Software Engineering Concepts (COMP3003)

7. Domain Specific Languages (DSLs)

Updated: 15/10/2019.

Domain Specific Languages (DSLs) resemble (or in some cases *are*) programming languages that are used in fairly specific situations. We're not just going to learn particular existing DSLs, though. We're going to learn how to *make* new ones.

(A **domain** is any one of the various real-world areas – "problem spaces", if you like – to which software engineering is applied. Different domains, although loosely defined, come with different kinds of functional and non-functional requirements, and different kinds of users.)

As an architectural decision, creating a DSL could be compared to implementing scriptability, and there's significant overlap between these ideas. However, Python, Lua, Javascript and the like are **General Purpose Languages** (GPLs). They do not make any real significant assumptions about the domain[a].

So why implement a DSL, rather than using an existing GPL? A well-designed DSL provides a more elegant way to express domain-specific logic, at the cost of being terrible for anything outside its domain. DSLs exploit the tradeoff between readability and generality (i.e. the ability to work in all situations), sacrificing the latter to improve the former.

Gradle's build logic is an example of this, for the domain of build engineering. All `build.gradle` files are written in Gradle's domain-specific language. Each construct in that language (`jar{}`, `dependencies{}`, etc.) could instead be implemented in a general-purpose language. You can observe the difference for yourself:

Listing 1: Adding external dependencies in Gradle's DSL.

```
dependencies
{
    implementation "org.apache.commons:commons-lang3:3.0"
    implementationTest "junit:junit:4.1"
}
```

[a] Javascript is overwhelmingly used in web applications, and a lot of its APIs are engineered for this purpose, but the core language is general-purpose.

Listing 2: Adding external dependencies in Java, using Gradle's API.

```java
import org.gradle.api.*;

public class MyPlugin extends Plugin<Project>
{
    @Override
    public void apply(Project proj)
    {
        DependencyHandler dh = proj.getDependencyHandler();
        dh.add("implementation", "org.apache.commons:commons-lang3:3.0");
        dh.add("implementationTest", "junit:junit:4.1");
    }
}
```

The difference between Listing 1 and Listing 2 should give you a good idea of why Gradle's DSL exists. Its `dependencies` construct hides a significant amount of syntactic complexity, while preserving all the crucial domain-specific information.

Although build engineering is highly unlikely to be done by non-technical users, DSLs in other domains can mean that non-technical users can become more involved.

1 DSL Design

To discuss how to create a DSL, it's useful to have a specific goal in mind. For this purpose, imagine that we want to create a 2D platform game along the lines of Super Mario Bros. That is, we have levels consisting of various objects in various positions in 2D space, some of which (including of course the player's character) can move and perform actions.

We could hard-code the behaviours of all these in-game objects – how they move, how they change state, if/when they create or destroy other objects, and what happens when they collide. We could also make these things scriptable, and thus create a game *framework* (as well as just a game). Creating a DSL would go one step further, and help to draw non-programmers into the game creation scene.

Before we get into the implementation details, let's look at a brief example of what a Mario DSL *could* look like:

Listing 3: Our example "Mario" language.

```
gameobject Mario
{
    when key SPACE then JUMP
    state Small
    {
        when collided with Enemy then Action.DIE
        when collided with Mushroom then transition to Big
    }
    state Big
    {
        when collided with Enemy then transition to Small
    }
}

gameobject Mushroom { ... }

gameobject Enemy { ... }
```

We're declaring a series of "`gameobject`"s, which contains a set of states and actions. Some actions are state-specific, and so occur within one of the "`state`" blocks. We use the "`when-then`" construct to link cause and effect. One such effect can be a state change, in which we refer to one of the declared states.

There is lots of additional detail we could put into both the language itself and the specific example of it given in Listing 3, but hopefully this conveys the objective. We want to be able to design and implement languages *like* this.

1.1 Language Structure = Class Structure

Making a whole new language sounds like a daunting task, especially if the purpose is to create new kinds of language constructs that don't yet exist. How can we approach this?

We must realise that code is just a hierarchy of constructs, collectively called an **abstract syntax tree** (AST). Each concept in the code (e.g. the idea of a method, if statement, expression, etc.) could be represented as a class, and each occurrence could be an object, many of which aggregate other code objects. Think about what kind of class structure you would create in order to represent source code[b].

So, having reduced *every language* to the level of a simple class structure, it's also important to realise that you can do the reverse: every class structure could form the basis of a language, if you wanted it to. Most of the time you don't, of course, but in some cases it's worth considering.

In our Mario example, putting aside the language itself, the actual implementation would probably have:

- A `GameObject` class, containing (at any one time) an instance of one of several `State` classes (e.g. as per the State Pattern);
- A class to monitor keystrokes and a class to detect collisions between objects;
- An observer/strategy pattern implementation to respond to keystrokes and collisions.

[b] It would probably be something a bit like the reflection API, although with much more detail, so as to represent executable statements and expressions.

Designing a DSL is largely a matter of choosing a new textual representation for this object structure. We are essentially creating **syntactic sugar**. We cannot achieve new functionality this way, but we *can* make existing functionality easier to use.

We'll explain how to do this by introducing two *styles* of very high-level programming, both of which can be used together, and both of which tend to fit naturally into DSLs.

1.2 The Declarative Style

In the **declarative style**, the logic of a program or script is represented as a series of declarations, rather than (or at least in preference to) a sequence of instructions. A declarative language provides a rich set of constructs for declaring *what* will happen in certain situations, rather than spelling out precisely *how* it will happen.

The difference is one of abstraction level. General-purpose languages can always describe "how" something happens, but within a specific domain, the "how" may be mostly the same from one case to another. Gradle's language – most of it – is a good example of this. Gradle knows how to look for dependencies, how to compile Java code, how to build a .jar file and a .zip file, etc. The logic for these things is the same, no matter what project you're working on. It's only the "what" that tends to differ – what the main class is, what the dependencies are, etc.[c]

Similarly, in our Mario example, we can create `state` and `when-then` declarations because the mechanics of states and state transitions are always the same. *What* those states and transitions are is the important bit.

At an implementation level, the declarative style is simply building up an object structure. A `gameobject` declaration will (probably) translate to an instance of the `GameObject` class, and a `when-then` declaration will translate to an association between an event source and an observer.

Thus, the declarative style emphasises the important information (as defined for a given domain). Since it abstracts away a lot (perhaps all) of traditional programming, it can also become accessible to non-programmers.

1.3 Fluent Interfaces

A **fluent** interface/API attempts to arrange individual lines of code into something that vaguely resembles natural sentence structures, or at least a natural flow of concepts. It tries to avoid "boilerplate" repetition, to give you a compact but easy-to-read way to perform certain actions in code.

Originally coined by Martin Fowler and Eric Evans,[1] fluent interfaces can be thought of as a way to create internal DSLs. Although they use the features of an existing language, they use them to create a new way of expressing code concepts.

Note 1:
At best, fluency makes code readable. That's it!

Let's just take a moment to put fluency into perspective. There is exactly one reason to use or develop a fluent API: to create more readable, and hence maintainable, code.

We are, emphatically, *not* doing anything to alter the capabilities of our code. Fluency is not going to make better use of resources, or be faster, or be able to do more things. Fluent code is no better or worse in those respects than any other code.

One of the key ingredients in a fluent interface/API/DSL is the **method chain**. Using this, we can perform a squence of method calls within a single statement, where each call returns an object that facilitates (and puts into context) the next call.

[c] The distinction between "what" and "how" is situation dependent. Be warned that there is some potential for confusion where one person's "what" is another person's "how", if they are thinking at different levels of abstraction.

Let's say we have an API for drawing geometric shapes on the screen. It will look like this:

Listing 4: Non-Fluent API.

```
public class Display
{
    public void addShape(Shape s) {...}
}

public abstract class Shape
{
    private int x, y; // Position
    private String colour; // Colour;
    ... // Getters and setters
}

public class Rectangle extends Shape {...}

public class Circle extends Shape {...}
```

In order to use such an API, we might do this:

Listing 5: Using a non-fluent API.

```
Display d = new Display();
Circle circ = new Circle(5, 10);
circ.setColour("red");
circ.setRadius(5);
Rectangle rect = new Rectangle(10, 5);
rect.setColour("green");
rect.setWidth(10);
rect.setHeight(5);
d.addShape(circ);
d.addShape(rect);
```

It is hopefully reasonably easy to see what we're doing, but not as easy as it could be. By altering the API a bit, we might achieve the same results with significantly more readable and compact code, like this:

Listing 6: Using a fluent API.

```
Display d = new Display();
d.circle(5, 10).radius(5).colour("red");
d.rectangle(10, 5).width(10).height(5).colour("green");
```

How is this actually done? Here are some of the tricks:

- The `circle()` method takes the place of `new Circle()` *and* `addShape()`, and similarly for `rectangle()`. These methods create the appropriate object, add it to `Display`, and also return it.
- Using the returned object reference, we can "chain" a call to any of its setters (`colour()` in this case). We can further arrange for `colour()`, `radius()`, `width` and `height` to themselves return the same object they were called on ("`this`"). This allows us to build up chains of setter methods in any combination.
- We can drop the word "set" from various method names, because the meaning is still clear without it.

This arrangement is not without cost. The API provider needs a method for each possible object, rather than just one for a common interface/superclass. Each setter also needs to return "`this`", and it's somewhat difficult to do this in an inheritance hierarchy while maintaining strict type safety. (If a superclass method returns "`this`", it will be the superclass type, which means you can't call subclass methods on it.)

Listing 7 shows what the implementation might look like (in part) for the fluent API. Note the return statements that make the method chaining possible.

Listing 7: Partial implementation of a fluent API.

```
public class Display
{
    private List<Shape> shapes = ...;

    public Circle circle(int x, int y)
    {
        Circle circ = new Circle(x, y);
        shapes.add(circ);
        return circ;
    }

    public Rectangle rectangle(int x, int y) {...}
    ...
}

public abstract class Shape {...}

public class Circle extends Shape
{
    private int radius = 1;

    public Circle(int x, int y) {...}

    public Circle radius(int radius)
    {
        this.radius = radius;
        return this;
    }
    ...
}
```

Setters are one thing, but method chains in fluent APIs often have a *progression* of ideas. For instance, we could implement part of our Mario DSL such that we can write this:

Listing 8: Fluent event handling.

```
when().key("A").then(mario).jump();
when().collision(mario, enemy).then(mario).die();
```

It is reasonably clear what this is supposed to mean within the context of the game (as long as you know what the game is about). However, it can take some imagination to determine how this code actually works. It relies on a couple of additional tricks.

First, in Java, we can make static methods (like `when()`) accessible *without* their class name, using static imports; e.g.,

Listing 9: Static imports.

```
import static edu.curtin.GameAPI.*;
...
when() ...
```

Second, we can arrange for some behind-the-scenes temporary state to keep track of what each method call in the chain is contributing to the overall meaning:

Listing 10: Fluent API implementation using temporary state.

```
public class GameAPI
{
    private Event currentEvent = null;   // Temporary state
    private GameObject affected = null;

    public static GameAPI when()
    {
        return new GameAPI();
    }

    public GameAPI key(String keyStr)
    {
        this.currentEvent = new KeyStroke(keyStr);
        return this;
    }

    public GameAPI collision(GameObject obj1, GameObject obj2)
    {
        this.currentEvent = new Collision(obj1, obj2);
        return this;
    }

    public GameAPI then(GameObject affected)
    {
        this.affected = affected;
        return this;
    }

    public void jump()
    {
        ... // Now actually set up the event handler.
    }

    public void die()
    {
        ...
    }
}

public abstract class Event() {...}
public class KeyStroke extends Event() {...}
public class Collision extends Event() {...}
```

Once we get to the last parts of the method chain – `jump()` or `die()` – the object has accumulated

enough information to actually properly set up the event handler to perform the requested action. (We'll skip over the mechanics of event handling itself, as you should already have some idea how to do that.)

1.4 Falling Back to a General-Purpose Language

We often also allow general-purpose code inside a DSL. This is so we can retain maximum flexibility, even though we're providing more constrained language structures. There may be a legitimate need to access arithmetic expressions, or conditional logic, or the standard API.

For instance, in our Mario DSL, we might want the player to be able to apply certain in-game items to the Mario character, by opening a dialog box and choosing one of the items. But perhaps, as developers of the DSL, we decide that it's not worth making a domain-specific language feature to describe this, and we simply want to fall-back to Java code instead:

Listing 11: Mario DSL – falling back to Java.

```
gameobject Mario
{
    when Keystroke.F2 then {
        if(!inventory.isEmpty())
        {
            Item item = inventoryChooser();
            if(item != null)
            {
                apply(item);
            }
        }
    }
    ...
}
```

In Listing 3, following "then", we had actions like `Actions.JUMP` and `Big`. In Listing 11, we've provided a block of Java code in place of an action.

2 "Meta" Formats

One of the easiest ways to implement a standalone DSL is to base it on an existing "meta" format, like XML or YAML. These languages are both designed to represent hierarchical data. They don't give us much control over the syntax. However, they do make things easy, because the parsers already exist, and are well-tested.

XML (Extensible Markup Language) is made up of "elements" represented using "tags" inside angle brackets. Each element can have a series of "attributes" (key-value pairs), as well as any number of other elements inside it.

You've seen examples of this before, of coures. It is used by Ant and Maven, for instance. (It looks very much like HTML, although XML and HTML are technically not related.)

When designing an XML-based language, you simply need to decide what elements should exist, what attributes they can have, and what other elements should be allowed inside them (if any). Here's what our DSL might look like as an XML file:

Listing 12: Our Mario DSL as an XML file.

```
<game>
    <gameobject name="Mario">
        <when>
            <event key="SPACE" />
            <action what="JUMP" />
        </when>
        <state name="Small">
            ...
        </state>
        <state name="Big">
            ...
        </state>
    </gameobject>

    <gameobject name="State">
        ...
    </gameobject>
</game>
```

YAML (YAML Ain't a Markup Language) has a minimalistic syntax designed for human readability. Instead of elements and attributes, it is made up of lists, maps (associative arrays) and strings. It uses whitespace indentation (like Python) to indicate what is inside what.

Here's a revised YAML version of our DSL:

Listing 13: Our Mario DSL as a YAML file.

```
---
gameobjects:
    - name: Mario
        events:
            - when: SPACE
              what: JUMP

        states:
            - name: Small
              events: ...

            - name: Big
              events: ...

    - name: State
        events: ...
```

It may be tempting to add a third option to this list: JSON (JavaScript Object Notation). This is conceptually a lot like YAML, except it isn't really designed to be a *file* format[d]. JSON's fatal flaw is that it has no commenting syntax. XML supports "`<!-- ... -->`" comments, and YAML supports single-line "`# ...`" comments.

You cannot design a language, usable by humans, that lacks a commenting capability. Comments are vital, in many situations, for capturing key information about *why* things are the way they are. Even if the language is simple, and the code is simple, the people writing that code may have very

[d] JSON is mostly used for sending messages to and from Web APIs.

complicated or unusual reasons that need explaining. Don't deny them the ability to do this!

3 Parsers

Deciding how to implement a DSL is important. That is, at a technical level, how do we actually make it work? How do we *read* a file written in our new language, and once read, how do we execute it?

The nuclear solution to making a language is to create a parser. You can do this yourself, by creating your own (often recursive) algorithm to read the script file while checking what each symbol and word is, and therefore what needs to be done.

However, it can often be easier to use a **parser generator**, a tool that will actually create the complete parser code for you, provided you can describe the language you want to parse. These include, for instance, yacc (Yet Another Compiler Compiler), JavaCC (Java Compiler Compiler), ANTLR (ANother Tool for Language Recognition) and many others. These tools allow us to specify the exact structure of a language (within some fairly flexible constraints), and will auto-generate code to read (and validate) files written in this new language of ours.

Parser generators allow us to design our DSL more-or-less however we want. They differ in how they work, but there are common elemenets.

3.1 Lexical Analysis (Tokenisation)

Parsers technically work with sequences of **token**s, rather than the raw sequence of characters that make up a text file. A token is the smallest meaningful unit of information in a given programming language, and is made up of one or more characters. For instance, the Java language includes the following types of tokens [2, Ch. 3]:

- Identifiers – the names of variables, methods, classes, etc.
- Keywords (or reserved words) – a fixed set of words that represent key concepts in the language: `class`, `void`, `if`, `return`, etc.
- Literals – numeric, character and string values written directly into the code; e.g. `42.5`, `'X'` or `"Hello"`.

  (This is a low-level meaning of the word literal. In other contexts, it can also represent more complex types too.)
- Operators – a range of symbols (and combinations of symbols) that appear inside expressions: `=`, `==`, `+`, `&&`, etc.
- Separators – a range of other symbols that are not operations per se: `(`, `[`, `{`, `.`, `,`, `;`, etc.

This sort of breakdown is very common, but that's not to say you must design your language this way.

A parser actually relies on another tool (or another part of the same tool), called a **lexer**, to pull out tokens from the raw file contents. That is, the lexer converts characters into tokens, and a parser works with the tokens themselves. The lexer also handles things that the parser doesn't normally care about, but which are part of the source code anyway: whitespace (spaces, tabs and new lines) and comments ("`/*...*/`" and "`//...`").

Tokens, whitespace and comments are typically defined and identified using **regular expressions** (or "regex"es). There are slightly different versions of regex syntax, and we'll follow the syntax used by JavaCC[e].

[e] This is *not* the same as the Perl/grep/awk/sed syntax.

Here are some example regexes for representing certain (simplified) Java syntax:

Identifers: `["_", "a"-"z", "A"-"Z"] (["_", "a"-"z", "A"-"Z", "0"-"9"])*`.

This is saying that an identifier begins with either an underscore or a lowercase or uppercase English letter, and is followed by zero or more other underscores, letters and/or digits. To break down the parts of this:

- `["_", "a"-"z", "A"-"Z"]` matches a single character, which can be an underscore, or a lowercase or uppercase English letter.
- `["_", "a"-"z", "A"-"Z", "0"-"9"]` matches a single character – underscore, letter or digit.
- `(...)*` means zero-or-more of whatever is in parentheses.

(This is slightly more complex in reality, because non-English characters are also allowed.)

Integer literals: `("-")? (["0"-"9"])+`.

- `("-")?` means an optional "`-`" character.
- `(["0"-"9"])+` means one-or-more digits.

(This is also a bit different in reality, because the "`-`" may well be considered a token in its own right, and also because we can write binary and hexadecimal numbers starting with "`0b`" and "`0x`".)

Single-line comments: `"//" (~["\n"])* "\n"`.

- `"//"` simply means that the comment begins with two slashes.
- `"\n"` means a newline character.
- `~["\n"]` means any character *except* a newline character, and so "`(~["\n"])*`" means any number of non-newline characters.

(Once again, the reality is more complex, because line breaks are not always represented by `\n` (technically the "line feed" character), but can also be `\r` ("carriage return") or `\r\n` (carriage return then line feed).)

The lexer will work its way through a source file, and attempt to match each of its regular expressions against the next part of the file. Thus, it generates a series of tokens, which it feeds to the actual parser.

3.2 Parsing Tokens

A parser works with tokens (produced by the lexer) to piece together the overall structure of the source file. In the case of Java, for instance, the parser works out how the tokens are combined into expressions, statements, methods, whole classes, etc. The parser won't understand what these things actually mean, but it will give you a hierarchical representation of them.

The parser will work from beginning to end (with some occasional "looking ahead"). However, unlike a lexer, a parser expects to see different kinds of things at different points in the code. If a parser of Java code has just seen the token "`class`", for instance, it will expect the next token to be an identifier. If it has just seen the token "`return`", it will expect an expression (although there are many different kinds of tokens that might be involved in that).

The rules for what to expect make up the **grammar** of the language. These are typically written in "**Backus-Naur form**" (**BNF**), or a variation thereof. At first glance these can look somewhat like regular expressions, but there are key differences:

- BNF rules build on top of one another. They refer to tokens as well as other rules. Ultimately there is one top-level rule that represents (at a very high level) the entire language.

- BNF rules can be recursive. A rule (representing one component of the language) can refer to itself in its own definition, and can also refer to rules that refer back to it.

  That is, you can make part of a language contain other instances of itself. In Java, for instance:

  - You can have an expression in parentheses, say "`(x + y)`", but each part of an expression can itself be an expression within parentheses, such as `(((a + b) * (c + d)) + ((e - f) / (g - h)))`.
  - You can nest `if`, `while`, etc. statements inside one another.
  - You can have a class nested inside a class, which itself is nested inside another class, and so on.

3.3 JavaCC Overview

JavaCC (the Java Compiler Compiler) allows you to generate both a parser and a lexer to support it. Broadly speaking, you use it like this:

1. Create a "grammar file" (e.g., `MyParser.jj`) that conforms to JavaCC's language. In doing so, you'll indirectly be creating a Java class.

   (You might consider JavaCC's grammar language to be a domain-specific language in its own right – one that's going to help us define other DSLs.)

2. Write code (normal Java code) in your application to invoke your parser, providing the text (read from file) that you want to parse.

3. Run JavaCC in order to generate the source code for the parser/lexer:

   ```
   javacc MyParser.jj
   ```

   This will create several auto-generated .java files.

4. Compile your application along with the auto-generated parser files. This will happen as one step.

5. Run, test, package, etc. (as per the normal build process).

JavaCC's `.jj` grammar files have (broadly speaking) four sections:

- An optional `options{...}` list at the very top. This allows us to tweak aspects of JavaCC's behaviour. In simple cases we can leave it out completely, and we won't really worry about it for the purposes of this unit.
- A class declaration, within `PARSER_BEGIN(...)` and `PARSER_END(...)`. JavaCC refers to this as a "**compilation unit**"[f]. Here you write a Java class, into which JavaCC will later insert auto-generated constructors and methods for performing the parsing.

  Here you can write your own methods that interact with the auto-generated ones. This interaction can go both ways. You can write a method (or methods) to start the parser (by providing an `InputStream`). You can also write methods to be called from within the parsing process, in order to store the information you're retrieving from the parsed file.
- A series of regular expression rules (or **productions** in JavaCC's terminology). These define the kinds of tokens (and whitespace/comments) that the language will deal with, and assign names to token types.

[f] Because, technically, so does Java itself [2, Section 7.3], though this term is rarely used.

- A series of BNF productions that define the large-scale structure of the language. They specify what arrangement of tokens to expect.

  These use a somewhat method-like syntax, and JavaCC will translate them to actual individual methods when it runs. They can call one another, supplying parameters and returning results. They can store local variables. They can also "break out" into regular Java code at any arbitrary point in the parsing process.

3.4 JavaCC Compilation Unit

In a simple case, you might invoke the parser to take input from the console, or from a file:

Listing 14

```
PARSER_BEGIN(MyParser)
import java.io.*;
public class MyParser
{
    public static void parse(String filename) throws IOException,
                                                     ParseException
    {
        MyParser p = new MyParser(new FileInputStream(filename));
        p.dsl();
    }
}
PARSER_END(MyParser)

... // Remainder of the .jj file
```

This class declaration obviously seems incomplete, because it is. However, JavaCC will make it complete, by inserting (among other things):

- A constructor that takes a `InputStream` parameter, and another that takes a `Reader` parameter. (These are closely-related Java concepts, but briefly, an `InputStream` provides a way of receiving bytes from some source, whereas a `Reader` provides a way of receiving actual characters.)

  So, we can make a `MyParser` object based on a `FileInputStream` or `FileReader` (to parse a file), `System.in` (to parse console input), or `StringReader` (to parse an in-memory string).

- A method for each BNF production. We're calling the `dsl()` method here, so there needs to be a corresponding BNF production defined later on.

3.5 JavaCC Regular Expression Productions (the Lexer)

Before we get to the large-scale actual language structure, we must define the small scale parts of it. In JavaCC, you specify tokens and whitespace like this:

Listing 15

```
SKIP : { " " | "\n" | "\r" | "\t" }

TOKEN : {
    < GAMEOBJECT: "gameobject" > |
    < WHEN: "when" > |
    < STATE: "state" > |
    < THEN: "then" > |
    ...
    < IDENTIFIER: ["_", "a"-"z", "A"-"Z"] ["_", "a"-"z", "A"-"Z", "0"-"9"]* > |
    < INT: ("-")? (["0"-"9"])+ >
}
```

The SKIP section defines regular expressions (in this case, just certain single characters) that should be skipped. We normally do this for whitespace, which we want the parser to ignore.

The TOKEN section defines the tokens themselves. Notice that we have a series of declarations separated by pipe ("|") characters, each of the form "< NAME: regex >". The names are referred to later in our BNF productions.

At each point in the file, the lexer will try to match each regex in the order given. So, the order *matters*. If we want to have identifiers as well as reserved words like "gameobject", we must define the latter first, or else any occurrance of "gameobject" in the text would be wrongly categorised as an identifier.

3.6 JavaCC BNF Productions (the Parser)

BNF productions in JavaCC are structured like this:

Listing 16: Structure of JavaCC's BNF productions.

```
<return type> <name>(<parameters>) :
{
    <Java code>
}
{
    <BNF expression>
}
```

The Java code (in the first pair of braces) can be any code, or simply empty. Typically it contains local variable declarations.

The BNF expression specifies what to "expect" in the input text, and what to do about it:

- <GAMEOBJECT> will cause the parser to expect that particular token. Writing several things in a row means you expect all of them, one after another (not counting whitespace or anything else in the SKIP declaration).
- (...)*, (...)+ and (...)? (or [...]) mean, respectively, zero-or-more, one-or-more, and zero-or-one (optional). Think of the first two as loops, and the last as an if statement.
- (...|...|...) means *any one* of the options separated by "|" characters. This is a bit like a switch statement.
- You can make a call to another BNF production. This looks just like a method call, complete with parameters and/or a return value assignment if needed. The called production will handle as many tokens from the input text as it's designed to, and then return, and the current

production will resume.

This allows you to break up the parsing logic into multiple parts, and perform recursion, just as you would for ordinary code.

- You can also insert any arbitrary Java code, anywhere in the BNF expression, by enclosing it in `{...}`. The code will be executed *if and when* the preceding token/expression is matched.

  This isn't designed to parse anything, but rather to perform actions based on what has been parsed. This is where you can build object structures based on the parsed text, or whatever else you need to do in response to it.

- `var = <IDENTIFIER>` tells the parser to store the token in a local variable (which should be type `Token`). The token text is then accessable as `var.image` from within your Java code.

- `<EOF>` is a special "end of file" token. You may want to include this once, at the end of the top-level BNF production.

For instance:

Listing 17: JavaCC BNF productions.

```
void dsl() : // The top-level production representing the overall language
{
    Token name;       // Variable declarations
    int stateNum = 0;
}
{
    (
        <GAMEOBJECT> name = <IDENTIFIER> "{"
        {
            System.out.println("parsing " + name.image); // Java code
        }
        (
            ( <WHEN> ... ) | <STATE> stateNum = state(stateNum)
        )*
        "}"
    )*
    <EOF>
}

int state(int i) : // Another production representing part of the lagnuage
{}
{
    <IDENTIFIER> "{" ... "}"
    { return i + 1; } // The passing around of ints is just for demonstration
}
```

Listing 17 is more an illustration of the JavaCC BNF syntax and concepts than a well-thought-through implementation of our DSL. But hopefully you see where it's going.

To complete the job, we just need to insert Java code to create an object structure based on what the parser finds. For instance, each time we encounter a `<GAMEOBJECT>` token (in a valid location), we'll create a new `GameObject` object and add it to a list. Each time we find a `<WHEN>`, we might add an entry to a list of actions to perform in particular circumstances.

4 Exploiting Language Flexibility

It may sound like a cheat, but one of the easiest ways to implement a DSL *may* be to use an existing general-purpose language. This is not actually a cheat, because certain GPLs do have enough flexibility to create things that look a lot like language constructs.

Groovy, itself a general-purpose language, was designed with domain-specific languages in mind. It is based on Java, but has a few elements that promote the creation of DSLs:

- Like most modern languages, it has **lambda functions** (which it calls **closures**) – anonymous functions defined within expressions and passed around as values, but callable like normal methods/functions.
- The scope of a closure – what the code inside it has access to – can be set and changed programmatically.
- Groovy has a variety of different method call syntax that can make a method call, or sequence of method calls, look like a control structure, especially if it takes a closure as its final parameter.

Gradle is a good example of how Groovy can be used to create DSLs.

Using an existing language has two important advantages:

- There is negligible effort involved in reading and parsing the code – the existing interpreter or compiler can do this for you, of course.
- When writing code in your DSL, programmers can always fall-back to the underlying language. For instance, when writing a Gradle build script, build engineers can fall back to Groovy, the underlying language, when necessary.

However, there are also a couple of potential drawbacks to consider:

- It can be more difficult to enforce the syntax rules of your language. In an "elegant" language, you may strive to have one obvious way of achieving any given effect, but language flexibility can somewhat ruin this ideal.
- By giving the programmer automatic access to an existing general-purpose language, you must deal with the same security issues presented by scripting. That is, if used in a security-critical situation, a DSL based on a GPL can give an attacker everything they need to compromise your software. Using a parser generator or XML/YAML/JSON do not completely avoid security issues, but they do give you much greater control over what you do with any general-purpose code.

5 Further Reading

- The official Groovy documentation on creating DSLs: docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html.

References

[1] Martin Fowler. FluentInterface. https://www.martinfowler.com/bliki/FluentInterface.html, 2005. Accessed: 2019-10-11. [**See section 1.3.**]

[2] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java Language Specification: Java SE 11 Edition.

https://docs.oracle.com/javase/specs/jls/se11/html/index.html, 2018. Accessed: 2019-10-07. [**See sections 3.1 and f.**]