Chapter 2 Exercises Solutions

Question	Answer
2.1.a	Example 5. Let us explore the concepts of syntax and semantics using an example. Consider a simple language for controlling mobile robots, with the uninspiring name robot. It loosely follows the principle of reactive control, a specific way of controlling the behavior of robots.7 An example model can be found in Fig. 2.2. There are two key aspects that organize models in this hypothetical language: modes of operation and flows between modes—continuations. We have four modes in the example model: RandomWalk, MovingForward, Avoid, and ShutDown. The modes can be nested. The last three modes are nested in the first mode (RandomWalk). A mode may contain other modes, actions, and reactions. Actions resemble regular programming-language statements—they are immediately executed as the mode is activated, in the order listed. The actions in the example are: move, turn, and return to base. Reactions, introduced using the keyword on, are not executed immediately, but registered and suspended. Each reaction is triggered by an event, when it switches the mode to a new mode. The two events in the example are: obstacle and clap. Reactions are only active if their mode is active. Reactions are registered on the fly when a mode is activated, but are only handled after all actions are completed (nonpreemptively). For instance, if the robot is in the MovingForward mode and encounters an obstacle, the active mode becomes Avoid. A mode can also have a continuation mode, a successor. These are indicated using the arrow symbol (->). If a mode has a successor mode, then the control switches to it immediately after all actions have been executed. For instance, after Avoid the control moves to MovingForward. If a mode has no successor, the control stays in place, and awaits for any possible reaction triggers. In robot, one can only define a single successor for a mode. If control needs to flow to various modes as a result of execution, this can only be done by registering reactions that have different targets. The same arrow symbol (->) is also placed before the initial mode, in the context of its containing mode (see MovingForward). There must be exactly one initial mode at each level of nesting. The syntax of a model (or a program) is what you can directly see and read. For instance, when looking at Fig. 2.2, you see the syntax of our example model. The syntax is described with phrases of the following kind: • A mode may contain other modes, actions, and reactions. • There must be exactly one initial mode at each level of nesting. The semantics of a model define what the model means: how the robot shall

	behave according to the model. For the model in Fig. 2.2, the semantics is that of a random walk. Semantics are defined over all instances of a language, but they are only implemented once for the language. Semantics regulate detailed aspects of behavior, for instance, whether modes are pre-emptive or not. If you specify in the semantics that modes are pre-emptive, this would mean that modes could be switched whenever a suitable reaction is triggered, leading to a new active mode, even when a computation is active in another mode. Statements like the following describe the semantics of robot: • Reactions are only active if their mode is active. • If a mode has a successor mode, then the control switches to it immediately after all actions have been executed. Sentences describing syntax → Highlighted as red. Sentences describing semantics → Highlighted as orange.
2.1.b	Rust offers the if statement for conditional code execution. If the condition isn't true, the code block is bypassed. Sentences describing syntax → Highlighted as red. Sentences describing semantics → Highlighted as orange.
2.2	Robot Operating System (ROS) Target Users: University students, Academic Researchers. Use cases: Executing various robotics algorithms. Most ROS APIs are highly pertinent to robotics, such as message passing, sensor data handling, and state management. Some aspects of ROS, like certain low-level operations, could potentially be abstracted or simplified by a DSL for specific tasks. ROS often requires a large amount of boilerplate code for setup and configuration, which could potentially be automated or reduced by a DSL. ROS is a suitable target for code generation due to its structured and extensive API, which could be effectively used as a backend for a DSL, simplifying complex tasks into more accessible commands.
Ex 2.3	ROS can be integrated with gtest framework and simulate the environment for testing and quality assurance support.
Ex 2.4	Domain: Chef manages the configuration and deployment of infrastructure resources on servers and workstations. This includes:

	• Installation, configuration, and updates of software packages. • Managing system settings like user accounts, network settings, and file permissions. • Starting, stopping, and restarting services on the system. • Reusable configurations for specific tasks or environments. Information Present: Chef uses a declarative model, meaning it specifies the desired state of the system rather than the steps to achieve it. This includes: • Resources: Definitions of specific infrastructure elements like packages, users, services, etc. • Attributes: Values that customize resource behavior (e.g., package name, user password). • Recipes: Collections of resources that define the configuration for a specific role or functionality. • Cookbooks: Packages of recipes that can be shared and reused across different systems. Abstracted Information: Chef hides the underlying system administration commands and scripts. Users define the "what" (the end state), while Chef determines the "how" (the process to achieve this state). Syntax Style: Chef uses a Ruby-like syntax with keywords, operators, and code blocks. This makes it relatively easy to learn for those familiar with Ruby or other scripting languages. Automated Tasks: Chef automates various infrastructure management tasks. These include: • Provisioning: Setting up new servers with the desired configuration. • Configuration Management: Applying consistent configuration across multiple systems. • Deployment: Deploying software updates and applications in a controlled manner. • Orchestration: Coordinating configuration changes across multiple servers. Information Source: Chef relies on "cookbooks" which contain the recipes and resources needed to configure the system. Source of Information for Chef 1. Attribute System: Chef uses a node attribute system that defines specific details about the nodes (servers), such as their operating system, IP addresses, and any other configuration details. These attributes can be used dynamically in recipes to adapt the configuration based on node-specific data.

	2. Data Bags: Chef uses data bags to store global variables and other data that can be accessed by recipes across different nodes. This helps in managing common data centrally and securely. 3. Role Definitions: Roles define a particular setup or configuration aim and can include multiple recipes and attributes. These help in managing common configurations across multiple nodes efficiently. 4. Environment Settings: Chef manages different settings for different environments (like production, testing, development) allowing the same recipes to behave differently based on the environment they are executed in.
Ex 2.5	a) Could not find the external dependency 'FunctionalCalculations.jar'. No such file or directory. Error Reported By: Plugin mechanism. Justification: The plugin mechanism is responsible for loading external JAR files containing calculations. If the file can't be found, it indicates an issue with the plugin mechanism itself (e.g., incorrect path or missing file). b) line 213: Expected keyword 'parameter' instead of EOL Error Reported By: Parser. Justification: The parser analyzes the configuration file and builds a model based on the syntax. Here, it encounters an unexpected end-of-line (EOL) where it expected a "parameter" keyword, suggesting a syntax error in the configuration file. c) Parameter group 'Engines' depends on itself Error Reported By: Constraints or the type system. Justification: This error indicates a circular dependency within the configuration. The constraints engine might be responsible for enforcing dependencies between parameters, or the type system might be involved if parameter groups have defined types. d) line 196: Expected an Integer value instead of String Error Reported By: The type system. Justification: The type system verifies that parameters have the correct data types. Here, it encounters a String value where an Integer value was expected. e) The enumeration type 'color' should have distinct values. Value 'pink' is repeated in lines 400 and 404. Error Reported By: The type system or constraints. Justification: Enumerations define a set of allowed values. The type system might enforce this, or specific constraints could be defined on the "color" type to ensure distinct values.

Ex 2.6	Explore using a fuzzing tool to generate random configurations with different syntax variations. This can help uncover unexpected parsing errors or edge cases not covered in manual tests.
Ex 2.7	Constraint 1: Distinct Triggers • Test Cases: ○ Valid case: A mode with multiple reactions, each having a unique trigger event (e.g., "on bump" and "on light"). ○ Invalid cases (2): ▪ Duplicate triggers within a single mode (e.g., two reactions with "on button"). ▪ Missing trigger for a reaction (empty trigger field). Constraint 2: Initial Sub-Mode • Test Cases: ○ Valid cases (2): ▪ A mode with no sub-modes (empty sub-modes list). ▪ A mode with an initial sub-mode defined. ○ Invalid case: A mode with multiple sub-modes but none marked as initial. Prioritizing Test Cases with Many Constraints: 1. focus on test cases that directly challenge each constraint to ensure they catch the intended errors. 2. covers edge cases.