Campaign Description Language Compiler

Section 4.2 already fully described the syntax and semantics of HCCDL. This section explains the implementation of a compiler that can parse this language, evaluate it, and deliver the encountered hypercall and delay requests to a generic interface (campaign listener), which can be implemented by hypervisor-specific modules. To parse HCCDL code, ANTLR (ANother Tool for Language Recognition) is used. All information about ANTLR can be found in the book about it [72], or in the reference manual [73].

ANTLR is a parser generator. That means that it takes a grammar, consisting of lexer rules and parser rules, as input, and generates code that will parse text matching the grammar into an Abstract Syntax Tree (AST). ANTLR supports generating parser code in the languages Javascript, Python, Java, and C#.

Lexer rules are used to transform the input sequence of characters into a sequence of tokens by grouping characters into tokens. A rule consists of a name, which can later be used by the parser rules, and a definition using RegEx (Regular Expression syntax. Lexer rules can be ambigiuous. If multiple rules match, the one producing the longest token is chosen. If multiple rules match with the same length, the first rule (in definition order) is chosen. Listing 5.1 shows the lexer rules used in the HCCDL grammar.

Listing 5.1: HCCDL Lexer Rules

```
STRING: '"' (~'"')* '"';
DEC: [0-9]+;
HEX: '0x' [0-9a-f]+;
BIN: '0b' [0-1]+;
PROC: 'proc';
FOR: 'for';
KEY: 'key';
VAL: 'val';
IDENTIFIER: [_a-zA-Z][_a-zA-Z0-9]*;
WHITESPACE: [ \t\n\r]+ -> skip;
```

The first four rules represent the elementary data types. Strings are a sequence of arbitrary non-quotation mark characters, enclosed by quotation marks. Decimal, hexadecimal, and binary numeric values consist of their prefix, together with a sequence of respective digits. `PROC`, `FOR`, `KEY`, and `VAL` represent keywords in the language. These rules make sure that they are not tokenized as generic identifiers. The last rule prevents spaces, tab, and newline characters from being tokenized, they are ignored. Doing this has the advantage that parser rules can focus on parsing actual tokens because they don't have to include whitespace tokens in between other tokens.

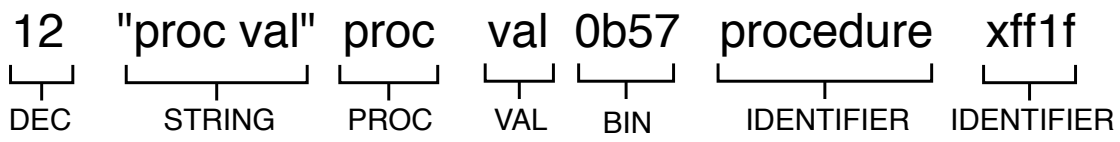

Figure 5.1.: HCCDL Lexer Rules Applied to Sequence of Characters

Figure 5.1 shows the rules tokenizing a sequence of characters. The spaces in between do not get carried over to the sequence of tokens. `proc` and `procedure` demonstrate the resolution of ambiguity. In the case of "procedure", the `PROC` rule matches and would create the token "proc". `IDENTIFIER` matches as well, and would create the token "procedure".

Because the second option produces the longer token, the `IDENTIFIER` rule is chosen. However, the character sequence "proc" is also matched by both, completely. Thus, the earlier defined rule, which is `PROC`, is applied.

With the input campaign file transformed, the parser rules can put the tokens together to more complex syntactic constructs. When the language was defined, a bottom-up approach was used to start from the tangible elementary data types and construct a more and more global view of the program from there. Now that the language is already completely known and only the rules have to be created to match it, it makes sense to begin with defining what a program is, and recursively go into more and more detail until everything is defined.

When parsing input, a parser rule has to be given that is supposed to match to whole input. The rule `program`, shown in Listing 5.2, is this rule for the HCCDL grammar. It states that a program consists of a sequence of procedure definitions and global variable declarations. This rule does not enforce the existence of the `main` procedure. Such requirements should not complicate the grammar; they can be checked after the parsing. Next, following the top-down approach, procedure and global variable rules should be stated. Both are also part of Listing 5.2. There are two alternatives for declaring global variables. Multiple can be declared uninitialized, separated by commas.

Listing 5.2: HCCDL Program Structure Parser Rules

```
program:
    (procedure | globalvar_declaration)+;

globalvar_declaration:
    IDENTIFIER (',' IDENTIFIER)* ';'
  | name=IDENTIFIER '=' val=number ';';

procedure:
    PROC IDENTIFIER parameterlistDeclare '{' statement* '}';

parameterlistDeclare:
    '(' (IDENTIFIER (',' IDENTIFIER)*)? ')';
```

Alternatively, it is possible to initialize a global variable with a number. Both variants have to be concluded with a semicolon. The `number` rule will be explicitly shown, as it is defined to be either a decimal, hexadecimal, or binary token.

More interesting is the definition of a procedure. It requires the keyword "proc" at the beginning, then an identifier, which names the procedure, a list of zero or more identifier (parameter names), separated by commas and enclosed in parentheses. The procedure body consists of zero or more statements, enclosed in braces.

Statements are defined analogously to their worded description in Section 4.2. Listing 5.3 displays the rule.

Listing 5.3: HCCDL Statement Parser Rules

```
statement:
    FOR '(' IDENTIFIER ':' expression ')' statement
  | '{' statement* '}'
  | expression ';';
```

A statement is either a for loop, which can iterate over a list of values (here the `expression` has to evaluate to a list, however, this cannot be checked by the syntax), or a block of statements, or an expression.

Listing 5.4: HCCDL Program Structure Parser Rules

```
expression:
    list=expression '[' index=expression ']'
  | expression '.' op=(KEY | VAL)
  | op=('+' | '-') expression
  | expression op=('*' | '/' | '%' ) expression
  | expression op=('+' | '-') expression
  | expression '->' expression
  | IDENTIFIER '=' expression
  | '[' (expression (',' expression)*)? ']'
  | '(' expression ')'
  | IDENTIFIER parameterlistCall
  | IDENTIFIER
  | number
  | STRING;
```

All elementary data values, all operators, and procedure calls are expressions. Thus, the rule is composed of many alternatives, which can be seen in Listing 5.4. Apart from the elementary data types and variables (the `IDENTIFIER` rule), all expressions are recursively defined. The order of alternatives is essential to resolve ambiguities, such as the example given in Figure 5.2. Both parse trees are valid in terms of the rules, but which one of

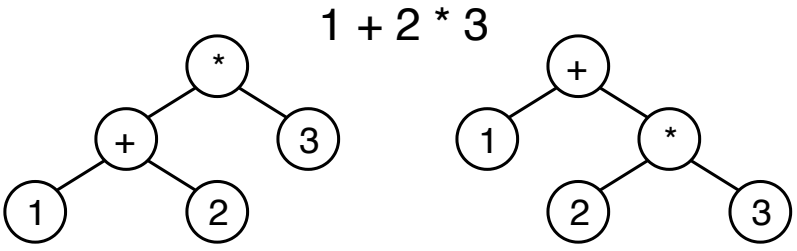

Figure 5.2.: Parser Rule Ambiguity Example

them will the parser build? Because the multiplication alternative is defined before the addition, it is applied first. Thus, the parse tree on the right side is correct.

These are all the grammar rules necessary to parse campaigns written in HCCDL. As mentioned before, the `program` rule has to match the entire campaign. From there, the parser generated by ANTLR builds the parse tree. Lexer rules and other parser rules used in a parser rule become child nodes. Lexer rules are always leaves. To aid in understanding the concept, Figure 5.3 shows the parse tree of the sample HCCDL campaign in Listing 5.5.

Listing 5.5: HCCDL Campaign for Parse Tree Example

```
x = 5;

proc main() {
    x = x + 1;
}
```

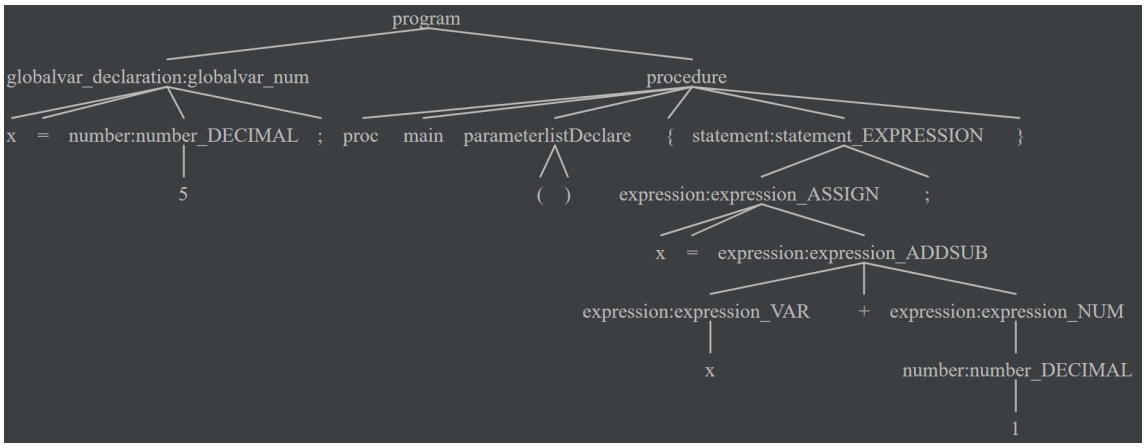

Figure 5.3.: HCCDL Parse Tree Example

For implementing the compiler, which has to process such parse trees, Java has been chosen. Figure 5.4 shows the somewhat unusual folder structure of the project.

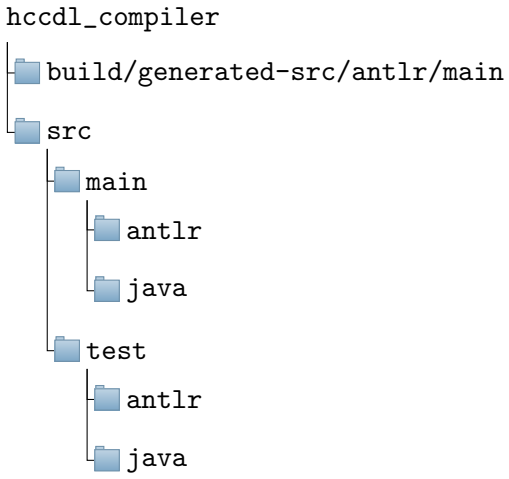

Figure 5.4.: HCCDL Compiler Java Project Structure

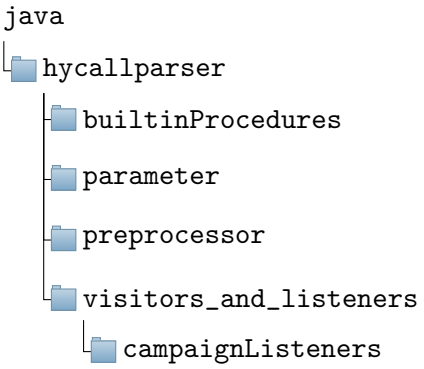

Figure 5.5.: HCCDL Compiler Java Package Overview

The `main` and `test` folders do contain not only a `java` folder but also an `antlr` folder. In `scr/main/antlr` reside the grammar file. When ANTLR is invoked, it generates Java source code for a parser according to the grammar. The Java source files are placed into `build/generated-src/antlr/main`. Subsequently, there are two distinct source folders. A Gradle build file has been written, which is configured to compile and execute the project correctly.

The parser code generated by ANTLR builds a parse tree out of Java classes, which correspond to the parser rules of the grammar. ANTLR provides two software design patterns to traverse through the parse tree: Listener and Visitor. When implementing a listener interface for a grammar, for every rule, an `enter` and an `exit` method exists. The parse tree is traversed in a depth-first manner. Every time a child node is encountered, the listener's corresponding `enter` method is called, and the node object is passed for context. When all children of a node have been traversed, the `exit` method of the listener is called.

Visitors are interfaces with a `visit` method for every parser rule. The `visit` methods have a generic return value, whose type has to be set when implementing a concrete visitor. In the beginning, the `visit` method for the root node is executed. The `visit` methods have to decide which of their child nodes should be visited, and how return values can

be aggregated. However, before the visitors and listeners implemented are explained, the other packages (see Java package structure, Figure 5.5) should be explained first.

Package `hycallparser.parameter`

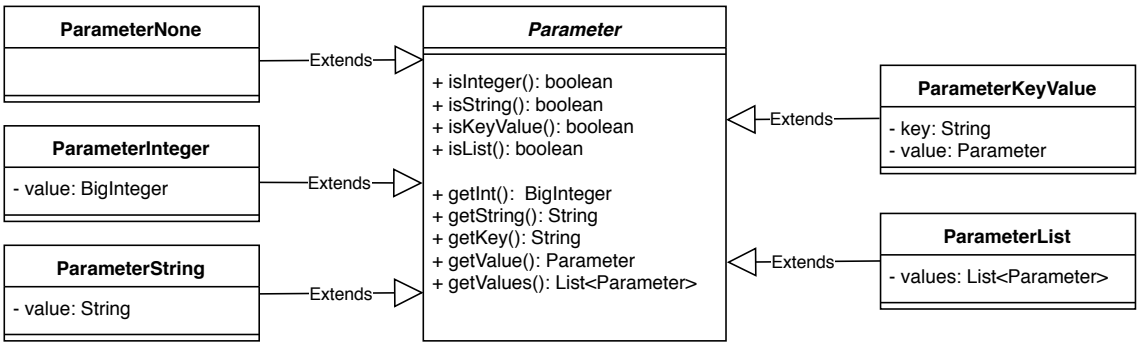

Figure 5.6.: UML Class Diagram of the `parameter` Package

The `hycallparser.parameter` package provides classes to store values of the different data types available HCCDL. Polymorphism has been used such that key-value pairs and lists can hold values of any data type. By defining all the methods in the `Parameter` class, all getters can be called on any Parameter. `UnsupportedOperationException`s are thrown on wrong calls. Implementing it this way has the advantage of not needing to cast classes to the specific implementation.

Package `hycallparser.preprocessor`

There is only one class in the `preprocessor` package: the `CampaignFileLoader`. Given a filename of a hypercall campaign file, it loads the content of the file into memory and searches for occurrences of `#include` statements. For every `#include` detected, it tries to load the file specified after the `#include` and replaces the statement with the file content. The newly added content might also contain `#include`s, so the `CampaignFileLoader` recursively repeats this process until no new occurrences are found.

Package `hycallparser.builtinProcedures`

All implemented built-in procedures reside in this folder and are instances of the `Procedure` interface, which is presented in Table 5.6.

Listing 5.6: Interface `hycallparser.builtinProcedures.Procedure`

```
public interface Procedure {
    String getName();
    int getParameterCount();

    Parameter perform(List<Parameter> args);
}
```

More built-in procedures can be implemented easily because, at compiler startup, the class `ProcedureManager` uses the Java Reflections API to create an instance of every Procedure implementation. During the evaluation of a campaign, when a procedure call is issued, but there is no user-defined procedure with the given name, the `ProcedureManager` queries the names and parameter counts of the `Procedure` implementations, and if it finds a match, the matcher's `perform` method is executed. So, adding a new built-in procedure requires only a new implementation of the `Procedure` interface, stored inside the `builtinProcedures` package.

Package `hycallparser.visitors_and_listeners`

All implemented visitors and listeners, which implement interfaces from ANTLR generated source code, are contained in this package, together with several helper classes. There is no single instance that processes the complete parse tree alone. Various listeners and visitors have been implemented for specifics tasks.

The `NumberVisitor` only implements the methods for visiting the `number` parser rule. For this visitor, the `visit` methods return `BigInteger`s. The idea is that if any object has to process a parse tree node corresponding to the `number` rule and wants to retrieve the numeric value, it can create a `NumberVisitor` instance, tell it to visit the node and get the numeric value as a return value. The `NumberVisitor` encapsulates the functionality of finding the integer value of decimal, hexadecimal, and binary strings.

The same principle applies to the `ExpressionEvaluator`. If any object has a parse tree node corresponding to the `expression` rule, it can tell an `ExpressionEvaluator` object to visit the node to retrieve the evaluation result of the expression. The return value of the `visit` methods is `Parameter`. Depending on the expression to be evaluated, an `ExpressionEvaluator` instance can either directly return the evaluation result (if the expression is just an elementary value or a variable lookup), or, if there are further expressions nested, call itself to evaluate the child expressions' values before performing the actual operation.

There are two helper classes, which keep track of state during the execution of campaigns. The `ProgramState` manages global variables and user-defined procedures. The `ProcedureState` keeps track of local variables for procedures. While there is only one global `ProgramState`, every called procedure has its own `ProcedureState`.

The first action happening to a campaign's parse tree is that a `ProgramStructureListener` traverses the tree, identifies all global variables and user-defined procedures, and creates a `ProgramState` with this information. The `ProgramState` object is passed to a `ProgramExecutor`, which checks whether an `init` procedure exists in the `ProgramState`. If it does, the `ProgramExecutor` creates a new instance of a `ProcedureExecutor` and lets it execute `init`. After `init` finished, the `ProgramExecutor` checks the `ProgramState` to see whether the `main` function exists, and either lets a `ProcedureExecutor` execute it, or throws an error stating that there is no `main` procedure.

Package `hycallparser.visitors_and_listeners.campaignListeners`

This package is responsible for managing the available hypervisor-specific modules, the campaign listeners. Campaign listeners do have to implement the `CampaignListener` interface in order to be informed about hypercalls and delays. Listing 5.7 shows the declaration of the said interface.

Listing 5.7: Interface `visitors_and_listeners.campaignListeners.CampaignListener`

```
public interface CampaignListener {
    void setParameters(String[] args);
    void encodeHypercall(List<ParameterKeyValue> parameters);
    void encodeDelay(long delayMicroseconds);
    void finish();
}
```

Additionally to the mentioned hypercall- and delay-related methods, there is also the `setParameters` method. It gets called before the evaluation of the campaign and transmits the command line arguments that the compiler application received to the listener. This is required, e.g., for report generators, because they have to know the file path to the

binary log file. Also, the `finish` method is called at the end of the campaign. It can be used to close open files or perform other cleanup actions. The campaign listener that should be used for compilation can be set by passing the name of the requested listener as a command-line parameter. Similarly to the built-in functions, Java Reflections are used to instantiate the correct `CampaignListener` implementation dynamically. However, the `CampaignListener`s can be located anywhere in the source files. In the `campaignListeners` package, only an implementation of the `CampaignListenerProvider` interface has to exist, which can instantiate the `CampaignListener`s. It also defines the names for them, which have to passed via command-line argument.

This concludes the implementation of the generic parts of the framework, with the interface to the hypervisor-specific details provided via the `CampaignListener`, which means the second goal of this work has been achieved:

G2: *Implement a generic framework for designing, executing and evaluating hypercall test campaigns with interfaces for tasks that are hypervisor-specific and can't be generalized.*