I.J. Information Engineering and Electronic Business, **2022, 5, 32-41**
Published Online on October 8, 2022 by MECS Press (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2022.05.04

Analysis of the New Generation source-to-source Compilers Using the Google Web Toolkit

Kire Jakimoski, Blagoja Chavkovski
Faculty of Informatics, AUE-FON University, Skopje, Republic of North Macedonia
Email: {kire.jakimoski, blagoja.cavkoski}@fon.edu.mk

Received: 27 March 2022; Accepted: 11 July 2022; Published: 08 October 2022

Abstract: The role of source-to-source compilers nowadays increases faster since each high-level language is in a need to gain more recognition in each field of development. One of the fields that became very popular in the last ten years is web development, where the browsers became the “machines” of our everyday life. They are relevant and powerful, and the only drawback they have is that they only understand one language, that is JavaScript. The need of other languages to be included in the client side of the web development, created the steam for source-to-source compiling, sometimes referred as Transpiling, where one high level language as Java, C, C# and many others are translated mostly, but not solely, to JavaScript. A famous tool that is well recognized is Google Web Toolkit (GWT) which translates Java source code to JavaScript source code. The core of this tool is the compiler which is covered in great details in this paper. Main goal and benefit of this paper is to analyze and compare the difference of the process of translation to JavaScript by transpiler to the process of “normal” compiling and to highlight key aspects of this process.

Index Terms: Google Web Toolkit, Interpreter, Source-to-source compiler, Transpiler.

1. Introduction

Computer hardware and languages have tremendous development in the past few decades. Nowadays there are a lot of computer programs for the novel technology equipment which are maintained and rewritten. Expenses are high when we want to rewrite the whole code from a scratch. Web programming is widely spread using scripting computer languages like JavaScript, TypeScript and partially Python. As a result of this many problems have arisen regarding the reliability and quality of the software packages. That is why source-to-source compilers known as transcompilers or transpilers has been developed in order to solve challenges in this field [1].

Authors in [2] present the design on a modern compiler that is used in the last decade. They put focus on the principles and techniques of wide application during the process of compiler creation. Interesting fact her is that it covers the time period of 12 years between the first and second edition. It is important to note here that standard compiling techniques and paradigms have stood the test of the time and have kept their importance during the time. Analyzing the optimization techniques, it is noticed that some new optimization techniques have been invented, existing ones have been improved, and some old ones have gained even greater prominence.
In [3], the industry and teaching famous “Dragon book”, we find a complete detailed structure of one modern compiler. Authors, also give us the pleasure to see the interconnection on compilers with machine architecture, language theory, algorithm, and software engineering. It is one of those books that anyone heading for the secrets of language creation must read. In [4] we can see the global overview of what compilers are, and it also covers in a very nice, shaped way Context-Sensitive analysis well known as Type System, all of its components, and the process of inference.

Performed research in [5-13] cover to some extend the Google Web Toolkit (GWT). Some of them are more focusing on the compilation, transpilation of the code, and some of them are focusing on the process of structuring the code for having better output and using the toolkit for creation of Internet Rich Application. GWT or the core of it, the compiler, is one real example on how one Transpiler can look like. In [6-9] there is a dipper dive in the compiler itself, so here we can find all the magic of parsing, scanning AST tree creation (both Java and after a JS). We can see how the famous Visitor pattern is used and how context is moved around. In [9-13] we can find explanation on how the tool can be used, how all components not just the compiler come in together to form the Toolkit. We can find also, examples on structuring Rich Internet Applications on a highly reusable way, and all the benefits that this brings on very large single page applications.

Section 2 of this paper covers the development phases of the programming language with more aspects on what compilers are, what are the main structural parts of one compiler, and how these parts come together in the final picture. Furthermore, in Section 3 in-depth analysis is presented on the Google Web Toolkit compiler, particularly in alteration of the structural parts when one compiler could become a "transpiler". In addition, it covers requests of the specific compilers, and analyzes the development for being key part of each major language. Finally, Section 4 concludes this work.

2. Compilers and Transpilers

The development of programming languages is as old as the first idea from the engineers that humans and computers can communicate, with one purpose, to help in the everyday tasks that people must finish. From early days was recognized that this process of communication is not that easy, and the need for higher level programming language became so evident. Computers nowadays are million times faster and more capable, with memories so fast and bigger in sizes, but the way how we communicate whit them is not much altered.

All higher-level languages have the same path of development, and they all cover same or similar parts, even from the very first COBOL compiler and up to the newest ones as the source-to-source compilers, e.g transpilers. Their way of functioning is almost the same.

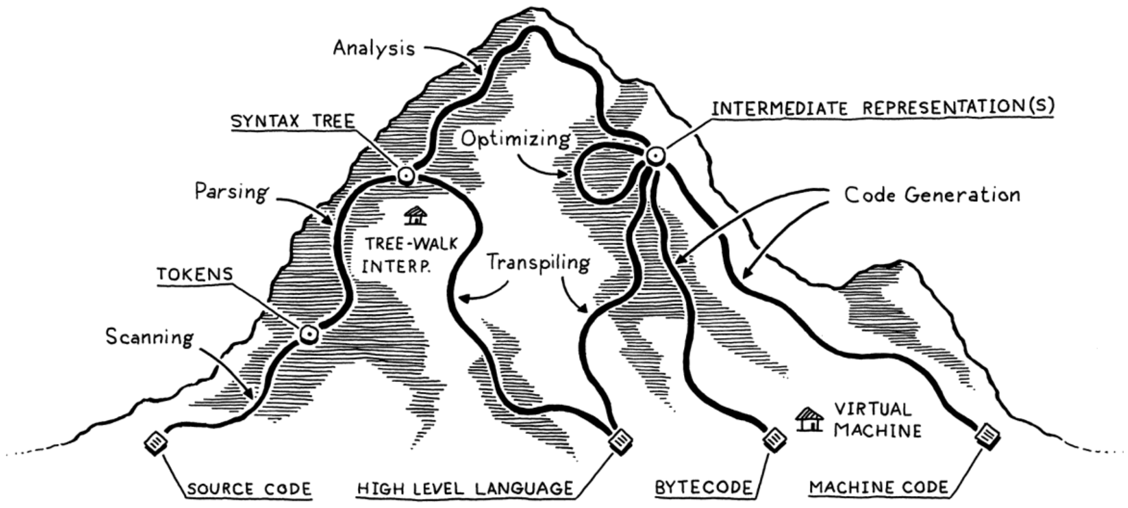

Fig. 1. Development phases of a programming language [14]

Phases of development of one programming language is best given in The Fig. 1 [14]. The process starts at the very bottom of the so-called mountain, where for input we have the source code of the language, then in every of the next steps the input is transformed in a higher form which is less human readable, but more and more machine understandable. After several phases in Fig. 1, we rich the top of the mountain, where we have overseen of what the inputted source code means and what exactly wants to be achieved. After all these ascending phases comes the part of descending, where the input is transformed to a form which can be understood by the processing unit where it needs to be executed. Authors in [15] put stress on the integrated development environment that consists of program building tools integrated in one environment. In the following subsections a short description will be given of the key phases in Fig.1.

A. Scanning

The very first phase is the scanning, or also called lexical analyses of the inputted source language. Scanner in this phase takes the linear input stream of characters and divides it to smaller parts, some could say in words manners, which in the programming languages are known as tokens. It worth noting that some of this tokens can be a single character e.g" (, * ", a combination of characters or can be bulked characters as are for example empty spaces. The result most usually looks like this in Fig. 2.

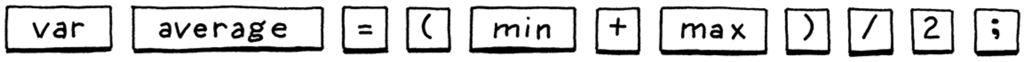

Fig 2. Scanner tokens.

B. Parsing

The next phase on the "climbing mountain" in Fig. 1 is the parsing. It works on the received tokens from the previous phase, and it has a purpose to give to these words from the scanning a "grammatical meaning", in a way to make it possible longer expressions to be created from these tokens. So, this phase is one level higher in this process. If scanning phase was treating the characters as words, what we call tokens, now the parser in this phase treats the tokens as a basic input from which a longer expression is created.

The rule of grammar in formal or context free grammar always consists of a head and a body. The head is just the name, and the body describes what it will generate. In its pure form the body is just a list of symbols and those symbols come in two forms, terminal and nonterminal ones.

After postulating the expression grammar for the language, next step is that grammar to be represented. It is well known that the grammar is always recursive so the best way to represent it is by forming Syntax Trees. They are then traversed in many ways to achieve the desired output. One of the most important parts of the grammar and creation of the Syntax Trees is the accounting of well-known mathematical theory of Precedence and Associativity. First is related to the operand's evaluation, for example "*, /" are always evaluated before "+, -", and the last one is related to left or right associatively, for example "-" is "left associative" so the expression:

$$6 - 2 - 1 \tag{1}$$

is equivalent to:

$$(6 - 2) - 1 \tag{2}$$

Assignment on the other hand is "right associative":

$$x = y = z \tag{3}$$

is equivalent to:

$$x = (y = z) \tag{4}$$

Well defined grammar is the key of the parsing phase. There are a lot of parsing techniques which names are combination of L and R, e.g. LL(k), LR(1), LALR together with more advanced ones, e.g. Earley parsers, the shunting algorithm, packet parsing etc. In this paper evaluation is based more closer to the recursive descent technique.

Recursive descent parser is what is known as a top-down parser because it starts from the most complex outer grammar rule (usually the expression), and recursively iterates over the nested expressions ending at the leaves of the syntax tree. For comparison, the LR type of parsers starts at the very bottom, with the leaves, and builds more complex expressions as it iterates to the top. The recursive descent parser is a direct translation of the grammar to an imperative code where each rule of the grammar becomes a function in the code. The process of descent is called "recursive" as this is how it is translated to the implementation, whenever a rule calls or refers to itself - the translation is a recursive function call.

What we have explained is the main job of any parser but coupled to this part comes the second, but also important job, detection of invalid token sequence(s) and reporting to caller as close as possible how to fix the encounter errors in the code.

Usually people underestimate the second part, but for a good language design this is as important as the main job. Nowadays programmers expect that parsing happens as users type something, when error catching and reporting, with the part of suggested fixes, happens almost instantly.

C. Analysis

After the parsing and creating the syntax tree, there are 2 ways how the journey could continue. It could be decided to be continued with one more climb on the mountain or to start walking down by interpreting. Both phases are possible by passing the Syntax tree, with a well-known so-called Visitor pattern. The aim here is not to go into too many details on the pattern. What this pattern does is just applying desired behavior in any of the tree nodes. Interpreter is nothing more, but a specific implement of the pattern interface which runs the program logic by following the grammar. This way of direct run of the code is not the fastest one, and languages which are interpreted, e.g., JavaScript, Python etc., tend to be slower than languages which after parsing and creation of the syntax tree continues one step forward, whereby visiting the nodes apply statical analysis, then do optimizations and code generation. This kind of languages are e.g. C, Java C# etc. Worth mentioning here is that all these phases and transformation of the syntax tree are done by different visitor passes that apply desired logic on the tree nodes.

The rest of the phases are out of the scope of our paper. As can be seen from the mountain representation, the process of translation to other high(er) level language happens on the same place where the interpretation happens, with the difference that during the visit of the syntax tree instead of running the program, visitor does a tree transformation in a desired state for an output closer to the desired translated language. Exactly with this pass, or multiple passes, which are done with the help of the visitor pattern, and exactly at this place the compiler is created which nowadays is well known as transpiler.

3. Analysis of the Google Web Toolkit Transpiler

To create a compiler which is known as transpiler, a scanning on the input code is first thing that should be done, which will produce the tokens. Using the tokens we can start the next phase, creating the syntax tree. Gwt syntax tree is specifically created to satisfy further requests for translation to higher language, in this case JavaScript. This is how the base tree node looks like:

```
public abstract class JNode implements JVisitable, HasSourceInfo, Serializable {

  private SourceInfoinfo;

  protected JNode(SourceInfo info) {
    assert info != null : "SourceInfo must be provided for JNodes";
    this.info = info;
  }

  public SourceInfogetSourceInfo() {
    return info;
  }

  public void setSourceInfo(SourceInfo info) {
    assert this.info.getOrigin() == info.getOrigin();
    this.info = info;
  }

  // Causes source generation to delegate to the one visitor
  public final String toSource() {
DefaultTextOutput out = new DefaultTextOutput(false);
SourceGenerationVisitor v = new SourceGenerationVisitor(out);
v.accept(this);
    return out.toString();
  }

// Causes source generation to delegate to the one visitor
  @Override
  public final String toString() {
DefaultTextOutput out = new DefaultTextOutput(false);
ToStringGenerationVisitor v = new ToStringGenerationVisitor(out);
v.accept(this);
    return out.toString();
  }
}
```

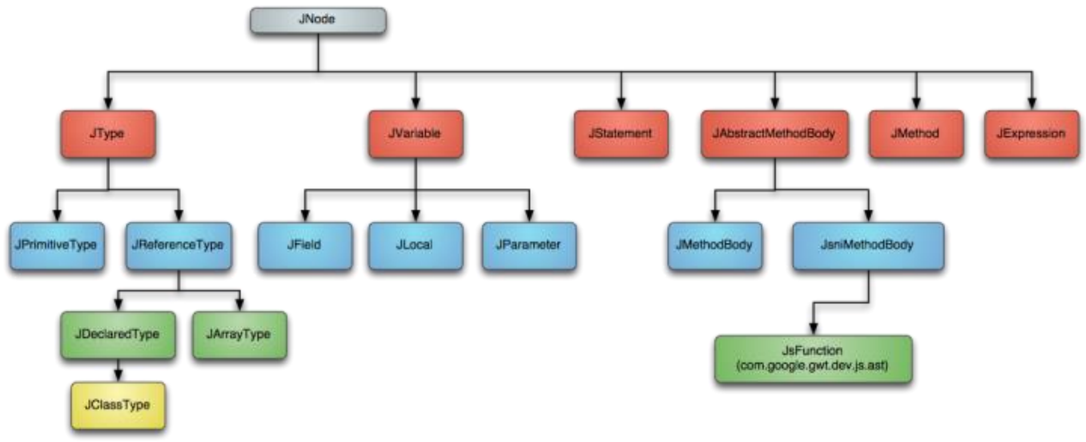

Fig. 3 Java AST tree.

As it is viewable from the class, it has the well-known implementation for the visitor pattern e.g., JVisitable and plus the HasSourceInfo, which gives a clue if the node has code that needs to be translated or its just carrying additional helpful information, and of course the well-known java Serializable interface. All Java types has their own implementation, which is subclass of the JNode, as can be seen in the Fig. 3.

The higher-level node is JProgram which is representing the full java program, e.g. the full entry code. It is taken into consideration now one simple representation on given entry code:

```
class Foo {
  int x;
  void bar() {
    x = 2 + 5;
  }
}
```

The related syntax tree is presented in Fig. 4.

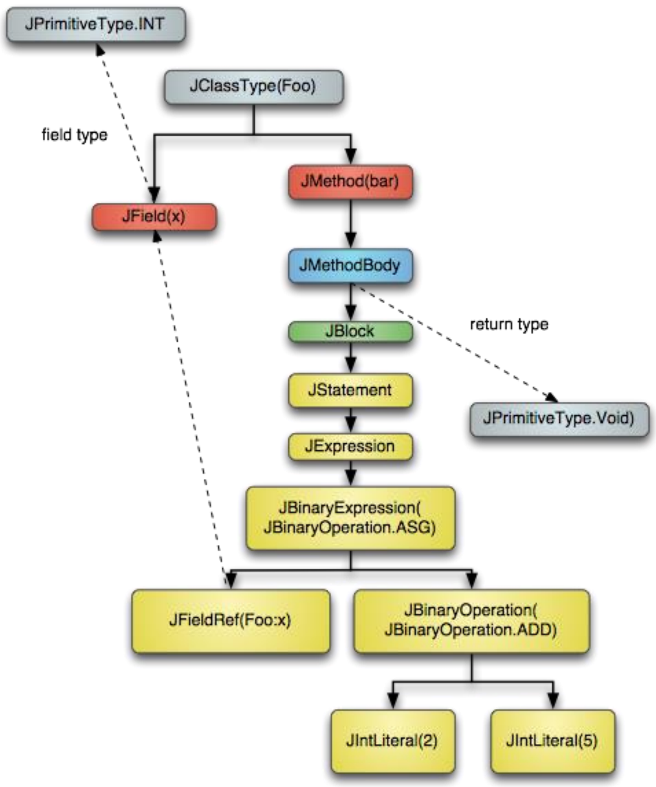

Fig. 4 Java AST tree for class Foo

What can be noticed is that as in any other representation of syntax trees, here also at the bottom are the primitive types represented with JIntLiteral, which are combined with the JBinaryOperaion for Add, which itself is represented as a JExpression, or to be more precise as a JBinaryExpression.

After parsing all of the entry code will be represented as java syntax tree, the process of translation continues with converting the Java AST tree to JavaScript AST tree. As a logic consequence, this process happens with the help of the visitor pattern. GWT has two base implementations for visiting the tree(s). One of them is:

```
/**
 * A visitor for iterating through an AST.
 */
@SuppressWarnings("unused")
public class JVisitor {

  protected static final Context LVALUE_CONTEXT = new Context() {
```

```
  public booleancanInsert() {
    return false;
  }

  public booleancanRemove() {
    return false;
  }

  public void insertAfter(JNode node) {
    throw new UnsupportedOperationException();
  }

  public void insertBefore(JNode node) {
    throw new UnsupportedOperationException();
  }

  public booleanisLvalue() {
    return true;
  }

  public void removeMe() {
    throw new UnsupportedOperationException();
  }

  public void replaceMe(JNode node) {
    throw new UnsupportedOperationException();
  }
};

protected static final Context UNMODIFIABLE_CONTEXT = new Context() {

  public booleancanInsert() {
    return false;
  }

  public booleancanRemove() {
    return false;
  }

  public void insertAfter(JNode node) {
    throw new UnsupportedOperationException();
  }

  public void insertBefore(JNode node) {
    throw new UnsupportedOperationException();
  }

  public booleanisLvalue() {
    return false;
  }

  public void removeMe() {
    throw new UnsupportedOperationException();
  }

  public void replaceMe(JNode node) {
    throw new UnsupportedOperationException();
  }

};
```

```
  protected static InternalCompilerExceptiontranslateException(JNode node, Throwable e) {
    if (e instanceofVirtualMachineError) {
      // Always rethrow VM errors (an attempt to wrap may fail).
      throw (VirtualMachineError) e;
    }
InternalCompilerExceptionice;
    if (e instanceofInternalCompilerException) {
      ice = (InternalCompilerException) e;
    } else {
      ice = new InternalCompilerException("Unexpected error during visit.", e);
    }
ice.addNode(node);
    return ice;
  }

  public final JExpressionaccept(JExpression node) {
    return (JExpression) accept((JNode) node);
  }

  public JNodeaccept(JNode node) {
    return accept(node, false);
  }

  public JNodeaccept(JNode node, booleanallowRemove) {
    try {
node.traverse(this, UNMODIFIABLE_CONTEXT);
      return node;
    } catch (Throwable e) {
      throw translateException(node, e);
    }
  }

  public final JStatementaccept(JStatement node) {
    return accept(node, false);
  }

  public final JStatementaccept(JStatement node, booleanallowRemove) {
    return (JStatement) accept((JNode) node, allowRemove);
  }

  public <T extends JNode> void accept(List<T> list) {
    int i = 0;
    try {
      for (int c = list.size(); i< c; ++i) {
list.get(i).traverse(this, UNMODIFIABLE_CONTEXT);
      }
    } catch (Throwable e) {
      throw translateException(list.get(i), e);
    }
  }

  public <T extends JNode> List<T>acceptImmutable(List<T> list) {
    accept(list);
    return list;
  }

  public JExpressionacceptLvalue(JExpression expr) {
    try {
expr.traverse(this, LVALUE_CONTEXT);
      return expr;
    } catch (Throwable e) {
      throw translateException(expr, e);
    }
  }

  public <T extends JNode> void acceptWithInsertRemove(List<T> list) {
    accept(list);
  }

  public <T extends JNode> List<T>acceptWithInsertRemoveImmutable(List<T> list) {
    accept(list);
    return list;
  }

  public booleandidChange() {
    throw new UnsupportedOperationException();
                                                                      }
```

Too many details on the code in this base visitor implementation are not needed, but what is worth nothing is that the visitor here has a context which is hand overed to every node visited. The idea with this is to have an easy way to transfer to, and from each node, some additional information.

All traversals are done with the help of two base visitor implementation:

- JVisitor, used by subclassing to collect some info for next visits or to do some user-friendly printing.
- JModVisitor, used by subclassing to modify the tree.

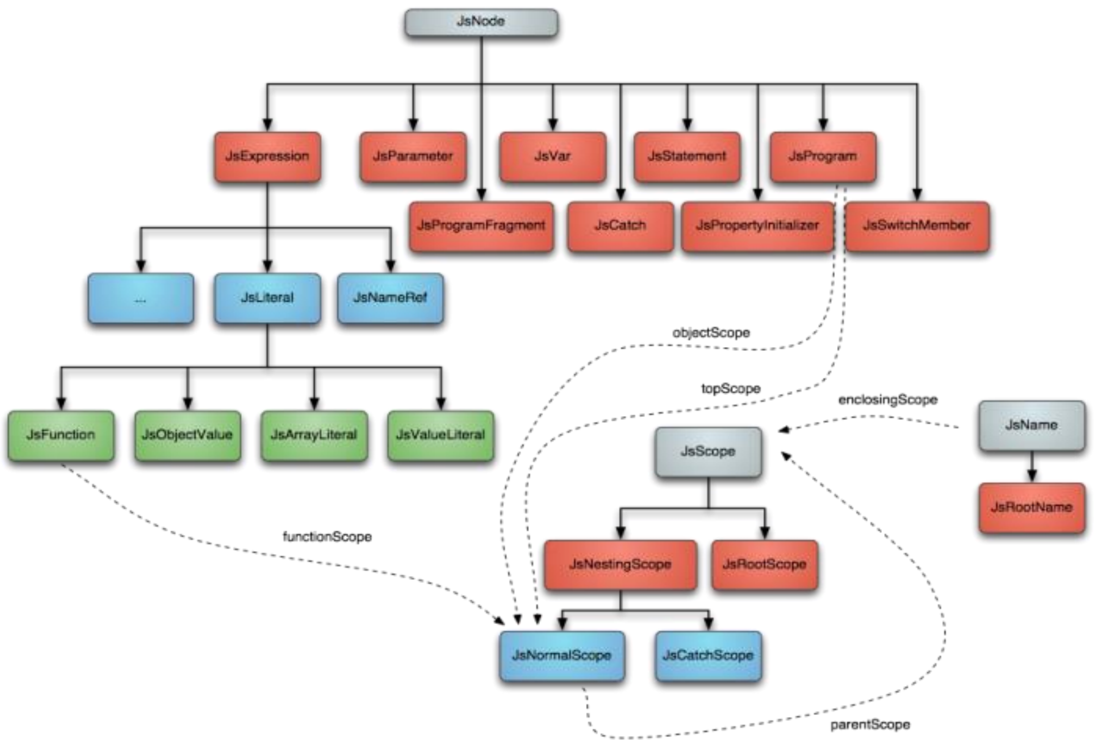

Fig. 5. JavaScript AST Tree

After the process of transformation to JavaScript AST tree is done, result is the next tree presented in Fig. 5. This process continues with specific phases which are done by visiting and revisiting the tree with the help of one of the base visitor classes, and have a purpose to optimize the code. Here are some of the phases:

- **CreateNamesAndScopesVisitor** - it does a variable resolution.
- **JsSymbolResolver** - it does symbolic resolution.
- **JsStaticEval** - constant evaluation.
- **JsNormalizer** - normalization of the JS AST tree for example (a, b)++ it is normalized to (a, b++).
- **JsInliner** - inlining some parts of the code[1].
- **JsUnusedFunctionRemover** - remove of all that is not used, not referenced functions and variables.

[1] en.wikipedia.org/wiki/Inline_expansion

- **JSDuplicatedFunctionRemover** - removal of duplicated functions, without function overloading as Java has, so all JavaScript with same name must be renamed or deleted. (Step is not mandatory but it is desirable as it will produce code with a smaller size).
- **JsNamer** - renaming all methods in a obfuscated way[2].
- **JsStackEmulator** - reorganization of the calls to have better performance while executed.

The last and most important step is to create the JavaScript code out of the JS AST tree. This phase is nothing more than just visiting each node and printing it out. The process of printing is done with a special control point which is later needed for easy point to the appropriate js code file. This mechanism is called linking. The idea behind is to have smaller files grouped in folders and loaded by the respected browsers as needed. This is how they look like:

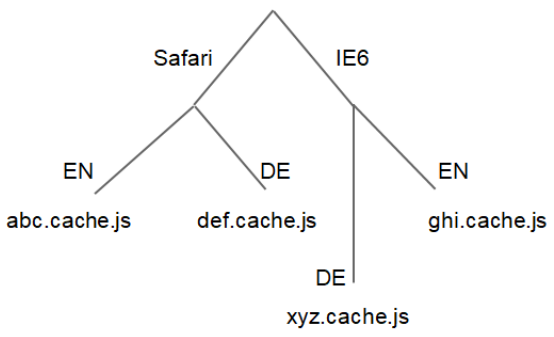

Fig. 6. Linking of JS output files

As it can be noticed in the output folders, specialized "js" files are present which are targeting specific browser for a specific language. This helps in optimization of the code file download and contains a specific optimization for a specific browser.

4. Conclusion

In this paper a real demonstration of a transpiler is presented, as well as in-depth analysis of the way GWT, the tool created by Google, translates Java to JavaScript. Furthermore, analysis is also done on all the steps needed to be performed, in order Java code to be outputted as a JavaScript. It starts with the intro on the process of creating a Java AST tree based on the Java input, then the creating of the revisited JavaScript AST tree, which is created by specific passes on the Java AST tree, and then the creating of the JavaScript output.

In this context logical question of this type is arising: Why are this kind of transpilers/tools needed? It can be concluded that this kind of transpilers/tools are very important, because Web browsers are the "machines" of our modern lives in the last 10 years, and the only machine code they understand is JavaScript. All other major languages nowadays have some tool included that translates their syntax to JavaScript, so they can be part of the internet development for the web. We can also note that new languages are now created, and immediately have bundled a tool that translates their code to JavaScript. This process of translation is done by a specific compiler known as transpiler, which is main subject in this paper, and it is in the focus of the analysis in this paper. Furthermore, another goal in this paper is to explain the difference between the present process compared to the process of "normal" compiling, highlighting the key aspects.

References

[1] Bastidas, F. A., & Pérez, M. (2018, October). A systematic review on transpiler usage for transaction-oriented applications. In *2018 IEEE Third Ecuador Technical Chapters Meeting (ETCM)* (pp. 1-6). IEEE.
[2] David, Galles (2014). *Modern Compiler Design 2nd Edition.*
[3] Alfred Aho, Monica Lam, Ravi Sethi (2016). *GCompilers: Principles, Techniques, and Tools 2nd Edition*.
[4] Keith, Cooper, Laura Torczon (2014). *Engineering a compiler 2nd Edition*.
[5] Chaganti, Prabhakar (2007). *GWT Java AJAX Programming: A Practical Guide to Google Web Toolkit for Creating AJAX Applications with Java.* Packt Publishing Ltd.
[6] Smeets, B., Boness, U., & Bankras, R. (2008). *Beginning Google Web Toolkit*. From Novice to Professional. Apress.
[7] Johnson, Bruce, and Joel Webber (2007). *Google web toolkit.* Addison-Wesley.
[8] Murugesan, San (2007). *Understanding Web 2.0*. IT professional 9, no. 4: 34-41.
[9] Prabhakar, Chaganti (2007). *Google Web Toolkit: GWT Java Ajax Programming*.
[10] Geary, David (2007). *Google™ web toolkit solutions: cool & useful stuff.* Prentice Hall Press.
[11] Lawton, George (2008). *New ways to build rich internet applications.* Computer 41, no. 8: 10-12.

[2] en.wikipedia.org/wiki/Obfuscation_(software)

[12] Meliá, S., Gómez, J., Pérez, S., & Díaz, O. (2008, July). A model-driven development for GWT-based rich internet applications with OOH4RIA. In *2008 Eighth international conference on Web engineering* (pp. 13-23). IEEE.
[13] McFall, R., & Cusack, C. (2009). Developing interactive web applications with the google web toolkit. *Journal of Computing Sciences in Colleges*, *25*(1), 30-31.
[14] Nystrom, R. (2021). *Crafting interpreters*. Genever Benning.
[15] Owoseni, A. T., & Akanji, S. A. (2016). Survey on Adverse Effect of Sophisticated Integrated Development Environments on Beginning Programmers' Skillfulness. *International Journal of Modern Education and Computer Science*, *8*(9), 28.

Authors' Profiles

Kire Jakimoski received his D.Sc. in Technical Sciences from the Ss. Cyril and Methodius University in Skopje, R. Macedonia in 2013. From 2002 to 2006 he works as an Officer for Telecommunications in the Ministry of Defense in the Republic of Macedonia. From January 2006 to February 2012, he works as an adviser for information security in the INFOSEC Division in the Directorate for Security of Classified Information in Skopje, Republic of Macedonia. From March 2012 he is with the Faculty of Informatics, FON University in Skopje. His research interests include Information and Communication Technologies, Wireless and Mobile Networks, Heterogeneous Networks, Computer Networks, Cyber Security, Network Security.

Blagoja Chavkoski obtained his M.Sc. in Computer Science and Technology from the Faculty of Informatics of AUE-FON University, Skopje, Macedonia, and his B.Sc. in Electrical Engineering and Computer science from the Ss. Cyril and Methodius University of Skopje, Macedonia. His 15 years of experience in computer engineering had lend him in diversity of projects. One of them in early 2010s was transition of internally used tools at Bosch to web based Rich Internet Applications. This was mainly done with the help of GWT, which later inspired him to give a dipper dive on the secrets that compilers and transpilers hide. His research interests include Computer Science, Language Creation, Compilers, Cloud Computing.

How to cite this paper: Kire Jakimoski, Blagoja Chavkovski, " Analysis of the New Generation source-to-source Compilers Using the Google Web Toolkit", International Journal of Information Engineering and Electronic Business(IJIEEB), Vol.14, No.5, pp. 32-41, 2022. DOI:10.5815/ijieeb.2022.05.04