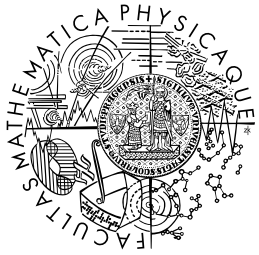

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Július Flimmel

Pascal with Truffle

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Vojtěch Horký
Study programme:	Informatics
Study branch:	Programming and Software Systems

Prague 2017

I declare that I carried out this bachelor's thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Pascal with Truffle

Author: Július Flimmel

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Vojtěch Horký, Department of Distributed and Dependable Systems

Abstract: Trupple is an unconventional Pascal interpreter built on top of Oracle's Truffle framework. By using this framework, it is virtually platform independent because it runs in Java Virtual Machine and can also easily communicate with other Truffle-based languages and Java itself. The interpreter builds an abstract syntax tree from any Pascal source code and consequently executes the tree from its root node. It supports Pascal according to ISO 7185 standard and implements some commonly used extensions introduced by Borland's Turbo Pascal compiler. In this work, we describe the architecture of the interpreter, important design decisions, used technologies and we also provide a brief performance evaluation of Trupple.

Keywords: Pascal Truffle JIT compiler

I would like to express my sincere gratitude to my supervisor, Mgr. Vojtěch Horký, for his advices and guidance throughout the work on this thesis without which I would not be able to finish it.

I also have to thank all my closest that supported me with my studies for their help, motivation and understanding throughout the past years.

Contents

1. Introduction

The Pascal programming language was firstly introduced in early 1970's by Niklaus Wirth. At that time, it gained a huge popularity because of its simplicity and strong emphasis on structural programming. However, it was later superseded by newer programming languages, which are more expressive and more tailored for current projects, primarily due to better possibilities in organizing large projects, using parallelism, OOP and other advantages.

Although Pascal was standardized only twice [1][2], many dialects have evolved from it, extending its syntax and semantics. Among the most popular ones were Turbo Pascal which introduced code modularization and Delphi that added OOP and brought modern GUI libraries. Even though Pascal is today considered obsolete by many, a lot of popular software was developed in Pascal (or one of its derivatives) that is still in use today. These software include *PSPad*, *HeidiSQL*, *Total Commander* and even Microsoft's *Skype*.

Still, Pascal keeps its prominent position in some use cases. Because of its verbosity in control flow constructs, it practically became a standard for writing pseudo – codes, because it helps to highlight algorithmical solutions. This property also makes Pascal a good choice for being the first programming language to learn as the to – be – developers will not get lost in the clutter of non – alpha – numeric letters.

Pascal is definitely not the top choice for software developers today, but even after almost half a century it is still popular and in use. Thus we decided to create a modern interpreter of this programming language on top of Java VM. By using the Truffle framework we would build a JIT-compiled interpreter that could, in theory, outperform traditional compilers, be portable to any device running Java and would easily integrate with other programming languages, ranging from Java, Ruby to JavaScript which are already implemented in Truffle.

1.1 Goals of the work

The primary goal of this project is to implement a fully functional Pascal interpreter in Oracle's Truffle framework. The interpreter shall meet these requirements:

- Support Pascal as it is standardized in ISO 7185 document to the most reasonable extent on current hardware and software,
- Be able to integrate Pascal into the Truffle framework making it possible to interact with other languages written in it,
- Support some of the most commonly used extensions to the language introduced by Borland's Turbo Pascal dialect. These extensions include:
  - A possibility to write custom units and support of some built-in units,
  - Passing function arguments by reference,
  - Syntax improvements,

  - Break statement, and others.

- Run reasonably fast – since we will not be compiling Pascal source files directly to native language but we will be using an intermediate language (Java bytecode) we cannot expect to be as fast as compiled languages including original Pascal programs. On the other hand it is not acceptable for us to perform like three orders of magnitude slower compared to them.

1.2 Structure of the thesis

The whole thesis is divided into multiple chapters. It begins with analysis of our possibilities of implementing the interpreter in Chapter 2. We discuss the problems that we will encounter, options how can we solve them and which one will fit best for our purposes. It continues with programming documentation in Chapter 3. In this chapter we will describe our implementation and how we have implemented the solutions chosen in the previous chapter. It is followed by user documentation in Chapter 4, where we describe how to run our interpreter and which inputs does it require. Afterwards, we look at multiple benchmarks and comparisons between our interpreter and other compilers or interpreters that concern us in Chapter 5. The work is finished with Conclusion where we sum up how much we were able to accomplish our goals and how can it be more improved.

2. Analysis

In this chapter we will describe specifics of Pascal in comparison with other programming languages. We will also analyze Truffle and our possibilities of implementing our interpreter using this framework.

We will continue with describing problems that we will have to solve, a list of solutions to them and discussions about which of them fits our needs the best. Not each single decision that we have made will be mentioned in this chapter because the thesis would end up unnecessarily long, but only the ones worth mentioning. Some of the less important decisions will be mentioned in the Programming documentation chapter.

2.1 Truffle

Truffle is a Java framework developed by Oracle company. It is made for building interpreters of dynamic languages. The framework provides means to build abstract syntax trees (later only ASTs) of codes of any language, fast implementation of its nodes and their consecutive execution. Being primarily designed for dynamic languages introduces some limitations for us as we will be implementing interpreter for a statically typed language. These limitations will be discussed later in this chapter.

A big part of Truffle is Polyglot engine which is responsible for evaluation of Truffle ASTs and the communication between arbitrary Truffle languages.

Also a new JIT compiler is being developed which is optimized for compilation of Truffle based interpreters. It is called Graal and is used exclusively for Truffle based projects.

Let us now look at some advantages of using Truffle in comparison to classical compilers:

- We do not have to implement the whole compiler, we only need to build the AST of an input source code which is much like implementing only the front end part of a compiler,

- The most commonly used feature of Truffle is node specializations. If we have a node that may be executed with multiple combinations of argument types (like node for + operation which can be executed with two integers or two strings) we only need to implement functions with all of these combinations in one class representing the node and mark them as specializations. Truffle will than take care of choosing the correct function to use at runtime. Listing 2.1 shows some specializations of node for + operation. We want to avoid generics here because they are just Java Objects which are stored on the heap and working with them instead of primitive types would ruin our performance. We are loosing a bit of an advantage of these specializations by implementing a statically typed language because Truffle is capable of changing nodes' specializations at runtime, which may happen in dynamically typed language if, for example, some variable in an expression changes its type, but can not happen in Pascal. On the other hand, we still do not

Listing 2.1: Example of Truffle's specializations

```
class AddNode extends Node {
    @Specialization
    int addInts(int left, int right) {
        return left + right;
    }

    @Specialization
    double addDoubles(double left, double right) {
        return left + right;
    }

    @Specialization
    char addChars(char left, char right) {
        return left + right
    }
    ...
}
```

have to implement a new node for each specialization or take care about casting node arguments to proper types,

- Deciding which specialization to use every time we execute a node would be really inefficient. Before a node is executed for the first time, it is in an unspecialized state. After the execution, a specialization is chosen which corresponds with the provided arguments and this specialization is then used on each subsequent call of this node. The overhead of checking whether the arguments still have the correct type is small,

- An important process in compilers is creating and managing so called frames. These are parts of a stack that belong to one function call and store its data. A frame typically consists of local variables, return address, parameters and address of the caller's frame. Truffle already contains and manages this structure itself so we will not have to implement it ourselves. The only problem with Truffle frames is that they do not contain the address of the caller's frame which we will have to add there ourselves. The only thing we need to do to call a function and create a new frame for it, is to:

  - Create a call target of that function in parsing phase and supply it a root node of its AST,
  - Execute *call()* method on the call target at interpretation time and provide it arguments and a frame descriptor. Frame descriptors are structures that define what variables a frame shall contain along with their types. Then a new frame for the called function is created automatically by Truffle.

- Truffle allows us to directly affect machine code generated by the Graal compiler by annotations like *@CompilationFinal*, allows us to force Graal to recompile Java bytecode into the machine code with *compiler directives*, make *Assumptions* and branch *Profiling* "for free".

Listing 2.2: Example of nested subroutines in Pascal

```
procedure helloWorld;
    procedure printIt;
    begin
        write('hello world');
    end;
begin
    printIt;
end;

begin
    helloWorld;
    printIt; { Compilation error }
end.
```

2.2 Pascal specifics

Pascal is today considered old and bit outdated programming language. One reason for this is that it uses some constructs and concepts that are not used in modern languages, and a modern framework like Truffle may not count with implementation of these. We will focus on them in this chapter and discuss whether we are able to implement them in Truffle or not. Firstly, we will compare them with similar Java constructs which will give us more ideas whether we will be able to implements them and how (for obvious reason and that is that we will be writing our nodes in Java). We will also illustrate them on a piece of code. All of them will be later discussed in more detail.

2.2.1 Nested subroutines

Pascal, Python and some others are a small number of popular languages that use this concept. They allow its users to define a function inside another function making this inner function available only from the function inside which it is defined. Since this concept is not available in Java language we will have to implement it ourselves. Simple example of nested subroutines can be seen in Listing 2.2.

2.2.2 Variant records

Records in Pascal are structures that hold some set of named variables. The user may define custom records in his source code. Variant record parts declare *variable* parts of these records. These are groups of variables from which only one at a time is active. Variables from the non active groups are in an undefined state. Turbo Pascal's variants are implemented like C's unions because they also physically share the same piece of memory. These extended records are too low level concept to be natively supported by Java (and is no longer really needed thanks to the class inheritance concept) and we will have to deal with them ourselves. A variant record thanks to which we can access individual bytes of an integer can be seen in Listing 2.3.

Listing 2.3: Example of Pascal's variant record

```
type
    parts = (int, bytes);
    integerDecomposition = record
        case parts of
            int: (value: integer);
            bytes: (decomposition: array[0..3] of byte);
    end;

var id: integerDecomposition;
begin
    id.value := 42;
    writeln(id.decomposition[2]);
end.
```

Listing 2.4: Example of returning a value in a Pascal function

```
function factorial(i: integer): integer;
begin
    if i < 2 then factorial := 1
    else factorial := i * factorial(i - 1);
    { the else branch has to be present here }
end;
```

2.2.3 Missing return

In today's languages we commonly return a value from a function with *return statement*. In Pascal, however, we return a value from function by assigning it to the identifier of the function. Another difference is that this assignment does not stop the function's execution. This is due to Pascal's strict adherence to structured programming concepts, hence also missing break and continue statements (however goto breaks this bonding). Because Java also uses return statements, we will have to simulate this behavior ourselves. Listing 2.4 illustrates this concept.

2.2.4 Units

Standard Pascal does not allow splitting code into multiple files. Each project consists of a single source file. This is problematic for large projects. On behalf of keeping source files short and easy to read, Turbo Pascal added concept of units which we have decided to support in our interpreter.

Units are pieces of Pascal code that can be reused in other sources. They have only a little different syntax compared to ordinary sources. Firstly, they specify interface (which consists of subroutine headings, types and variables that are visible from the outside) and then implementation of the subroutines specified in the interface part.

Since Truffle is designed for evaluating single scripts we will have to implement some system for storing everything that was declared in each included unit and make it available from the main source.

Listing 2.5 shows simple unit containing only function *add* in its interface.

Listing 2.5: Example of unit declaration

```
unit math;
interface
function add(a,b:integer): integer;

implementation
var callsCount: integer;

function add(a,b:integer): integer;
begin
    callsCount := callsCount + 1;
    add := a + b;
end;
end. { end of unit }
```

Listing 2.6: Example of using a function from unit declared in Listing 2.5

```
program unitTest;
uses math;
begin
    write(add(2,6));
end.
```

Listing 2.6 then shows how we can use that unit in a source file.

2.2.5 Visibility modifiers

Pascal does not use common public/private visibility modifiers like many other languages. Each identifier is simply accessible only from the scope where it was defined, or inner scopes of arbitrary depth, making all identifiers behave like public. This does not apply only to identifiers defined in the implementation part of a unit and not defined in the interface section at the same time. These identifiers then behave like private and may be accessed only from within the unit. Listing 2.5 shows a variable *callsCount* which is not accessible from the outside of the unit *math*.

2.2.6 Pointers

Since pointers in Pascal do not support arithmetic operations they are pretty similar to Java's references. The most significant difference between them is, that in Pascal we are able to manually dispose an object where in Java we are not. There are more ways of implementing Pascal pointers and will be discussed later in this chapter.

2.2.7 Files

Standard Pascal treats files a bit differently then what we are used to in modern languages. Firstly, there is no way to specify a path to a file. This is due to historical reasons. Pascal files are treated more generally. These "files" did not have to be only files as we know them today (bunch of data stored on

Listing 2.7: Code that copies content of one textfile into another in Pascal

```
program copytext (input, output);
{ This program copies the characters and line structure of the
textfile input to the textfile output }
var ch : char;
begin
    while not eof do
    begin
      while not eoln do
        begin read(ch); write(ch)
        end;
      readln; writeln
    end
 end.
```

Listing 2.8: Code that copies content of one file into another using *get* and *put* subroutines

```
program copy (f, g);
var f, g : file of real;
begin reset(f) ; rewrite(g);
    while not eof(f) do
    begin
        g^ := f^; get(f); put(g)
    end
end.
```

an external memory having some header in a filesystem) but they could also be, for example, mounted tapes (which were still in use at the time of creation of Pascal) or some other devices. Thus Pascal leaves bounding of files to variables to the OS.

Listing 2.7 shows a program that copies content of input file to the output file. The code is taken from the Pascal standard. We can notice that the *input* and *output* program arguments are not even used. The input and output are bound directly to the files but it is not done inside the Pascal code.

Another example can be seen in Listing 2.8 (also taken from the Pascal standard) which is even more minimalistic. Once again it copies content of input file to the output file. This time it directly uses the file variables which are handed over to Pascal as program arguments. The bounding of these variables to actual files is again done outside of the Pascal code.

We can also notice that the files in Listing 2.7 are treated like text files while in Listing 2.8 like binary files. Pascal is therefore capable of working with both of these file types. If we are working with binary files we can also specify semantics of the binary data (whether they are integers, reals, chars, etc.).

Two basic operations on files next to writing and reading from them are *eof(f)* and *eoln(f)*. Both of them return boolean value, the first mentioned returns true if we are at the end of a file while the second returns true if we are at the end of a line. The *eoln* function can be used only on text files.

Turbo Pascal has extended support for work with text files compared to the standard. In the standard Pascal, the text file type is simply a *file of char*. This

Listing 2.9: Example of work with files in Turbo Pascal

```
program textfiles;
var f: text; i: integer; r: real; s: string;
begin
    assign(f, 'text.out');
    rewrite(f);
    writeln(f, 42); writeln(f, 3.14); writeln(f, 'All the clowns');
    close(f);
    reset(f);
    readln(f, i); readln(f, r); readln(f, s);
    close(f);
end.
```

Listing 2.10: Example of a valid goto on line 4 and an invalid goto on line 9

```
1  begin
2      while true do begin
3  beginloop:
4          goto endloop;
5      end;
6  endloop:
7
8      while true do begin
9          goto beginloop;
10     end;
11 end.
```

means that we can only read or write one single-byte character at a time to a file with one read/write call. Turbo Pascal's *text* type which represents a text file offers more functionality. We are able to write or read multiple standard types (like strings, integers, reals, etc.) from a single file. This is depicted in Listing 2.9. We are also able to assign a path to a file.

2.2.8 Goto statement

There are multiple issues with goto statements. Not only that they do not have their equivalent in Java, they also cannot be interpreted in a syntax tree and may be implemented only on the machine code level. However, Pascal standard does some restrictions on them which make them easier to implement. We will have to analyze possibilities of rewriting goto statements into statements supported by Java and their consequent placement in the resulting AST. Listing 2.10 shows a valid goto statement (the first one) and an invalid goto statement (the second one).

2.3 Parsing input

From now on, we will move to the analysis of problems mentioned in the previous section and some others. First of all, we need to be able to parse a source code that we get on input and build its syntax tree. We could implement a custom

parser but there already exist numerous tools that can do it for us if we provide them a grammar, so we will choose one of them.

We will use the Coco/R tool because it was used for implementation of interpreter for *SimpleLanguage*, which is a demonstrational language made by Oracle to promote the Truffle project (and also serves as an example of usage of Truffle) so we know that it works well with the framework.

2.3.1 Coco/R

Coco/R is an LL(1) parser which is less powerful than other commonly used grammar recognition tools. On the other hand, it is sufficient enough to parse a Pascal grammar. If get to a situation where lookahead of depth 1 is not sufficient enough, it provides a way to insert a conflict resolver in which we may use lookahead of arbitrary depth. As any other tool, it allows us to insert our own code (Java code in our case) inside grammar rules which we will be using for creation of nodes and updating current parsing state.

We will be using common conflict resolving techniques like extracting conflicting token from rules that are in conflict. In cases where grammar is unable to distinguish between multiple options we will use Java code. For example the following line:

```
a := count;
```

The identifier *count* may represent either a *variable* or a *function* what grammar alone can not distinguish.

2.4 Multiple dialects support

One of the goals of our thesis is that our interpreter will be able not only to interpret standard Pascal, but also support some chosen Turbo Pascal extensions. Because these extensions affect also grammar (for example units, break statement, multiple type statements in one source and many more) we need to be able to choose which grammar to use when parsing an input source. We may solve this problem multiple ways:

- We may solve this at runtime, and if something does not correlate with the chosen dialect, we will throw an exception. This solution is not very user friendly because we want to inform the user about syntax and semantic errors at compile time,
- We may solve it at the time of building AST – we will add numerous checks to our Java code that takes care of building nodes,
- Another option is that we will have two parsers, one using standard Pascal grammar and the other using extended Turbo Pascal grammar. This option will make our grammar files much cleaner but it comes with disadvantage in grammar duplicity. Throughout the implementation process, when we will want to apply some changes to one grammar file we will most likely need to apply them also in the other one. On the other hand, the resulting Java code will also be more cleaner and readable.

We will use the last mentioned approach since it is the most user friendly one – for users of our interpreter and for implementing possible extensions to our implementation.

2.4.1 Units parsing

Units in Turbo Pascal have only a little different syntax than ordinal Pascal sources. It uses mandatory *unit identification* statement at the beginning instead of optional *program identification* statement, then definition of interface followed by implementation section. This last section is almost identical to an ordinary Pascal source except that it does not include the main program block. The interface section is a list of subroutine headers, variables and type definitions. Being almost identical to the ordinary Pascal sources, we do not want to use the same approach as in the previous section and create another grammar file exclusively for the units. Thanks to the fact that unit must always start with special keyword that does not occur in ordinary sources, we may add a grammar rule at the beginning of our grammar file, which will decide whether it will be parsing an ordinary source or a unit source. Advantage of this approach is that the unit grammar rules will be able to use already defined rules for Pascal sources.

2.4.2 Representation of unit files

In order to use units in Turbo Pascal, they need to be compiled. This compiled version may be afterwards loaded in other Pascal sources. There is a problem with implementation of this behavior in our project and that is, that we are not compiling, but interpreting the sources. We would have to be able to interpret this compiled binary files, and even compile units that we get on the input, which is out of scope of our thesis.

Because of this, we will not be working with units in their compiled form, but rather directly with their source codes. This will allow us to interpret them, store their content in some global context and then execute the main source.

2.5 Static type checking

Pascal uses static type checking. This means that type of each identifier has to be known already at compile time. However, we may consider whether it will not be better to to implement this type checking to be done at runtime because it has its advantages. It is considerably easier thanks to Truffle's specializations which take care of it themselves. If a node gets unsupported combination of argument types, Truffle automatically throws an exception. This means that we would not have to implement type checking for most of the nodes (some nodes may not be using specializations). On the other hand, the Pascal standard specifies that this type checking has to be done at compile time, and it is also more user friendly. As Truffle is designed for dynamically typed languages, it will not help us in implementing it this way. However, we decide to follow standard and implement the type checking at compile time.

To implement it this way, we will need to remember identifiers that we have already encountered and their types. For this purpose, we will implement a table

Listing 2.11: Example of a nested subroutine call

```
program nestedFrames;
var c: char;

procedure outer;
var o: integer;

    procedure inner(i: integer);
    var a,b: integer;
    begin
        o := i;
    end;

begin
    o := 26;
    inner(42);
    writeln(o);
end;

begin
    outer;
end.
```

of identifiers, that will hold type information about each identifier. This will make us able to compare types of each expression wherever it is needed.

2.6 Nested Subroutines

Pascal is very simple when it comes to visibility of identifiers as already discussed in Section 2.2.5. They may either be global (defined at the very beginning of the source code, or if using Turbo Pascal then anywhere in global scope) or local (defined at the very beginning of a subroutine). This leads us to simple implementation, just to hold one set of global identifiers, and one set of local identifiers for current scope.

The concept of nested subroutines is a little problematic here. When in a subroutine, we need to be able to access local variables of a parent subroutine or local variables of parent of parent subroutine, etc. This means that we will have to implement a parent-child hierarchy of lexical scopes and be able to access parent scope from its children scopes.

This is not a problem to implement it for parsing phase but for the runtime. We know that Truffle's frames do not contain reference to their parent's frame. We will solve this by supplying our current frame to the arguments of each called function. This way we may access caller's frame from arguments of our current frame up to the global frame.

On Figure 2.1 we can see how individual frames will look like when the *inner* procedure is called from the *outer* procedure call in Listing 2.11.

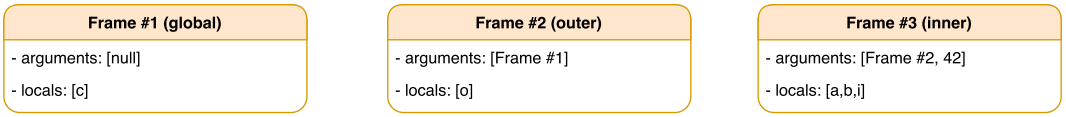

Figure 2.1: Illustration of frame objects from the execution of code in Listing 2.11

2.7 Input / Output

In this section we will discuss which Java constructs are best fit to interpret Pascal's I/O operations. Pascal supports operations with files and standard input and output, they are analyzed in the following subsections.

2.7.1 Standard I/O

Pascal's possibilities of operations with standard input and output are rather simple. It can read and write only primitive types. In addition it can not read boolean type variables from the standard input. This allows us simply to use a *Scanner* to read input and a *PrintStream* to print output.

2.7.2 Files

We also need to be able to perform read and write operations on files. There are numerous streams in Java that can be used to work with them. However, almost all of them have the same problem and that is the way they work. They do not contain any method that can tell us whether we are at the end of the input. This means that we are unable to implement Pascal's *eof(f)* function. All of the Java input streams use *trial and error* technique. Firstly, we have to read from it and then wait for an end of file exception. The problematic part is that if the exception is not thrown, we have moved forward in the file and skipped some data. This leads us to use some stream that is able to return read data back to its buffer.

Java's standard library contains one stream with this functionality and that is *PushbackInputStream*. Using this stream to read from files, implementation of Pascal's *eof(f)* method will be to firstly read data from the file and if no exception is thrown push this data back to the stream and return false. Otherwise we return true and we do not have to push back anything since we have reached the end of file.

We also need to be able to store and read any variable type. For this purpose we need to use *ObjectInputStream* and *ObjectOutputStream*. However, using these streams we are unable to implement *eof(f)* again.

The previous thoughts lead us to use the *ObjectStreams* to read and write to the files and the *PushbackInputStream* only inside *eof(f)* calls. Since Pascal does not support multithreading we do not have to take care of simultaneous access to one file with different streams and this solution should be safe.

2.8 Records

Records in Pascal are like *structs* C. They contain a set of predefined variables with their types. They may also contain an optional variant part discussed in Section 2.2.2. In this section we will analyze what options do we have to implement them.

2.8.1 Parsing

First of all, we have to decide how we will represent variables of record types. Then we must be also able to recognize that a record accessing expressions (record.element) are valid during parsing phase. Let us have the following record type:

```
record
 i, j: integer;
 c: char;
end;
```

In Pascal, this record definition says, that each variable of this type shall consist of two integer variables with identifiers i and j and one character variable with identifier c. Let us also have one variable of this type with identifier r. One option is that we will not register only identifier r to our identifiers table, but identifiers $r.i$, $r.j$ and $r.c$ as well. Thanks to the fact that Pascal identifiers cannot contain a dot, registering these identifiers will not cause any conflicts with other user defined identifiers. Advantage of this approach is that it is simple to check whether record accesses are valid. We only need to read the whole identifier sequence and look it up in our identifiers table. Disadvantage of this method is, that it does not work for arrays so it can not be used in the end.

The second approach is to let the record type variables have their own identifiers table in parsing phase and frame at runtime. This way, reading a complex record access (for example expression r2.data.innerRecord.i) will require us to check each inner variable if it is really a record and contains the specified identifier. On the other hand, it works also for arrays, so we will stick with this approach.

2.8.2 Records' variant part

A variant part in Turbo Pascal's records is like *union* structure in C language. This is a very low level construct to be natively supported by Java. These variant parts were also abused to get access to data that Pascal does not explicitly allow. For example we may define a variant part that consists of two sets of variables. One containing only an integer and the second containing only an array of bytes. This way, we are able to get the individual bytes of the integer as they are stored in the memory. Another example is that we may similarly also look at the addresses stored in pointers. For these reasons we will not support Turbo Pascal's type of variant records.

Just like the other variables in records, we will have to store those defined in the variant part in the record's identifiers table and later in the record's frame. This way, the switching between active and non-active variant parts becomes

an extra work for our interpreter. On each switch, we would have to "deinitialize" variables from each non-active part. Because of this, we have decided to implement the variables from variant record parts the same way as the others, and do not distinguish between the active and non-active ones.

2.9 Pointers

Pointers are a concept that is not natively supported by Java (the Sun's Unsafe class is marked as deprecated). Fortunately for us, Pascal standard does not support pointer arithmetic operations. What remains are functions *new()* and *dispose()* (allocation and freeing of memory space), pointer *dereferencing* and *pointers assignment*. In the following sections we will analyze two ways of implementing them. In the first one we allocate new objects to the frame and in the second one we implement our own heap and store them there.

2.9.1 Allocating to the frame

This approach is to implement pointer variables the way so that they will contain a frame where the object they point to is located, and its slot. The idea is that:

- The initial value of a pointer variable will be some singleton instance representing Pascal's *nil*,
- The new() procedure will create a new slot in the actual frame and assign it to the pointer,
- The delete() procedure will assign the nil singleton to the pointer variable,
- Dereferencing will be done by reading value from the frame stored inside the dereferenced pointer.

This approach, however, has many problems. One of them is with multiple calls of new() on the same pointer. We need to assign some identifier to the newly created objects in the frame which would not collide with already existing identifiers. What is worse, this method can create massive memory leaks. The problem is that pointers may survive a subroutine call. As an example, we may imagine a Pascal procedure that appends a new value to a linked list. Since the new pointer variable created inside the subroutine will survive the subroutine's call and it holds a reference to the subroutine's frame, the frame also survives the end of the subroutine and will not be garbage collected. As that frame holds each variable defined in that subroutine (and its arguments), it may be pretty large. This problem is illustrated on Listing 2.12. On the Figure 2.2 we can see the resulting linked list (with the first two elements highlighted) that holds reference to the frames of finished subroutines so Java's GC will not collect them unless we dispose those pointers.

Listing 2.12: Code demonstrating memory leaks of the wrapping-frame approach of implementing pointers

```
program linkedList;
type pelement = ^element;
     element = record
          value: integer;
          next: pelement;
     end;

var list, listTail: pelement;
     i, value: integer;

procedure appendToList(listTail: pelement; initialValue: integer);
var newElement: pelement; data: array[1..30000] of real;
begin
     { Some computations }
     new(newElement);
     newElement^.value := data[26270];
     listTail^.next := newElement;
end;

begin
     new(list);
     listTail := list;
     for i := 0 to 5 do begin
          readln(value);
          appendToList(listTail, value);
          listTail := listTail^.next;
     end;
end.
```

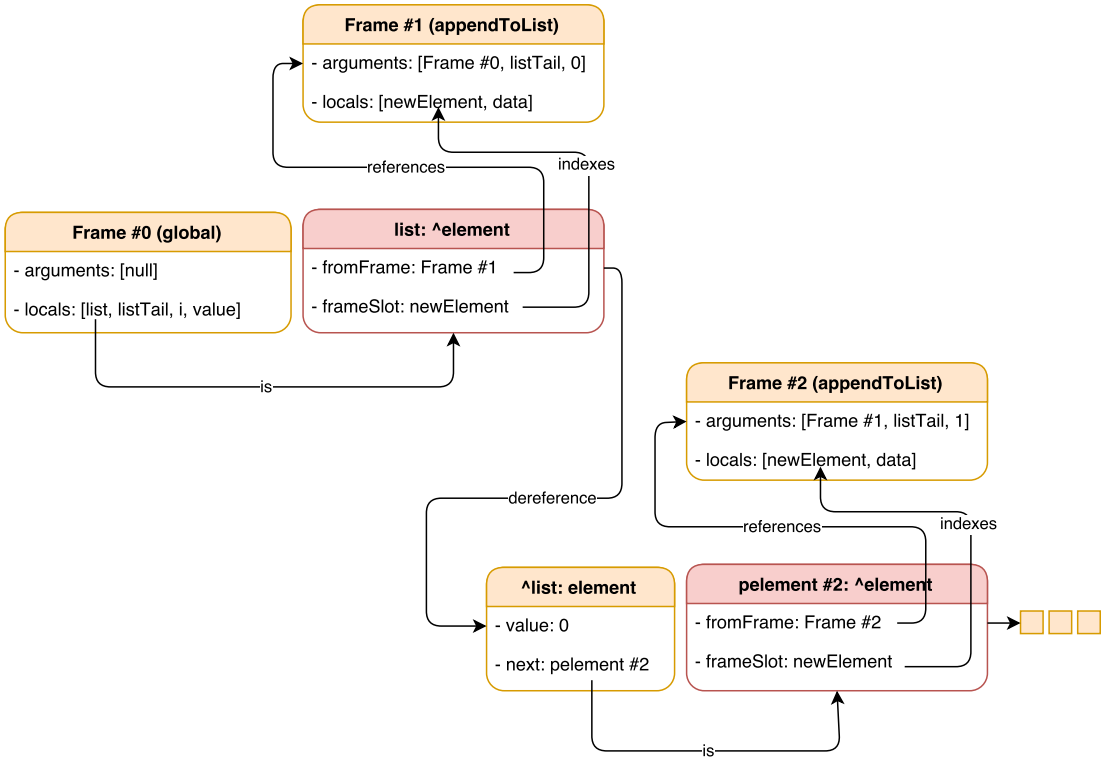

Figure 2.2: Illustration of frame leaks originating from Listing 2.12 and implementation of pointers using wrapping frames. The pointers of the linked list are colored in red.

2.9.2 Implementation with heap

The next solution consists of implementing a heap (not the binary tree-based structure but the structure used in programming languages to store non-primitive objects). Each pointer will hold an address to the heap on which some object is located. Operations on pointers will be handled the following way:

- All pointers will be initialized to point to some special address in the heap that will represent the nil value,
- The new() procedure will create a new object in the heap and make the pointer point to the address of the newly created object,
- The dispose() procedure will remove the object from the heap at address the pointer points to. If the address is empty or nill, the call throws a segmentation fault exception. The pointer will be consequently assigned to nill,
- Dereferencing a pointer will just return the value stored at the address the pointer holds. Dereferencing a nil pointer or empty address will result in an exception again.

First thing to note is that this implementation of pointers is more natural and the actual pointers are more simple. Secondly, it does not have any of the problems that the previous method had and does not introduce any new ones.

2.9.3 The heap

The main goals in implementing the heap are having access, creation and removal of the objects in O(1) time. This is easily achieved by implementing the heap internally as a Java Map with key being an integer (the "address") and the value being the object stored at that address. We do not want to use a List because the Java documentation says that removal operation on a List runs in O(n) time [3]. Another disadvantage of using a List is that we can not simply remove an object from it after a *dispose* call because we need to preserve correct indexes in the other pointers.

The secondary goal is to correctly detect the out of memory state. This state means, that the heap has already allocated an object on each address and the program is trying to allocate another one. The problem here is that when we allocate an object to the address with the highest number, we can not simply say that the heap is full. Most of the time the heap will not be full when this happens, because the program has most likely disposed an object on some of the previous addresses. To solve this issue we will implement a heap defragmentation that will rearrange the objects in the heap the way that there will be no empty places in the heap and they will be positioned from the first index.

The problem here is that the defragmentation will most probably run in O(n) time. On the other hand, we shall have in mind, that this process is called (for the first time) only after more than four billion *new()* calls (the heap size). In most applications this process will never get to be executed throughout the whole application's execution so this is not much of an issue.

Let us just point out that this solution is not fully equivalent with Pascal's pointers. On the *dispose(o)* call we do not immediately remove the object from actual memory but only allow the Java GC to collect it at some time. However, we can not get any closer to removing an object from memory in Java language so it is sufficient enough.

2.10 Goto and labeled statements

Goto is nowadays a rare branching statement. It is used in combination with labels, which allow users to mark positions in their code. Goto statements then perform jumps to one of these labeled statements.

Goto itself cannot be represented in a syntax tree because it may specify a jump to an upper level of the tree which would create a cycle in the graph. Let us look at the syntax tree on Figure 2.3.

An obvious approach to directly execute the labeled node from a goto node does not work for several reasons without changing the overall semantics of AST. The first problem is that the execution would return back to the goto node and continue with execution of all the following instructions. This can be solved but it is cumbersome. The second problem is that we would miss some instructions after the labeled statement's node (the while node and its subtree on Figure 2.3).

What we actually need to do is to stop the execution upon encountering a goto node and then we need to be able to start the execution again from the main node and omit all the nodes that are before the labeled statement's node. We can see that this method is also too cumbersome and not effective.

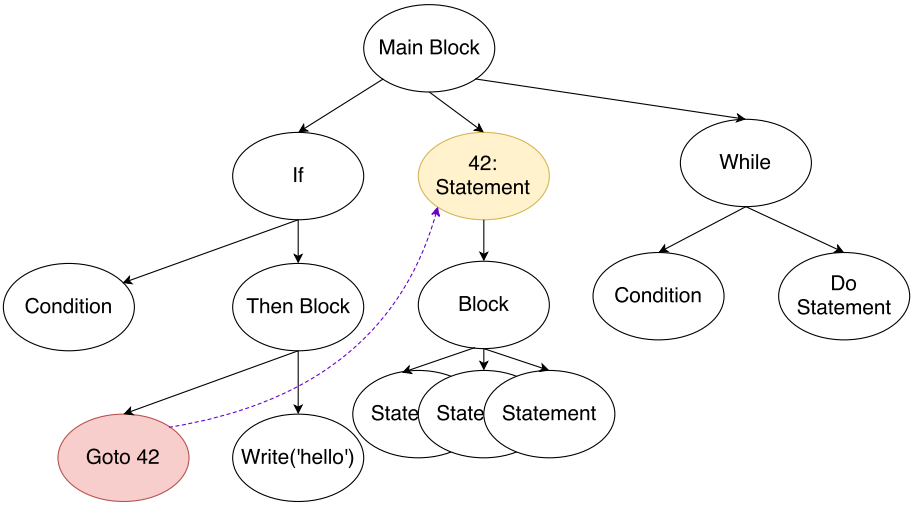

Figure 2.3: Example of a goto jump to higher levels in a syntax tree

Listing 2.13: Example of Turbo Pascal's extended goto support

```
for i:=5 to 10 do begin
    26: doSomething;
end;

while condition do begin
    goto 26;
end;
```

Another approach is to rewrite the input code by substituting the goto and labeled statements by something that is natively supported by Java because then we will surely be able to implement it in Truffle nodes. Solving this problem may seem difficult or even impracticable in some situations, but thanks to the restriction on goto statements by the Pascal standard it becomes solvable.

There is a difference between goto in standard Pascal and goto in Turbo Pascal. Turbo Pascal has no restrictions on goto statements which allows us to write jumps like in Listing 2.13. These kind of codes are not really readable or useful. On the other hand, standard Pascal's restrictions on goto are meaningful, so we will implement this variant of goto and we will *not* support goto as it is in Turbo Pascal.

Let us look at these restrictions. There are three of them and can be found in Pascal's standard [1]:
"A label, if any, of a statement S shall be designated as pre fixing S. The label shall be permitted to occur in a goto – statement G if and only if any of the following three conditions is satisfied.

- S contains G
- S is a statement of statement sequence containing G
- S is a statement of the statement sequence of the compound statement of the statement part of a block containing G."

Listing 2.14: Example of a goto jump to statement that contains the goto

```
42: if condition then begin
        statement1;
        while condition2 then begin
            goto 42;
        end;
    end;
```

The last one may be a little difficult to understand but it simply means that it allows goto jumps to any position in the topmost block of the subroutine containing the goto.

In the following sections we will look at these rules more like on a features than restrictions and analyze them separately. We will be trying to rewrite all of these three situations to a code that does not use goto nor label statements but instead of it it uses statements that are supported by Java. We will then take the resulting Java code and create an AST of it, which will be used in our interpreter.

The first situation is illustrated in Listing 2.14. We have some labeled statement and we want to be able to execute it again if a goto is executed inside it. Firstly, we need to stop the execution at the goto statement. We can achieve this by throwing an exception when executing it. This exception will be caught inside the labeled statement's node. When it is caught, it means that the goto statement was executed, and that we want to execute the labeled statement again. This may happen an arbitrary number of times, so we wrap this whole new bunch of code inside an infinite loop and add a break statement to the end of it. If the goto exception is caught, the break statement does not get executed and we execute the labeled statement again. On the other hand, if it is not caught, it means that goto was not executed, our break statement gets executed so the loop will finish and we continue with the interpretation with the following node.

We also need to check whether the goto statement referenced the label of our labeled statement or a different label. This may be achieved by including the label referenced by the goto inside the goto exception. The resulting rewritten code of the given example in Listing 2.14 in a side to side comparison can be seen on Figure 2.4.

To implement this behavior, we do not need to change the syntax tree, we only add a specialized statement node for labeled statements which will contain the mentioned loop, try-catch and the original statement. Our parser will generate these specialized nodes for each statement prefixed by a label.

The second and third situations are more complicated. Firstly, we generalize them into the following statement (using the same notation), which will simplify their implementation:

S is a statement of a statement sequence containing a statement that contains G in any depth.

This definition allows a bit more than the standard. It allows goto to jump to nodes in any level higher than itself. The standard says that it can jump only one level above or to the top level of a subroutine. We will solve this inconsistency later.

Figure 2.4: Example of a rewritten goto inside a statement to Java statements

```
while (true) {
  try {
    if (condition) {
      statement1;
      while (condition2) {
        throw new GotoException(42);
        break;
      }
    }
  } catch (GotoException e) {
    if (e.labelId != 42) {
      throw e;
    }
  }
}
```

```
42: if condition then
    begin
      statement1;
      while condition2 then
      begin
        goto 42;
      end;
    end;
```

Listing 2.15: Example of labeled statements in a block

```
begin
    statement1;
    statement2;
    42: labeledStatement3;
    statement4;
    43: labeledStatement5;
    statement6;
end;
```

Figure 2.5: Example of a rewritten goto inside a block to Java statements

```
int jump = 0;
int[] labels = { 42, 43 };
while (true) {
    try {
        switch(jump) {
            case 0:
                statement1;
                statement2;
            case 42:
                labeledStatement3;
                statement4;
            case 43:
                labeledStatement5;
                statement6;
        }
        break;
    } catch (GotoException e) {
        if (labels.contain(e.labelId)){
            jump = e.labelId;
        } else {
            throw e;
        }
    }
}
```

```
begin
    statement1;
    statement2;
    42: labeledStatement3;
    statement4;
    43: labeledStatement5;
    statement6;
end;
```

Listing 2.15 shows us a block containing a number of statements from which some are prefixed by a label. Any of these statements may contain a goto inside them. This or these goto statements may jump only to either *labeledStatement3* or *labeledStatement5*. These jumps may be forward or backward in the code. Once again, we will be throwing a goto exceptions with identifier of the referenced label. If an exception is thrown when executing any of these statements, we will have to stop the execution of the current block node and start its execution from the statement that is prefixed with the correct label. For this purpose, we will separate our block into smaller pieces with labeled statements serving as separators and wrap these pieces into a switch statement without breaks. When a goto exception is thrown, we execute this switch statement, which will start executing the labeled statement and execute every following statement thanks to the missing breaks. This may again happen multiple times so we will again wrap the whole new code inside an infinite loop with break at the end. Figure 2.5 shows these code adjustments in a side by side comparison with the original code.

Because these code adjustments also change syntax tree (which can be seen on Figure 2.6) by splitting block nodes into smaller block nodes, they are pretty robust and they slow down the syntax tree creation (in every block node!). On the other hand, not every program use goto statements of this type. For this reason, we add an option to turn on or off these code adjustments.

Now we are only left with the before mentioned inconsistency with the standard. This inconsistency, however, may be left in the implementation because we find it as a reasonable extension to the language.

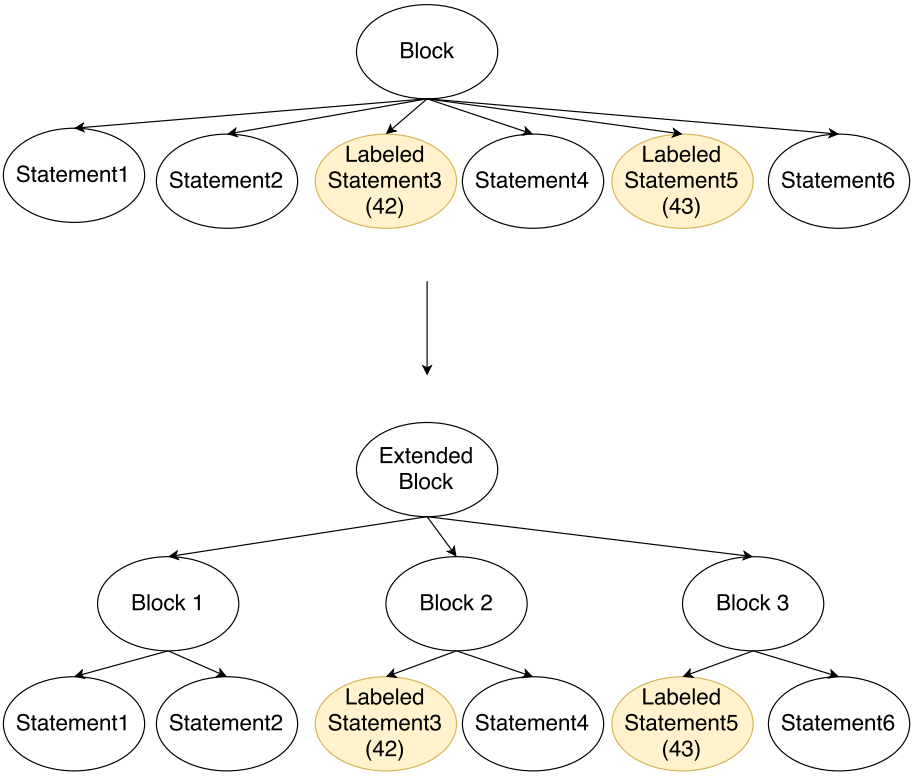

Figure 2.6: Recreation of syntax tree after applying changes to support goto inside a block

2.11 Interoperability with Truffle languages

In this section we will discuss our possibilities of integration of our interpreter into Truffle VM. We will analyze possibilities of interoperability of Pascal sources with other Truffle based languages (and Java itself). To achieve this, we have to firstly look at how can we pass inputs to a Pascal code and how to produce outputs. Next, we will have to make the output compatible with other Truffle languages.

2.11.1 Program arguments

As already mentioned in Section 2.2.7, we may define input arguments of a Pascal program. These arguments, however, may be only of string or file type. If we want to achieve better and more usable interoperability we will have to be able to pass more types of inputs to our program.

If we keep only string and file type inputs allowed, the user will have to encode each input to a string and then decode it in Pascal source, or store inputs in files and then read it inside the program. There is a problem with storing input arguments in files when they have different types. They will have to be either separated into multiple files, which leads to unclear code and use of more OS resources, or saved in one text file which leaves us again with the encoding and decoding of each argument.

Better solution seems to be to allow the user to use more types of program arguments. This solution, on the other hand, is problematic with custom types

Listing 2.16: Returning a value from main block using assignment to program identifier

```
program add(a,b): integer;
var a,b: integer;
begin
    add := a + b;
end;
```

like records or enums. How to pass a record from non Pascal language, and what if the user does not even declare this record in his source? To avoid these problems, we will allow only primitive types as program arguments and the original string and file types.

2.11.2 Program output

Implementing output of a Pascal source is accompanied by even more problems. The major problem is that the main program block of a standard Pascal code does not return any value.

One solution to this issue is that we can extend Pascal's syntax and semantics. Instead of treating Pascal's main block as a procedure block, we may treat it like it is a function whose identifier is the identifier of whole program. This way, we can return a value like from any other function. We may also consider whether we want to specify a return type or simply let the user return any type. This idea is represented in Listing 2.16. The problem with this approach is breaking the standard's rules and the fact that Turbo Pascal source does not require using the *program statement* at the beginning.

Another approach is to not execute the main block when interpreting a code, but some function from that source. The identifier of the function that is to be executed may be "hardwired" or we may let the user specify which function he wants to execute. This solutions solves the problem with violating Pascal's standard, but introduces another problems. The first one is, what to do with the main block? Since it will never get to be executed, it may be always left empty which is rather odd. The second problem is, if we let the user choose which function to execute, that the AST which we will be executing will have to differ depending on which function the user chose. This is because we have to execute our initializations nodes[1] at the beginning of the interpretation, so they will have to be placed at the beginning of the executed function. They also initialize global variables, which means that if the user recursively calls this function, each such call would reset values of all global variables.

The last solution takes advantage of Turbo Pascal's *halt* function. The function stops program execution and returns a number it got as an argument. A big advantage of using this function to solve our issue is that it doesn't break Pas-

[1]The non primitive type variables in Truffle's frames have to be initialized to *null*. Because of this, a node that reads one of these variables may also return *null* from its execution if the variable was not assigned before. In addition, if such node appears as a child of some other node which uses specializations, Truffle is unable to choose the correct specialization and throws an exception. To solve this issue, we add one initialization node for each variable in each frame which initializes the variable to its default value.

cal's standard and does not create any weird constructs (like empty main block). On the other hand, it can return only integer values and may be used only with Turbo Pascal extensions turned on.

From these three options we have chosen the third one (the *halt* function) because it seems that its advantages outweigh disadvantages.

2.12 Built-in units

The Turbo Pascal's built-in units that we have decided to support in our interpreter are *graph*, *crt*, *dos* and *strings*.

The *graph* unit allows its users to implement some simple graphics in their applications. After its graphic mode is initialized, it makes the application run in fullscreen and provides access to simple drawing procedures like *Bar*, *Ellipse*, *Arc* and many others. We have multiple choices of simulating this mode. We may either use some third party graphic library for Java, or we can use some of the Java's built in GUI libraries capable of rendering some graphics. Because the performance of the graphic mode is not one of our primary goals we will stick with Java's *Swing* GUI library to keep it simple.

The *strings* unit provides extended support for usage of strings in Pascal. The standard *string* type is limited in size (255 single-byte characters) and its size is stored in the byte at index 0. The *strings* unit comes with new string type which is *PChar* and is null-terminated. We have to keep in mind that it is a whole new type which is basically a pointer to *char* type. For better usability, the usage of *strings* unit extends Pascal's syntax. For example, we are able to assign a string literal to a *PChar* variable or implicitly convert string to *PChar*.

We can again choose from different approaches to implement these built-in units. Either we will implement them whole in Java or we may implement only some core subroutines in Java and then implement the others in a Pascal unit which will be part of our interpreter. Implementing them whole in Pascal is not possible, mainly due to the mentioned updated syntax of *strings* unit or simulation of graphics in *graph* unit. In order to demonstrate usability of units in our interpreter, we choose to implement only the necessary parts of each unit in Java and the rest in Pascal units.

3. Programming documentation

In this chapter we will look at the requirements of our project for compilation or further development. We will also present basic structure of the project, describe some important sources and connections between classes. We have also implemented well known Tic Tac Toe game that uses our interpreter. Its implementation is described later in this chapter.

3.1 Project dependencies

The project is written in Java version 8. There are four dependencies required for its compilation. These are:

Truffle API (version 0.26) which represents the Truffle framework. Because version 0.26 is still a beta version, it is possible that our project will not be compatible with newer versions of the framework, so we recommend using this specific version,

Truffle DSL Processor (version 0.26) is responsible for generating some source files, mainly specialization nodes. The project can not be compiled without them,

Args4j (version 2.33) is used to parse command line arguments provided to our interpreter, thus is also required,

JUnit (version 4.11) is used only for unit testing. This dependency is required only to run our tests. The project can be compiled and run without it.

Our project uses Maven project management and comprehension tool. All of the previously mentioned dependencies are specified in the project's build file (*pom.xml)*, so they will be automatically downloaded from the maven repository when this tool is used to build the project.

3.1.1 Other requirements

The Truffle DSL is an annotation processor, so the annotation processing must be turned on. Some IDE's have this option turned off by default and are even unable to automatically recognize the annotation processors from sources/JAR files on classpath. Because of this, we list all the required annotation processing classes below:

com.oracle.truffle.dsl.processor.InstrumentableProcessor
com.oracle.truffle.dsl.processor.InstrumentRegistrationProcessor
com.oracle.truffle.dsl.processor.interop.InteropDSLProcessor
com.oracle.truffle.dsl.processor.LanguageRegistrationProcessor
com.oracle.truffle.dsl.processor.TruffleProcessor
com.oracle.truffle.dsl.processor.verify.VerifyTruffleProcessor
com.oracle.truffle.object.dsl.processor.LayoutProcessor

For further development and extension of our grammar, the user also needs the Coco/R tool to generate new versions of parser from updated grammar files.

This library may be freely downloaded from the Internet [4] along with its user manual.

3.2 Basic code structure

Because our project is a Maven project, it uses common Maven structure. All sources are located inside *src/* folder except Pascal sources of built-in units which are located inside the *builtin/* folder. The *src/* folder is then divided into two folders *test/* and *main/*, from which the first mentioned contains all of our unit tests and the second one sources of our interpreter.

3.3 Interpretation proccess

This section is a basic overview of the whole interpretation process. As we get further in the thesis, we will describe each component in more detail.

The process is depicted on Figure 3.1. The *CompilerMain* class contains the entry point of our interpreter. It receives a Pascal source file as an input and options like which standard to use or which directories to include. It creates an instance of *PolyglotEngine* [5] which is responsible for evaluating sources of registered Truffle languages. The representation of Pascal in Truffle is our *PascalLanguage* class. This class is responsible for creating a call target when it gets a source file which will be executed by the engine. Our *Parser* class generated by Coco/R from grammar file is responsible for parsing and returning the root node of the source it received.

3.4 Entry point

Entry point of our interpreter is located inside the *CompilerMain* class. When it is launched, it has to firstly evaluate all the included units using the Polyglot engine. This evaluation results in registering all the subroutines, variables and custom types into our global scope. We have separated this process into another class – *UnitEvaluator*. Then it evaluates the input source code.

To be able to call a script with some arguments, the initial evaluation of it returns a function, which when executed, executes the main program block's root node. This function may be executed inside Java with arbitrary number of arguments.

All the optional and required arguments are handled by *Settings* class with help of *args4j* library. Format of these options and their usage is described in the User Documentation chapter.

Module decomposition diagram of these classes is depicted on Figure 3.2.

3.5 Registering to Truffle

To register a language to Truffle, we annotate a class that represents it with *@TruffleLanguage.Registration* and make it extend *TruffleLanguage* class. The annotation also contains a mime type as an argument, thanks to which

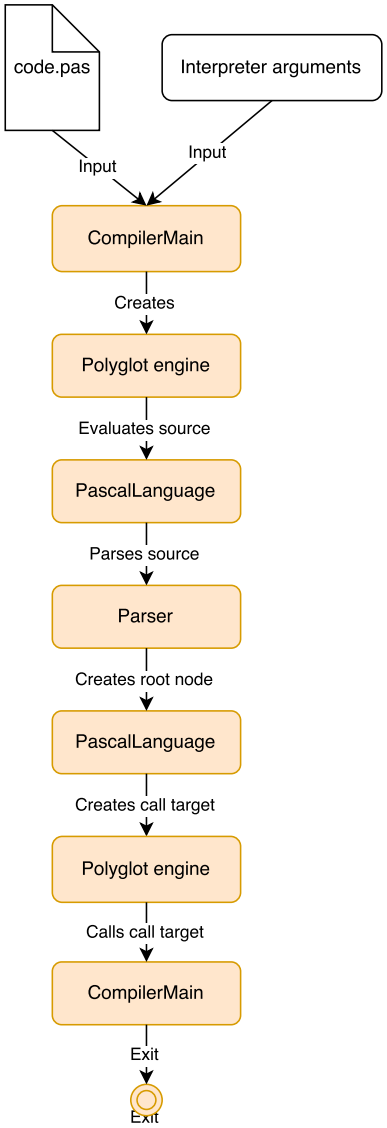

Figure 3.1: Diagram of the interpretation process in *Trupple*

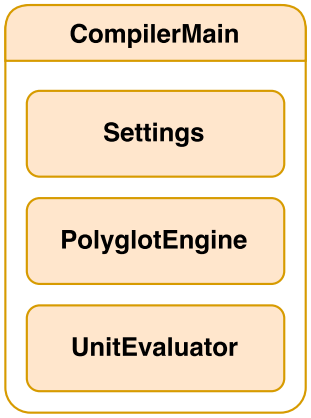

Figure 3.2: Module decomposition diagram containing class with *Trupple's* entry point

the polyglot engine is able to choose corresponding language of the source it

Listing 3.1: Declaration of an implicit cast. Can be found in PascalTypes.java

```
@ImplicitCast
public static double castIntToDouble(int value) {
    return value;
}
```

receives.

By extending *TruffleLanguage* class we have to implement methods that are responsible for parsing a source file and cooperating with other languages. It is also a typed class. Instances of the class that we type with our language class represent some global context or state of the language.

The parsing method simply parses the source using our parser (which returns its AST's root node), creates a call target for the root node and returns it.

3.5.1 Type system

Truffle needs to know which types are we going to use in our language, so we need to specify our type system class. This class has to be annotated with *@TypeSystem* annotation. The class that represents our types is *PascalTypes*. The annotation contains list of classes that define our types (e.g.: long.class, boolean.class, char.class, ... and some of our own classes that represent Pascal specific types like *RecordValue*, *PointerValue*, etc.).

Another thing that we may define in this class are implicit type casts. As an example, let us look at the addition node and at its numeric type arguments. It may contain an integer type on any side of the operator but also real types or they may occur in any combination of these. This leads us to definition of four specializations for this node with each possible combination of integer and real types as operands. However, we have defined that integer type may be implicitly cast to a real type (which we represent by double in our implementation). This way, number of the required specializations is reduced. This implicit cast is defined by *@ImplicitCast* annotation on an arbitrary function that does this cast (takes an integer value and returns a double value). The implemented method can be seen in Listing 3.1.

3.6 The parser

In this section we describe our implementation of parsing a Pascal source file. Hierarchy of classes that take care of this process is depicted on Figure 3.3.

The *Parser* and *Scanner* classes are generated by Coco/R. *NodeFactory* is our class that generates our AST nodes and holds actual parsing state including *LexicalScope* which is a slight wrapper on *IdentifiersTable* which holds all defined identifiers and their types.

We have two parsers (and scanners), one is used for parsing standard Pascal sources and the other for parsing Turbo Pascal sources as it was decided in the analysis in Section 2.4. There is also an *IdentifiersTableTP* class which extends the *IdentifiersTable* and adds some built-in identifiers from Turbo Pascal which are not present in the standard Pascal (for example *string*, *random* and *assign*).

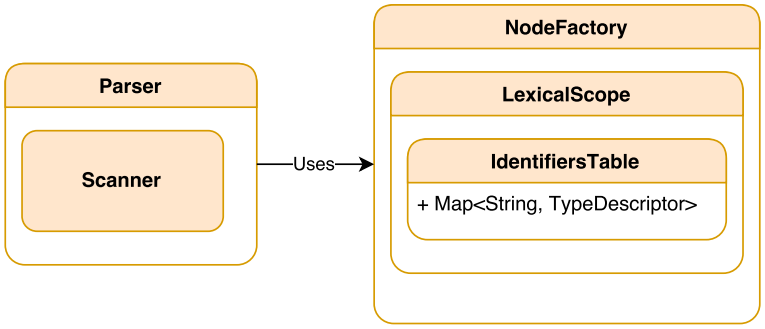

Figure 3.3: Module decomposition diagram of classes dealing with parsing a Pascal source

Both parsers consist of the following files: *Pascal.atg, Parser.frame, Scanner.frame, Parser.java and Scanner.java*. Coco/R uses the .atg file and both .frame files to generate a parser (Parser.java) and a scanner (Scanner.java).

3.6.1 Grammar file

Pascal.atg is file containing our grammar and there are two of them. These grammar files are separated into three main parts: compiler name, scanner specification and parser specification.

The scanner specification consists of Coco/R options and definitions of characters and tokens. The only option that we are using is *IGNORECASE*, which specifies that the scanner shall be *case insensitive*.

In the parser specification part we declare our grammar rules. Cocor/R's grammar rules use EBNF. We also declare semantic actions written in Java inside these rules. We mostly use these Java codes to call methods from our *NodeFactory* class to generate AST node representing currently parsed rule.

3.6.2 Frame files

The Coco/R requires two additional files to create the parser and scanner classes. These files are Parser.frame and Scanner.frame. They are like outlines for the generated java files. The contain mostly Java code (class definitions) and some markers.

The Parser.frame contains definitions of multiple classes: *Parser*, *Errors* and *FatalError*. When Coco/R generates the resulting Parser.java file it copies innards of this file and inserts grammar rules converted into Java methods inside the Parser class. We have added some custom Java functions inside this frame file that are used as conflict resolvers in the grammar rules.

The Scanner.frame is layout for Scanner class and some other classes used inside it (for example Token and StartStates). This frame file was not changed.

3.6.3 Table of identifiers

To hold informations about each declared identifier and its type we have implemented *IdentifiersTable* class. Its core consists of a map from String representing

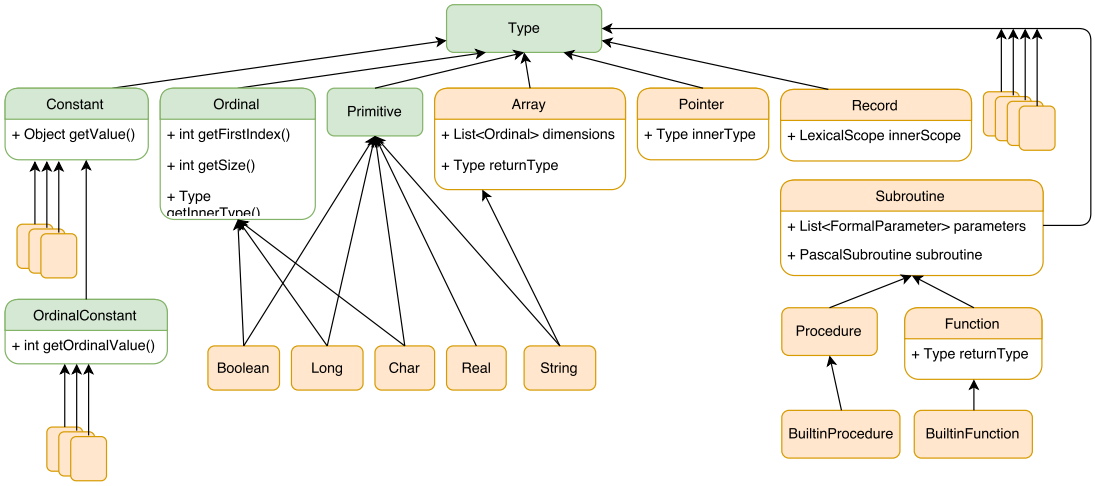

Figure 3.4: Simplified hierarchy of our type descriptors. The green modules are interfaces and the orange ones are classes. The "Descriptor" suffixes in their names are omitted.

the identifier to a type descriptor which holds its type informations. When instantiating this table, it automatically inserts all built-in identifiers like types (integer, char, boolean, ...) and built-in subroutines (write, read, ord, ...) to its data.

3.6.4 Type descriptors

Each built-in or custom defined type is represented by an instance of *TypeDescriptor* interface. The two main purposes of these descriptors are:

- Holding some additional information about the type (for example inner identifiers of record types, bounds of an array type or subroutine's signature),
- Compile time type checking.

To achieve fast type checking, all type descriptors of primitive types are implemented as singleton instances. This way, when we want to check if an expression is, for example, a boolean, we just compare the expression's type descriptor with boolean descriptor singleton. A simplified hierarchy of type descriptors can be seen on Figure 3.4.

3.7 Syntax tree nodes

In this section we describe implementations of our syntax tree nodes. Each node must extend Truffle's abstract *Node* class. If a node contains any children nodes they must be properly annotated, so that Truffle is able to build the resulting AST. A simplified diagram of inheritance hierarchy of our nodes can be seen on Figure 3.5. We will not describe each node that we have implemented

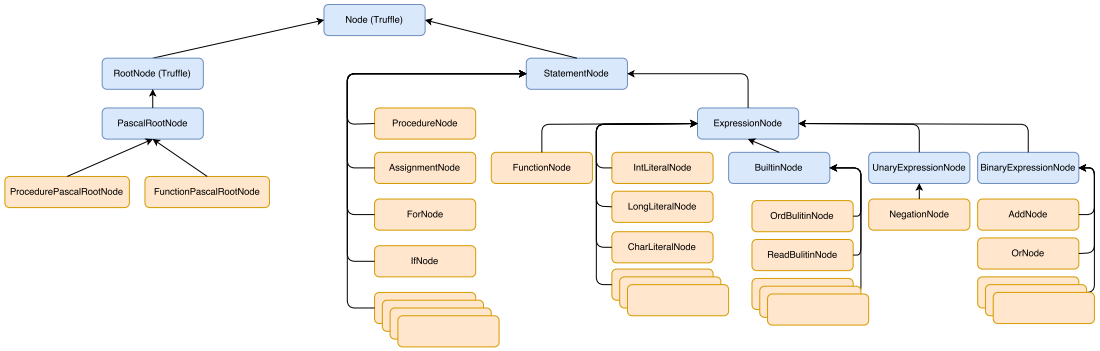

Figure 3.5: Hierarchy of our node classes. All the abstract classes except those containing specializations are colored in blue.

because the thesis would get unnecessarily long and not all of them are that interesting from the implementation point of view. Only the most important nodes are described in detail.

3.7.1 Root node and call target

A root node is the topmost node of an AST. We build one for each subroutine (including the main program block). Each root node must extend Truffle's abstract *RootNode* class. This *RootNode* contains a *RootCallTarget*. This call target is much like entry point of a callable object which are only subroutines in our case. It is created using Truffle's *Truffle.getRuntime().createCallTarget(RootNode rootNode)* method. As we can see, we must provide our root node to the call target. This way, a root node contains reference to its call target and call target holds a reference to its root node. In Truffle, we are not executing root nodes but calling their call target using *call(Object[] arguments* method on it. This method automatically creates a new frame for the called object and fills the frame's arguments list with the arguments the *call* method received. The frame is then pushed to the Truffle's stack. Afterwards, the call target executes the root node by calling its *execute()* method which we had to implement ourselves. We have implemented abstract *PascalRootNode* class that extends the Truffle's abstract *RootNode*. This root node also has to return a resulting value of the call. In our case it is return value of a function call.

3.7.2 Pascal root nodes

We have three implementations of root nodes. One for procedures, one for functions and one for the main program block. All of them extend the *PascalRootNode*. The reason to differentiate between these is that a root node has to return a value when it finishes its execution. The *FunctionPascalRootNode* returns the return value of the function call and the *ProcedurePascalRootNode* executes its body and returns null. The *MainFunctionPascalRootNode* has to catch *HaltException* (if any), which is thrown after executing Turbo Pascal's *halt* procedure, and return its *return code* or 0 if the exception is not thrown (*halt* was not called).

3.7.3 Statement node

StatementNode is a base class for each of our nodes except only root nodes. Statement nodes do not return any value in Pascal (for example if statement, while statement, assignment, ...). It contains an abstract method

```
void executeVoid(VirtualFrame frame)
```

which is called when executing any of these nodes. Functionality of statement nodes is implemented inside this method. The method receives current frame as an argument, through which we can access our variables. We also had to annotate this class with *@TypeSystemReference(PascalTypes.class)* annotation, so Truffle knows what type system it uses. This information is inherited to each subclass so we do not have to annotate other node classes as far as they implement the *StatementNode*.

3.7.4 Concrete statement nodes

The following list describes some of the basic statement nodes:

BlockNode represents a block in a code. Blocks in Pascal are sequences of statements inside *begin* and *end* keywords. These inner statements are represented by their statement nodes and are stored in a list inside this block node in the same order that they appear in the source code. When a block node is executed, it simply executes each children node in the list,

ProcedureBodyNode represents body of a procedure. It consists only of one block node that is executed when the body node is executed,

WhileNode and *BreakNode*. *WhileNode* represents a while loop. It contains two children nodes: a *BlockNode* which is the body of the loop and an *ExpressionNode* which represents the loop condition. We always check whether it is a boolean expression in the parsing phase. When this node is executed, it uses Java while statement with condition being the return value of the condition node and body of the loop being the body node. When a *BreakNode* is executed, it only throws an *BreakException* which is caught in every loop node so it gets correctly caught in the "closest" cycle to the break statement. When the exception is caught we also break the Java cycle inside the node finishing its execution. The *BreakException* extends Truffle's *ControlFlowException* so Graal can optimize it,

*ExtendedBlockNode*s are used instead of *BlockNode*s when extended goto support is turned on. They are implemented just like we have described them in the Analysis chapter in Section 2.10. The only difference is that we can not really use switch statement because there may be variable number of the inner blocks so we have replaced it with a for loop,

GotoNode contains a label to which it should jump. When it is executed, it throws a *GotoException* with the label identifier,

Assign nodes are used to assign a value to a variable stored inside Truffle's frame. Firstly we have to know the variable's slot in the frame. Because

we are already building a *FrameDescriptor* (a structure that holds information about how the resulting frame will look like) during the parsing phase (for each subroutine), we know the slots of all our variables. When we are instantiating a *SimpleAssignmentNode*, we provide it a slot of a variable to which we want to assign a value, which is represented by an *ExpressionNode*. We do the type checking in the parsing phase by comparing types of the *ExpressionNode* and the variable. When the *SimpleAssignmentNode* is executed, it assigns the value from the expression node to the variable on the specified slot. This node takes care of assigning a value only to some primitive type variable. There are other nodes for assigning to a reference, array, pointer or record because in these assignments we are not assigning a value to a slot inside a frame but we are assigning a value to our own representation of these types.

3.7.5 Expression node

Another important base node class that we implemented is *ExpressionNode*. It is a base class for each node that shall return any value. It extends the *StatementNode* and adds another abstract execute methods like

```
boolean executeBoolean(VirtualFrame frame) { ... }
long executeLong(VirtualFrame frame) { ... }
chat executeChar(VirtualFrame frame) { ... }
. . .
```

However, we do not implement these type specialized execute methods for most nodes directly because we are using Truffle's specializations which generate their own execute methods so they match signature of our specializations.

3.7.6 Specializations

Each node that uses specializations has to be abstract. This way, Truffle can finish their implementations in its own generated classes by adding argument type checks and choosing of the right specialization to use at runtime. Truffle gets these argument values by executing node's children nodes in the order they are defined in our abstract nodes. The resulting values are then passed into our specialization as arguments. If there is no specialization that matches received combination of argument types Truffle throws an exception.

3.7.7 Binary expression nodes

The nodes that represent binary and unary operations use Truffle's specializations heavily. A *BinaryNode* class is a base class for each node that represents an operation with two operands. It specifies that is has two children nodes which are *ExpressionNode* instances and represent values of the operands. We have also implemented type checking methods here. They are implemented the way that each subclass has only to define table which specifies which combinations of arguments it takes and what is the resulting type of each combination.

Example of nodes that inherit from the abstract *BinaryNode* are *AddNode*, *OrNode*, *SymmetricDifferenceNode* and many others.

3.7.8 Concrete expression nodes

The following list contains chosen expression nodes and their descriptions:

FunctionBodyNode represents body of a function. It contains one child node, a *BlockNode*, which is executed along with this node. We also have to secure returning correct value from a function call. We store a variable with the same identifier as the function in the function's frame. This way, we can assign to the function's identifier and the return value of the function call will be the value of that variable. We also had to make the variable write only by adding proper checks inside our *NodeFactory*,

InvokeNode – at runtime, we store the root nodes of all defined subroutines inside the frame of the scope they are defined in. The *InvokeNode* is used to look them up and invoke them. It gets the subroutine's slot in a frame when it is instantiated. At its first execution, it extracts the subroutine's root node from the frame and caches it. In each its consequent execution it directly invokes the cached subroutine. Before the subroutine's root node is executed, its arguments are evaluated. They are represented as the *InvokeNode*'s children nodes and are instances of *ExpressionNode*. We execute these nodes and store the values they return in an array, insert reference to the current frame into the beginning of the array and finally execute the root node's call target providing it the array of arguments. This way, the inner subroutine is able to access its caller's frame. The resulting value of the whole call is then returned. We also check, if the given argument types match the subroutine's signature during the parsing phase,

StoreReferenceArgumentNode – when we want to pass a variable by reference to a subroutine, we cannot pass only its value. Instead of that, we wrap the frame in which the variable lives and its slot into a *Reference* instance. This reference is then passed to the subroutine as an argument. This is why we had to implement special read and write nodes for references because we can not simply write or read value of the variable from the current frame but firstly we have to extract it from the *Reference* object. The node that takes care of wrapping specified variable into a *Reference* object is *StoreReferenceArgumentNode*. Similarly, when we want to pass a subroutine as an argument, we pass a *Subroutine* object that represents the subroutine (contains its root node). These objects are already created during the parsing phase so we do not have to add special node for wrapping them,

ReadArgumentNode – when we pass some value as a subroutine's argument, we need to assign this value with the corresponding argument variable. This can be achieved by *SimpleAssignmentNode*. The passed arguments are stored in list of arguments inside the called subroutine's frame (thanks to the *InvokeNode*). To read that values from the frame we have implemented a *ReadArgumentNode*. This node receives index of the argument which it should read. This index is incremented by one because we have inserted also caller's frame as the very first argument. Having this node, we only iterate over each variable declared as subroutine's formal parameter, create assignment node that assigns the value read with the *ReadArgumentNode* to

the argument variable and insert these assignment nodes to the beginning of each subroutine,

ReadAllArgumentsNode – each subroutine has exactly one AST including *read*, *readln*, *write* and *writeln* subroutines. The problem with them is that we cannot insert the formal parameter assignment nodes to the beginning of them because they may be called with arbitrary number of arguments. We also cannot simply use specializations for their nodes because they must have fixed number of arguments. For this purpose, we have implemented *ReadAllArgumentsNode* that reads all arguments from the current frame and returns array of values it read (Object[]). This node is the only child node of these subroutines. Then, we could implement one specialization for them which receives an array of Objects as single argument. Inside the *write* and *writeln* nodes we only print values from this array to standard output or to a file if the first argument is a file type. *Read* and *readln* subroutines receive references of variables into which we write values read from the standard input or file if the first argument is a file type.

3.8 Heap

Implementation of our heap consists of *PascalHeap* and *HeapSlot* classes. We have decided in the Analysis chapter in Section 2.9.3 that when our heap reaches the last memory slot, we want to defragment it, so we do not reach out of memory state if there are still some empty slots that may have been disposed before. The problem is that defragmentation itself would break pointers. We need to ensure that pointers point to the right addresses even after this process. We also want our solution to be transparent for the pointers, so they are not aware of this process.

Because of this we have implemented heap slots. Instead of pointers holding the direct address to the heap memory, they hold a heap slot which contains the actual address. The heap then holds a set of references to all heap slots that are currently in use. The defragmentation then iterates over this set instead of whole address space and changes the addresses inside the slots. This way, each time an object is moved, the pointer will still point to the correct address through its heap slot.

We have also achieved full transparency for pointers and more security to the heap memory because the memory can not be accessed directly, and heap slots may be created only by the heap itself.

3.9 Units

Units are implemented the way that they are assigned their own instance of *VirtualFrame* in which all their subroutines and variables are stored. These frames are accessible via *PascalLanguage.INSTANCE* singleton. Thanks to this, they are globally accessible. It also means, that if we would have some long running application that is executing Pascal scripts using our interpreter and use some units, these units do not need to be parsed each time. They only need to

Listing 3.2: Including java part of a built-in unit

```
unit graph;
uses graphcorejava; { the java part of the unit }

interface
procedure Bar(x1, y1, x2, y2: integer);
procedure Bar3D(x1, y1, x2, y2, depth: integer; top: boolean);
procedure SetColor(color: integer);
. . .
```

be parsed once to create their frames and store it "inside our language". The scripts can be then evaluated independently.

3.9.1 Built-in units

The built-in units, which we have decided to support were *not* fully implemented because they are pretty large and implementing all of their subroutines would not bring more academic contribution then implementation of only some of them.

More interesting fact is that we have implemented some parts of these units directly in Pascal. The core subroutines that could not have been implemented in Pascal are implemented as subroutine nodes in our interpreter and stored in implementations of *BuiltinUnit* interface. These implementations serve as containers for Java content of the built-in units. These subroutines can be included in the Pascal part of the units just like any other unit by using the *uses statement*. When our parser encounters the mentioned statement, it firstly looks if the specified unit is one of these core units. If it is, the corresponding implementation of *BuiltinUnit* takes care of importing this core content into units' lexical scope of currently parsed source. If not, it looks whether a unit with the specified identifier was parsed before and imports it to units' lexical scope or throws error if unit with that name was never seen by the interpreter before. Including of unit from another unit can be seen in Listing 3.2.

3.9.2 Graph mode

One of the built-in units that we have implemented is the *graph* unit. This unit enables user to enter a fullscreen graphics mode. Our simulation of this mode is done inside *PascalGraphMode* class. Module decomposition diagram of this class can be seen on Figure 3.6. The mode is simulated using Java's standard GUI library Swing and its *JFrame*. The frame contains one *JPanel* on which the graphic content is drawn.

The frame contains infinite drawing loop and is running in a separate thread to not block the code interpretation. Demonstrational usage of the unit can be seen on Figure 3.7 which is a result of executing code in Listing 3.3.

Listing 3.3: Example of usage of Pascal's graph mode

```
program graphics;
uses graph;
const red = 16711680; green = 65280; blue = 255; yellow = 16776960;
    size = 200; margin = 20; leftMargin = 70;

procedure drawit(color, left, top, size: integer);
const count = 15;
var cleft, ctop, margin, colordiff, i:integer;
begin
    colordiff := color div count;
    for i:=1 to count do begin
        setcolor(color - i*colordiff);
        margin := i*5;
        cleft := left + margin;
        ctop := top + margin;
        bar(cleft, ctop, cleft + size - 2*margin,
            ctop + size - 2*margin);
    end;
    {closegraph;}
end;

begin
    initgraph(0,0,'');

    drawit(red, leftMargin + margin, margin, size);
    drawit(green, leftMargin + size + 2*margin, margin, size);
    drawit(blue, leftMargin + margin, size + 2*margin, size);
    drawit(yellow, leftMargin + size + 2*margin, size + 2*margin,
        size);
end.
```

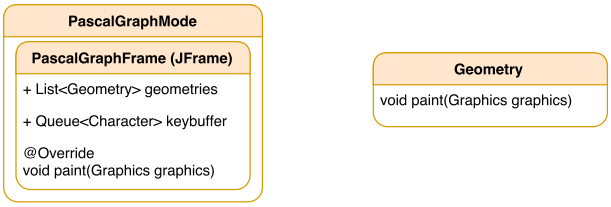

Figure 3.6: Module decomposition diagram of classes that are used for simulation of our graph mode.

Figure 3.7: Example of an output from the Pascal's graph mode.

3.9.3 Support

In this section we list supported content of built-in units. Each of the unit is also divided into two parts, one for content implemented in Java and the second for content implemented in Pascal.

- *CRT* unit
  - Java
    - Delay(i: integer) – delays program execution,
    - KeyPressed() – checks whether there is a keypress in the keybuffer,
    - ReadKey() – reads key from keybuffer,
- *DOS* unit
  - Java
    - Exec(cmd, args: string) – executes another program, and waits for it to finish,

        * GetDate(y, m, d, w: longint) – gets the current date,
        * GetMSCount() – gets number of milliseconds since a starting point,
        * GetTime(h, m, s, sh: longint) – returns the current time,

- *STRINGS* unit
  - Java
    * PChar type,
    * StrAlloc(size: longint) – allocates memory for a new null terminated string on the heap
  - Pascal
    * StrCat(dest, source: PChar) – concatenates 2 null terminated strings,
    * StrComp(str1, str2: PChar) – compares 2 null terminated strings (case sensitive),
    * StrUpper(str: PChar) – converts null terminated string to uppercase,

- *GRAPH* unit
  - Java
    * InitGraph(gd, gm: longint; ptd: string) – initializes graphic mode,
    * CloseGraph() – closes graphic mode
    * PutPixel(x, y, color: integer) – colors specified pixel
  - Pascal
    * Bar(x1, y1, x2, y2: integer) – draws filled rectangle,
    * Bar3D(x1, y1, x2, y2, depth: integer; top: boolean) – draws filled 3-dimensional rectangle
    * SetColor(color: integer) – sets foreground drawing color,

3.10 Unit tests

Our project contains a large set of unit tests. These tests are focused on testing individual concepts of the language (like if statement, pointers usage, goto, etc.) rather than testing wrong inputs (invalid sources, type mismatches, wrong arguments supplied to the interpreter, etc.). They are contained in the standard Maven location */src/test/java* and are further separated into multiple files depending on the concepts they are testing.

Code coverage of our tests is 83% of classes and 70% of lines. This includes also classes for our both entry points (and their helper classes) which are not tested. If we look only at our node classes then 93% of classes are tested and 73% lines covered. The missing lines are mostly uninteresting specializations (those that are practically the same as the others that are tested, for example addition of two longs, two integers and two doubles), wrong input handlers (for example wrong input in *read* method or break outside a cycle) or those that cannot be tested by unit tests (like graph unit, random and randomize functions, etc.). All

of these percentages were counted by IntelliJ IDE and the coverage of generated classes are not included in them.

3.11 Tic Tac Toe

As an example of interoperability of our Pascal interpreter with Java, we have decided to implement well known game Tic Tac Toe. The participants of the game are user at one side and an AI on the other. The main point of this implementation is that it is written in Java, but the AI uses script written in Pascal to decide what to do at its turn. The script is evaluated by our interpreter.

The whole game is implemented in *TicTacToeMain* class. It is implemented very simply, without any GUI, using only the standard input and output for interaction with the user.

The game instance receives a source containing AI's Pascal script. When the game starts, an instance of *PolyglotEngine* is created that parses the script. The game consequently starts with user's move by reading row and column from the standard input. It is followed by AI's turn. The polyglot engine evaluates the parsed script providing it the state of game in arguments. The script returns which cell is to be played. This cycle repeats until the game is over, meaning either one of the opponents wins or it is a tie. Our AI script serves only as an example and it always plays the first empty cell counting from top left corner. The script is inside our project with name *tttAI.pas*.

4. User documentation

This chapter serves as a user manual to our interpreter and the Tic Tac Toe game. Here describe how to start interpretation of some Pascal source and which options does the interpreter takes. We also describe what inputs and outputs does the mentioned game use and how to run it.

4.1 The interpreter

The whole interpreter is compiled into single executable JAR file. This means that it requires a running JVM on the target device. It can be executed as any standard JAR file, for example with the following command:

```
$ java -jar Trupple.jar
```

For simplification, there is also a shell script which executes this command so it can be simply executed with

```
$ ./trupple
```

4.1.1 Input Source file

The interpreter takes single argument which is a path to the source file that is to be interpreted. The following command executes our interpreter and starts interpreting a source file named hello.pas which is located in the same directory as the interpreter:

```
$ ./trupple hello.pas
```

The path does not need to be relative, the interpreter can also receive absolute path to the source file:

```
$ ./trupple /home/me/hello.pas
```

4.1.2 Turbo Pascal extension

As the interpreter is able to switch between standard Pascal and Turbo Pascal, it may be told which one to use. Option *std* chooses one of these two dialects. It takes one of the following values: *wirth* (for standard) or *turbo* (to use supported Turbo Pascal extensions). Specifying any other value will result in immediate stop of execution and displaying an informational message to the user. The following example interprets source file hello.pas using the ISO 7185 standard:

```
$ ./trupple -std=wirth hello.pas
```

The next command interprets the same source file using supported extensions from Turbo Pascal:

```
$ ./trupple -std=turbo hello.pas
```

If this option is not specified, the default behavior is to use the standard Pascal.

4.1.3 Units

To be able to use unit sources from main source file, the interpreter needs to know where to find them. This can be achieved by using the *I* option. The option receives a list of directories where the interpreter will look for the unit files. It will try to include each file from these directories with extension .pas. This option is *not* recursive meaning that it will not look recursively into subdirectories of the specified directories. The following example includes unit files from directories *mathunits* and *graphicunits* and interprets *hello.pas* source:

```
$ ./trupple -std=turbo -I mathunits,graphicunits hello.pas
```

Note that -std=turbo is required here because units are not supported in the ISO standard.

4.1.4 Extended goto support

The goto statements are not implicitly fully supported. The full support for these statements may be turned on with *-j* option.

4.1.5 IO

The interpreter expects its input in the standard input and its output is printed to the standard output.

4.2 Tic Tac Toe

Our Tic Tac Toe implementation is also deployed as an executable JAR. It requires one argument which is path to the Pascal AI script. It can be executed using the following command (requires running JVM):

```
$ java -jar TicTacToe.jar aiScript.pas
```

The user is always first on the move. When the game asks the user to specify which cell he wants to play, he has to write row and column numbers of the cell to the standard input separated by arbitrary whitespace. Both rows and columns are numbered from 1 to 3 where cell (1,1) is the top left one and (3,3) is the bottom right one. After the player plays its turn, the AI script chooses which cell to play. This cycle repeats until the game is over.

After each turn (either player's or AI's) the play area is drawn to the output and the game also checks whether the game is finished. The player's cells are drawn as ✓symbols, the AI's cells use *X* symbol and the empty ones are represented with □ sign. Example of an output of the game can be seen on Figure 4.1.

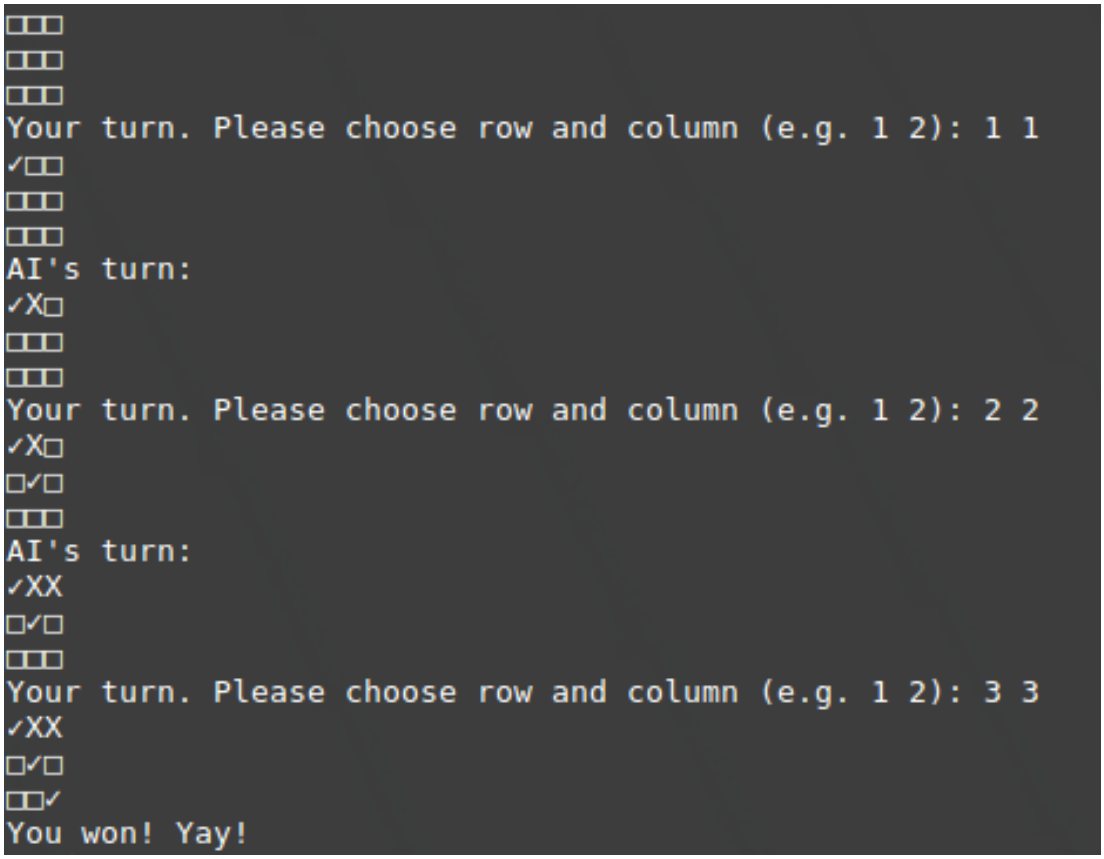

Figure 4.1: Tic Tac Toe game output

5. Evaluation

In this chapter we will look at the performance of our interpreter. We will compare it with compiled Pascal programs and also with Oracle's interpreter of *SimpleLanguage*, which is interesting in the way that it is also implemented in Truffle.

5.1 Limitations

Firstly, we will mention limitations that we have encountered during implementation of programs used for benchmarks. The problem with the Pascal compiler was that it uses only two-byte integers, which limited us mostly in the sizes of arrays and lead us to inability of using asymptotically faster algorithms than quadratic ones, because their run time was too short. Because *SimpleLanguage* is only a sample language, it does not support arrays and we had to use some very simple algorithm that can be written in this language for its benchmarks. Another issue with this language was that its interpreter caches return values from function calls (after 1000 calls), so we had to use time of the first execution which is not precise because of Java's class loading and the bytecode is not JIT-compiled, yet.

5.2 Configuration

In the following benchmarks, we have used *FreePascal* (version 3.0.0) compiler, our interpreter run atop JRE version 8 Standard Edition and also *GraalVM*, and the *SimpleLanguage* run only in *GraalVM*. They were executed on machine with 12 x *Intel Xeon X5650* processors at *2.67GHz*, *48GB* RAM and *Fedora 25* operating system.

5.3 Benchmarks

We have used multiple algorithms for our benchmarks. Each algorithm ran 1000 times *in a loop inside one program* for multiple input sizes. We have measured execution time of each cycle pass, therefore, the parsing phase time of the interpreters is *not* included in these measurements. We have used high precision functions provided by the used languages (Java's *System.nanoTime()* in the interpreters and *getTickCount* in Pascal). For each input size, we have computed average execution time from the last 20 cycle passes (excluding SimpleLanguage programs), so we let Java's JIT compiler some time to "warm-up" (load classes, JIT-compile bytecode, etc.).

The used algorithms were *Bubblesort* and *matrix multiplication* for comparison with compiled Pascal programs and *computation of Fibonacci's number* for comparison with the *SimpleLanguage* and compiled Pascal. The exact codes used inside the benchmark loops can be seen in Listings 5.1, 5.2, 5.3 and 5.4.

The results of these benchmarks are depicted in tables 5.1, 5.2 and 5.3. These tables follow the following conventions:

Listing 5.1: Code that was executed inside of the bubblesort benchmark loop

```
const size = 500;
type arrType = array[0..size] of integer;

procedure bubbleSort(arr: arrType);
var i,j, tmp: integer;
begin
    for i:=0 to size - 1 do begin
        for j:=0 to size - 1 do begin
            if arr[i+1] < arr[i] then begin
                tmp := arr[i];
                arr[i] := arr[i+1];
                arr[i+1] := tmp;
            end;
        end;
    end;
end;
```

Input size	1 000	2 000	4 000	8 000	10 000	12 000	14 000	16 000
Trupple/OpenJDK	100	400	1 588	7 375	11 242	14 649	19 914	30 560
Trupple/GraalVM	119	419	1 969	6 643	12 788	18 299	20 754	32 529
FreePascal	4	17	73	286	455	657	896	1 142

Table 5.1: Bubblesort benchmark results. The execution times are in milliseconds and averaged from last 20 runs of total 1000 runs.

- The first row always contains input sizes of the algorithm,
- The following rows contain name of the interpreter/compiler used for the benchmark in the first column, including platform atop which it was executed,
- The other cells contain execution times in *milliseconds* with respect to the input size and selected interpreter/compiler.

The first two tables show pretty expected values. Our interpreter can not compete in performance with compiled programs, but it also does not run too slow, being within the allowed range set up in our goals.

Table 5.3 shows us that our interpreter seems to be faster (we can not be a hundred percent sure because of limitations mentioned in Limitations) than Oracle's *SimpleLanguage* interpreter which is a very positive fact. The *SimpleLanguage* benchmarks for higher input sizes are off the scale because of its caching, which took part also in the first run.

Input size	100	200	300	400	500	600
Trupple/OpenJDK	201	1 297	4 825	10 710	20 207	37 582
Trupple/GraalVM	144	1 239	3 904	9 363	18 231	
FreePascal	5	39	131	346	634	1 137

Table 5.2: Cubic matrix multiplication benchmark results. The execution times are in milliseconds and averaged from last 20 runs of total 1000 runs.

Listing 5.2: Code that was executed inside of the matrix multiplication benchmark loop

```
const size = 100;
type arrType = array[0..size, 0..size] of integer;

function multiply(a, b: arrType): arrType;
var i,j,k,tmp: integer; res: arrType;
begin
    for i:=0 to size do begin
        for j:=0 to size do begin
            tmp := 0;
            for k:=0 to size do begin
                tmp := tmp + a[i][k] * b[k][j];
            end;
            res[i][j] := tmp;
        end;
    end;
    multiply := res;
end;
```

Listing 5.3: Code that was executed inside of the fibonacci's number benchmark loop (Pascal)

```
function fibonacci(i: integer): integer;
begin
    if i < 3 then fibonacci := 1
    else fibonacci := fibonacci(i-1) + fibonacci(i-2);
end;
```

Listing 5.4: Code that was executed inside of the fibonacci's number benchmark loop (SimpleLanguage)

```
function fibonacci(i) {
    if (i < 3) {
        return 1;
    }
    return fibonacci(i-1) + fibonacci(i-2);
}
```

Input size	24	25	26	27	28	29	30	31	32
Trupple/OpenJDK	55	89	146	258	388	627	1 030	1 636	2 611
Trupple/GraalVM	56	90	146	232	396	613	998	1 621	2 643
SimpleLanguage	259	405	645	883	1 417	1 295	1 399	1 304	1 341
FreePascal	0	1	1	2	4	7	12	19	41

Table 5.3: Fibonacci's number benchmark results. The execution times are in milliseconds. They are averaged from last 20 runs of total 1000 runs, except for the SimpleLanguage, which shows execution time of the first loop pass.

Conclusion

To conclude this thesis, we will sum up how much successful we were in implementation of our project.

We have implemented a working Pascal interpreter using Truffle framework that follows ISO 7185 standard (except some very rare occasions). All of the language's constructs that are reasonable on current hardware and JVM were implemented. We have also implemented several extensions of the language that were introduced by Borland's Turbo Pascal. We can say that the goal to implement a Pascal interpreter was fulfilled.

We also wanted our interpreter to run reasonably fast. It is obvious that we can not run as fast as compiled languages, but the benchmarks we made showed, that it runs in the limits that we have set up in our goals. On the other hand, they also showed that using the Graal JIT compiler did not improve our performance too much, which means that we could have utilized its advantage more than we did.

Our implementation of Pascal language in Truffle is also able to cooperate with other Truffle languages and Java. The cooperation with Java was demonstrated in our implementation of Tic Tac Toe game. Thanks to that, this goal of our thesis was also achieved.

Future work

Although we have fulfilled our goals, the implementation may still be extended. One of possible improvements to our interpreter may be to implement other extensions of Turbo Pascal that we have left out. The implementation is also extensible by missing subroutines in built in units and also more built in units. One of the core concepts that may be improved is the interoperability, which may be upgraded with support for more complex program arguments like arrays or records. The same applies to the possible return types of the evaluation of a Pascal source.

Despite the above mentioned improvements which may still improve our interpreter, we can conclude that it is ready to use.

Bibliography

[1] Pascal ISO 7185:1990. http://www.pascal-central.com/docs/iso7185.pdf. Accessed: 2017-12-07.

[2] Extended Pascal ISO 10206:1990. http://www.eah-jena.de/~kleine/history/languages/iso-iec-10206-1990-ExtendedPascal.pdf. Accessed: 2017-12-07.

[3] ArrayList (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. Accessed: 2017-05-27.

[4] The Compiler Generator Coco/R. http://www.ssw.uni-linz.ac.at/Coco/. Accessed: 2017-04-28.

[5] graal/PolyglotEngine.java. https://github.com/graalvm/graal/blob/master/truffle/src/com.oracle.truffle.api.vm/src/com/oracle/truffle/api/vm/PolyglotEngine.java. Accessed: 2017-06-29.

List of Figures

List of Abbreviations

OOP – Object oriented programming
GUI – Graphical User Interface
VM – Virtual Machine
JVM – Java Virtual Machine
AST – Abstract syntax tree
JIT – Just in time compiler
OS – Operating system
GC – Garbage collector
EBNF – Extended Backus-Naur form
IDE – Integrated development environment
AI – Artificial Intelligence
JAR – Java archive
JRE – Java Runtime Environment

Attachments

The electronic attachment to our thesis is divided into the following three parts

sources folder contains source code to our interpreter along with the implementation of Tic Tac Toe game. It also contains grammar and frame files required to generate our parser with the Coco/R tool. It does *not* contain generated sources which have to be generated by build process manually. An example Pascal sources can be found inside *sources/examples* folder. They contain sources used for benchmarks and an sample AI script to our Tic Tac Toe game.

documentation folder contains documentation to our project. It is a classic *JavaDocs* documentation consisting of HTML and CSS files extracted from the documentation comments in our sources.

bin folder contains relevant executable JARs. The *trupple.jar* is our interpreter. It comes with shell script called *trupple* which simplifies execution of the interpreter. There is also *TicTacToe.jar*, which starts our Tic Tac Toe game.