanging bytes or by invoking methods on one another, but that can communicate at the *knowledge level*. That is, we want agents to be able to communicate their *beliefs*, *goals* and *plans* to one another.

1.3 Multi-Agent Systems

So far, we have talked about agents occupying an environment in isolation. In practice, 'single agent systems' are rare. The more common case is for agents to inhabit an environment which contains *other* agents, giving a *multi-agent system* [103]. Figure 1.2 gives an overview of a multi-agent system. At the bottom of the figure, we see the shared environment that the agents occupy; each agent has a 'sphere of influence' in this environment, i.e. a portion of the environment that they are able to control or partially control. It may be that an agent has the unique ability to control part of its environment, but more generally, and more problematically, we have the possibility that the spheres of influence overlap: that is, the environment is *jointly* controlled. This makes life for our agents more complicated, because to achieve an outcome in the environment that our agent desires, it will have to take into account how the other agents with some control are likely to act.

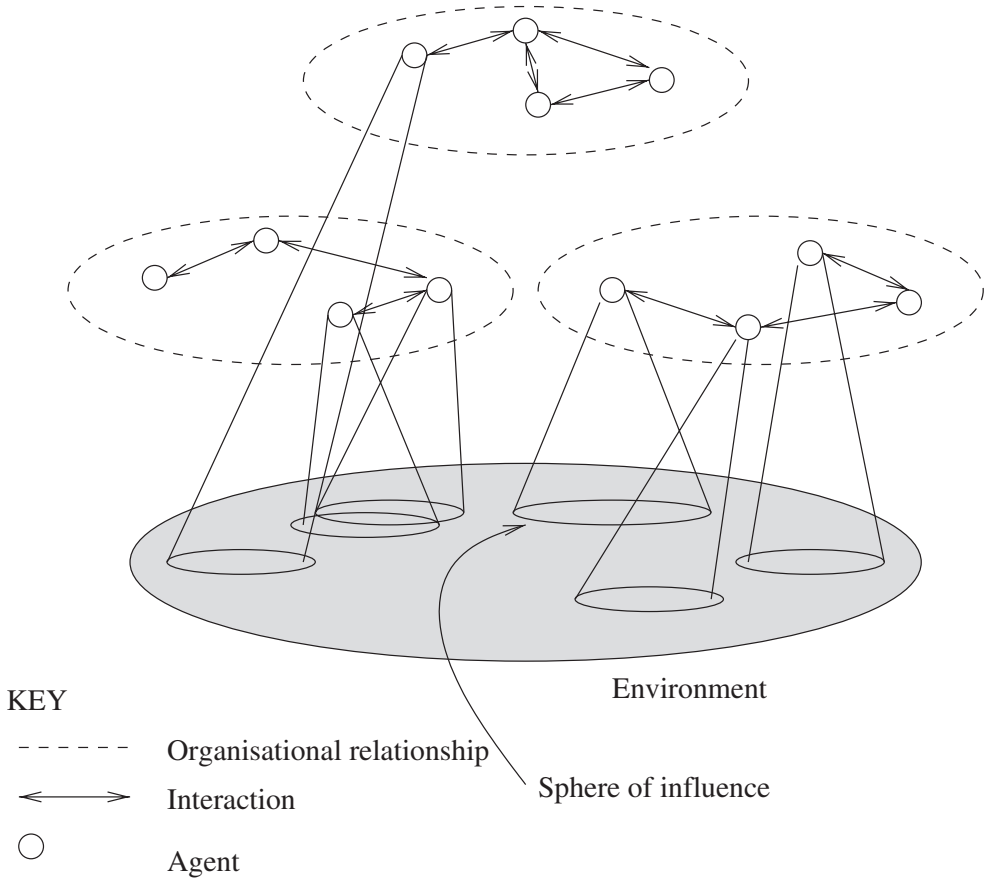

Figure 1.2 Typical structure of a multi-agent system (after [58]).

Above the environment, we see the agents themselves, which stand in various organisational relationships to one another (for example, one agent may be the peer of another, or may have line authority over another). Finally, these agents will have some knowledge of each other, though it may be the case that an agent does not have complete knowledge of the other agents in the system.

Programming Languages for Agents and Multi-Agent Systems

We now have some idea of what kinds of properties we are thinking of in our agents. So, suppose we want to program these things: what do these properties tell us about the kinds of programming language or environment that we might use for programming autonomous agents? We can identify the following requirements:

- The language should support delegation at the level of goals. As we noted earlier, when we delegate a task to an agent, we do not generally want to do this by giving the agent an executable description of what to do. Rather, we want to communicate with it at the level of goals: we should be able to describe our goals to an agent in a high-level way, independent of approaches to achieving these goals.
- The language should provide support for goal-directed problem solving. We want our agents to be able to act to achieve our delegated goals, systematically trying to achieve them.
- The language should lend itself to the production of systems that are responsive to their environment.
- The language should cleanly integrate goal-directed and responsive behaviour.
- The language should support knowledge-level communication and cooperation.

These are the main requirements that AgentSpeak and ***Jason*** are intended to fulfill. In the following section, we will give a very brief introduction to AgentSpeak, which will give a feel for how some of these features are provided.

1.4 Hello World!

When introducing a new programming language, it has become the tradition to give a short example program, the purpose of which is simply to display the text 'Hello World!' to the programmer.[1] For example, here is a 'Hello World' program in Java:

```
public class HelloWorld {
   public static void main( String args[] ) {
      System.out.println( "Hello World!" );
   }
}
```

Trivial though they are, running a 'Hello World' program helps to give a programmer confidence with the new language, and very often they give a useful insight

[1]There are even web sites devoted to 'Hello World' programs: `http://www.roesler-ac.de/wolfram/hello.htm` is one example, with 'Hello World' programs for hundreds of languages.

into the 'mind set' of the language. Therefore, we strongly encourage you to try this exercise:

```
started.

+started <- .print("Hello World!").
```

Let us try to understand a little of what is going on here, although we will save the details for later.

The first thing to understand is that this constitutes the definition of a single agent. This definition will often be saved in a single file, and let us suppose that on our system we have called this file `hello.asl`; the `.asl` extension will be used for all our AgentSpeak programs. Now, the definition of an agent in AgentSpeak consists of two parts:

- the agent's *initial beliefs* (and possibly *initial goals*); and
- the agent's *plans*.

The first line defines one initial belief for our agent. (Although there is only one initial belief here, we could have given a list.) AgentSpeak does not have variables as in programming languages such as Java or C; the constructs are specific agent notions, such as beliefs, goals and plans. We need 'beliefs' because the intuition is that they represent the information that the agent has currently been able to obtain about its environment. The full stop, '.', is a syntactic separator, much as a semi-colon is in Java or C. Therefore, when our agent first starts running, it will have the single belief `started`; intuitively, it has the belief that it has started running. Notice that there is no magic in the use of the term '`started`'; we could just as well have written the program as follows, and we would have obtained exactly the same result:

```
cogitoErgoSum.

+cogitoErgoSum <- .print("Hello World!").
```

So, while its important and helpful for programmers to pick sensible and useful names for beliefs, the computer, of course, could not care less. One rule worth remembering at this stage is that beliefs must start with a lowercase letter.

Next, we have the line

```
+started <- .print("Hello World!").
```

which defines a plan for the agent – which is in fact the only plan this agent has. Intuitively, we can read this plan as meaning

> whenever you come to believe 'started', print the text 'Hello World!'.

The plan, like all AgentSpeak plans, comes in three parts. The first part is a *triggering event*. In this case, the triggering event is simply

```
+started
```

The symbol '+' in this context means 'when you acquire the belief . . . ', and so overall the triggering condition is 'when you acquire the belief "started" '. We know from the above discussion that the agent acquires this belief when it starts executing, and so in sum, this plan will be triggered when the agent starts executing.

However, what does it mean, to trigger a plan? The idea is that the trigger of a plan defines the *events* that it is useful for handling. In this case, the event is the acquisition of a particular new belief. A plan is triggered when events occur which match its trigger condition, and when this happens, the plan becomes 'active'; it becomes something that the agent 'considers doing'. However, before an agent selects a plan to become active, it checks that the *context* of the plan is appropriate. In the hello world case, the context is in fact empty, which can be understood as meaning 'this plan is always good'. As we will see later, in the context part of plans, we can define complex conditions which an agent uses to determine whether or not to choose a particular plan for a given event.In particular, an agent can have *multiple* plans triggered by the *same* event which deal with this event in *different* ways: thus an agent can have multiple different responses to events, and can choose between these depending on the situation in which it currently finds itself.

In this case, there is just one plan that can be triggered by the event, and since the context is empty, it is always applicable, and so the AgentSpeak interpreter *directly executes* the body of that plan. In this case, the body of the plan is very simple, containing a single action:

```
.print("Hello World!").
```

As you might guess, the effect of this action is simply to display the text 'Hello World!' on the user's console. Running this example on ***Jason***, the result is that the following gets displayed in the user's console (see Figure 1.3):

```
[hello] saying: Hello World!
```

There are several points to note here. First, although `.print(...)` looks like a belief, it is in fact an action, as it appears in the plan body; to give the reader some syntactic clues when reading a ***Jason*** program, 'internal' actions begin with a full stop (actions normally change the environment, but not *internal* actions). In fact, `.print(...)` is a pre-defined internal action in ***Jason***: other pre-defined actions

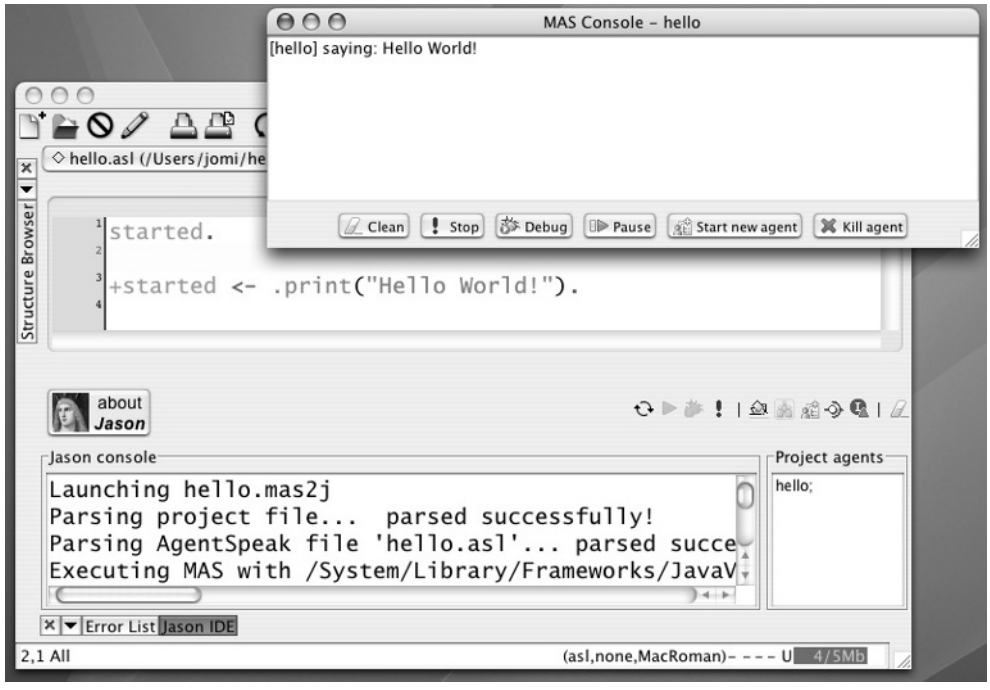

Figure 1.3 Running the 'Hello World!' program with ***Jason***.

include, for example `.send(...)` and `.broadcast(...)` for agent communication, and `.stopMAS`, to cause a halt to the execution of the multi-agent system. Appendix A describes other internal actions available in ***Jason***.

Don't be misled by the very simple structure of this example into thinking that plan bodies are just like sub-routines, methods or procedures from conventional programming languages: they are in fact very much more than that. For example, one of the key ideas in AgentSpeak is that we can have a *goal* in a plan body. The idea is that, when the interpreter comes to a goal in a plan body, it tries to find a plan that achieves this goal; as we discussed above, there may be several such plans, which may or may not match the current context. The upshot of this is that it is possible, within AgentSpeak, to invoke code *with reference to the effect of the code*, and the same invocation, in different contexts, results in different code being invoked. This is a very substantial difference to conventional languages like Java or C.

Finally – and this may at first sight seem somewhat perverse to those with a grounding in the theory of computing – if you run this example, you will see that *the program does not terminate!* We are used to thinking of non-termination (infinite loops and the like) as a bad thing. In the case of AgentSpeak, however, we

are after agents that are *aware of* and *respond to* their environment. It is therefore quite natural that they should not terminate: in fact, if ever the belief `started` was deleted and then added to our agent's belief base again, then the agent would fire into life, once again printing the 'Hello World!' text. Of course, this does not happen in this example; the agent patiently watches for events that never occur. Programmers familiar with the Prolog language might contrast this behaviour with that of Prolog programs: a Prolog program never does anything until a user asks a query, and the behaviour that the program then generates is a side-effect of trying to prove a theorem.

Another Simple Example: Computing the Factorial

To conclude, we will present another short program, which does something marginally more useful than just displaying some text; it is again a classical example rather than a typical agent program such as those shown later in the book. Specifically, the code computes the factorial[2] of 5. The point about the factorial example is that it illustrates how loops' work, and for some languages, which make heavy use of recursion, the factorial example can be very illustrative. We will start with an example which computes the factorial, even though it is not a very elegant example of AgentSpeak code:

```
fact(0,1).

+fact(X,Y)
   :  X < 5
   <- +fact(X+1, (X+1)*Y).

+fact(X,Y)
   :  X == 5
   <- .print("fact 5 == ", Y).
```

As before, the first line defines the agent's initial belief: the belief `fact(0,1)` means that the agent believes the factorial of `0` is 1. In this example, a belief of the form `fact(X,Y)` will mean that the factorial of `X` is `Y`. So, when the agent starts executing, this belief is added to the agent's belief base.

We have two plans in this example: they have the same triggering condition, but different contexts. The first plan may be explained as follows:

> whenever you acquire the belief that the factorial of X is Y, where $X < 5$, add to your beliefs the fact that the factorial of $X + 1$ is $(X + 1) * Y$.

[2] If n is a positive integer, then the factorial of n, written $n!$, is $n * (n - 1) * (n - 2) * \cdots * 2 * 1$; for convenience, we define $0! = 1$. We know that *you* know what factorial means, this is for those other people who do not know.

The second plan can be understood as follows:

> whenever you acquire the belief that the factorial of 5 is Y, print out the value of Y.

Therefore, when the agent starts executing, and the belief `fact(0,1)` is added to its belief set, the first plan is triggered; the context condition is satisfied, (since $0 < 5$), and so the agent then adds the belief `fact(1,1)`; this again triggers the plan, resulting in the belief `fact(2,2)` being added, the plan again fires, resulting in `fact(3,6)` being added, and so on. Notice that the second plan will not become active as a consequence of these events, since the context part (`X == 5`) is not satisfied. Eventually, the belief `fact(5,120)` is added, and at this point, the context condition of the first plan is not satisfied, while the context of the second plan is. As a result, when running this example with ***Jason***, we get the following displayed on the console:

```
[fact] saying: fact 5 == 120
```

This example shows how multiple plans with the same trigger condition but different contexts can respond to events in different ways.

Unfortunately, this example is rather messy with respect to AgentSpeak programming style. The problem is that it does not make use of recursion; instead, it fills up the agent's belief base with `fact(0,1)` up to `fact(5,120)` predicates. This does not matter so much in this case, but in general, this is not an efficient way of doing things in AgentSpeak. So let us consider a slightly more complex, but much more efficient, version of the program.

```
!print_fact(5).

+!print_fact(N)
   <- !fact(N,F);
      .print("Factorial of ", N, " is ", F).

+!fact(N,1) : N == 0.

+!fact(N,F) : N > 0
   <- !fact(N-1,F1);
      F = F1 * N.
```

The first line defines the agent's initial goal: the exclamation mark indicates that this is a goal to be achieved. Therefore, the initial goal of the agent is to print the factorial of 5. This can be achieved by first calculating the factorial of `N` and then printing out the result, as stated in the first plan.

We have two more plans in this example, both for calculating the factorial. The first of them may be explained as follows:

> whenever you acquire the goal to calculate the factorial of `0`, this is known to be 1, therefore nothing else needs to be done.

The second plan can be understood as follows:

> whenever you acquire the goal to calculate the factorial of `N`, provided `N` $>$ `0`, the course of action to take is to first calculate the factorial of `N` $-$ 1, and then multiply that value by `N` to get the factorial of `N`.

When running this example with ***Jason***, we get the following displayed on the console:

```
[fact] saying: Factorial of 5 is 120
```