Investigating A Linear Approach to Arithmetic Re-Association within GraalVM

by
Nathan Corcoran

School of Electrical Engineering and Computer Science,
The University of Queensland.

ii

Nathan Corcoran
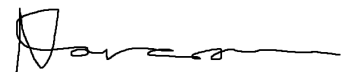n.corcoran@uq.edu.au
November 4, 2024

Prof Michael Bruenig
Head of School
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, QLD 4072

Dear Professor Bruenig,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Software Engineering, I present the following thesis entitled "Investigating A Linear Approach to Arithmetic Re-Association within GraalVM". This work was performed under the supervision of A/Prof. Mark Utting, Prof. Ian J. Hayes and, Mr. Brae Webb.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Nathan Corcoran.

iv

Acknowledgments

I would like to acknowledge my supervisors A/Prof. Mark Utting, Prof. Ian J. Hayes, and Mr. Brae Webb for their continual support and guidance throughout the entire project. Thank you all for enabling me to learn a great deal through this experience.

Abstract

Optimising compilers are essential tools for building efficient modern programs. The GraalVM compiler, released by Oracle, aims to achieve high levels of optimisation. Arithmetic expressions provide many opportunities for optimisation, with arithmetic re-association playing a crucial role in identifying chances for further optimisations. Currently, when re-associating complex expressions, the GraalVM compiler's method of graph manipulations are insufficient. By reducing arithmetic expressions to a list of operands, they can be ranked common factors and through a simple sort like-terms can be grouped. From this, algebraic simplifications are as trivial as removing elements within the list. This approach simplifies the re-association process and allows the GraalVM compiler to apply further optimisations that were not previously identifiable.

Contents

List of Figures

List of Code Snippets

Chapter 1

Introduction

Optimising compilers are a crucial tool for building efficient modern programs. They allow developers to write highly abstracted code without sacrificing program performance. High-level abstractions are relied upon when programming complex models that many of modern software requires. The GraalVM compiler, released by Oracle, is one such compiler that aims to achieve high-levels of optimisation. Arithmetic expressions allow for many optimisation opportunities. Arithmetic re-association plays a pivotal role in this. The method is not used to optimise the code by itself, but rather to create opportunities for further optimisations. Consequently, to maximise the effectiveness of an optimising compiler, we must ensure effectiveness of re-association methods. That is, expressions need to be represented in such a way that eases the implementation of optimisations. Currently, the GraalVM compiler optimises expressions by way of graph comparisons and manipulations invoked via the `ReassociationPhase`. This phase, for example, would take an expression such as `x + y + z - z - y` and re-order its sub-expressions such that like-terms are grouped, possibly forming `x - x + y - y + z`. The compiler can then non-trivially apply, in this instance, algebraic simplifications to produce just `z`. For lengthy or complex expressions GraalVM's graph manipulations can become computationally expensive and the `ReassociationPhase` is only utilised on expressions within loop bodies [14]. By replacing the current method with our list-based approach, arithmetic re-association is trivial.

In this report, a method of arithmetic re-association is proposed that re-orders expressions by manipulating a list of its operands. It will be shown that by ranking and sorting a list of an expression's operands, further optimisations can be identified and applied by the GraalVM compiler.

Chapter 2 covers background information regarding arithmetic re-association; some common algebraic simplifications; the role of intermediate representation within compiler design, detailing how arithmetic expressions can be represented; and finally the GraalVM compiler itself.

Chapter 3 investigates known methods of arithmetic re-association and a method of testing optimising compilers.

Chapter 4 details the proposed list-based approach, covering the steps of graph preparation, expression tree flattening, operand ranking, a possible method of optimising certain expression trees, and expression tree reconstruction. Testing performed and problems that arose during the implementation process are also included.

Chapter 5 shows and discusses that the proposed approach successfully re-associates and simplifies complex arithmetic expressions.

Chapter 6 summarises this report and presents possible further work for future researchers.

Chapter 2

Background

2.1 Arithmetic Re-Association

Arithmetic re-association utilises the associative and commutative properties of certain arithmetic operators to re-arrange expressions [13]. This may allow simplifications that were not easily identified in the original expression to be made evident. Re-association methods use associative, distributive, and commutative properties of some operators to re-order arithmetic expressions. Using re-association, a compiler would transform the expression `x = 23 + i + y + 43 + i` into the form `x = i + i + y + 23 + 43`. It is now easier for the compiler to reason that the sub-expression `23 + 43` can be optimised to a single constant. Arithmetic re-association only alters the representation of a given expression, and not its resulting value. Hence, it can only be applied to operators that are associative. Due to hardware limitations, the additive operator is not associative across floating-point values, therefore, re-association methods are generally not applied [12].

2.2 Optimisations

2.2.1 Algebraic Simplification

Algebraic simplification can allow for large performance optimisations in programs. It is the process of simplifying expressions which can produce smaller but equivalent expressions, or replace operators with which are faster to compute. This process utilises mathematical properties such as associativity, commutativity, distributivity, and operator identities. Some simple examples of algebraic simplification can be seen in Fig. 2.1.

```
-(-a) --> a
a - a --> 0
a * 2 --> a << 1
2 * a + 3 * a --> (2 + 3) * a
```

Figure 2.1: Examples of algebraic simplifications

```
for i = 1..10 {
    a = i + b * 3
}
```

```
b0 = b * 3
for i = 1..10 {
    a = i + b0
}
```

Figure 2.2: Example of before and after hoisting respectively

2.2.2   Constant Folding

Constant folding is the process of trivially merging sequences of sub-expressions containing only constant values. Simply, the expression `2 + 3 - 1` would be transformed to be `4`.

2.2.3   Loop Invariant Code Motion

Loop invariant code motion, also referred to as *hoisting*, aims to reduce the number of computations required within loop bodies. If a sub- expression is reasoned to be constant, that is invariant, across all loop iterations, this value can be stored elsewhere and its resulting value can be referenced inside of the loop. A trivial example of this can be seen in Fig. 2.2. More complex scenarios arise when function calls are hoisted, yielding larger performance optimisations.

2.3   Intermediate Representation

Generally, compilers do not perform optimisations directly on high-level code. Instead, an *intermediate representation* (IR) is used to represent the program in such a way that optimisations are easily implemented. Intermediate representations are split into three levels: high, medium, and low. Each level generally represents the source code in different ways, these differences are defined by the optimisations expecting to be performed. Common forms of IR are graph-, tree-, and stack-based structures. These are usually selected depending on how much information is needed from the source code in order to apply an optimisation. This report will focus mainly on graph-based IR, however both tree- and graph-based representations of arithmetic expressions will be discussed further in Section 2.3.1.

2.3.1 Arithmetic Expressions

Arithmetic Expression Trees (AETs) leverage the hierarchical nature of binary trees to represent mathematical expressions [16]. The internal nodes of the tree correspond to *operators* whilst the leaf nodes represent *operands*. These structures can be specialised by the operator stored at their root, where only internal nodes of the same type as the root are also considered operators and the remaining nodes are handled as operands. This allows the AET to exhibit properties of its root operator such as associativity, commutativity, and distributivity. An appropriate expression tree, such as an addition-expression tree or a multiplication-expresssion tree, can be re-associated without affecting the resulting value of the expression. Fig. 2.3 shows an example of an addition-expression tree, where the operator nodes are shown in red and the operand nodes in blue.

Whilst traditional AETs following the strict structure of binary trees, arithmetic expressions can also be represented using a graph-based structure, as they are in the GraalVM compiler. Although the slight difference in structure, both representations are interfaced in the same way as expression traversal is not affected.
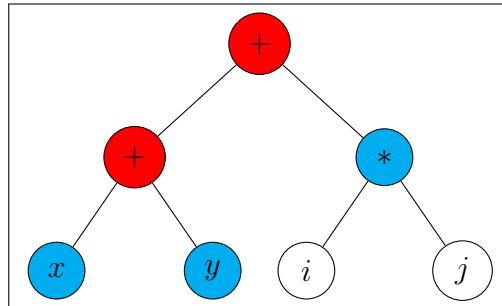
Figure 2.3: Example Addition-Expression Tree

2.4 GraalVM

The GraalVM compiler, released by Oracle Labs, is a polyglot optimising compiler for languages that run on the Java Virtual Machine (JVM). It focuses on aggressively optimising programs, with around 62 possible methods [7] of doing so.

The GraalVM compiler uses a graph-based structure as an intermediate representation. Graphs are an efficient way to store program flow contexts. Nodes represent basic code blocks or data values, and directed edges represent dependencies between nodes. Lacking circular dependencies [3], the graph is more accurately a directed acyclic graph (DAG). DAGs minimise redundancy by sharing common sub-graphs. The GraalVM IR also uses a single graph to model both control- and data-flow,

creating a *program-graph* [11]. Traversing control edges outlines the order in which a program must be run, whilst traversing data edges outlines how data is passed and what instructions are manipulating it throughout the program's run. This benefits the implementation of optimisations as both contexts are easily accessed. Though, this makes graphs with nested or complex control flows messy and difficult to follow. Hence, it is often stated that GraalVM uses a *sea of nodes* to represent program behaviour.

Within GraalVM optimisations are referred to as *phases*. Execution order of phases are defined as `PhaseSuite`s, an ordered list of optimisations to be applied in sequence. `PhaseSuite`s can be scheduled within other `PhaseSuite`s. The compiler uses three tiers to run phases; low, mid, and high. Progressing from the high to the low tier, lower and lower level optimisations are applied to the graph. After each tier has completed, the graph is prepared for the next, called *lowering* [9]

Chapter 3

Literature Review

3.1 Effective Partial Redundancy Elimination

This report addresses methods to increase the effectiveness of partial redundancy elimination within an optimising compiler. Cooper and Briggs [8] propose a method of *global re-association* to re-arrange expressions within a program.

The report uses a system to sort sub-expressions by ranking the operators within them. It uses these ranks as a heuristic to determine a sub-expressions new location within an expression and to determine which sub-expressions should be distributed. The report proposes the use of *expression trees* to store sub-expressions and their associated ranks, and are forward-propagated.

Our approach utilises the operand ranking scheme detailed by Cooper and Briggs, however their arithmetic re-association implementation is replaced with a list-based method simplifying re-association and removing the need for any propagation.

3.2 Finding Missed Compiler Optimizations by Differential Testing

This report investigates the usage of differential testing to detect missed compiler optimisations. Barany [10] generates random programs with well defined source code constraints and compares generated binaries across 3 C compilers; GCC, Clang, and CompCert. The binaries are compared with a Python tool called `optdiff`. The randomly generated code was restricted by omitting any code that would (A) invoke undefined behaviour from the language, and (B) rely on compiler-defined implementations of certain features, order of evaluation of expressions is presented as an example. The report correctly identifies some difficulties in using differential analysis on generated binaries, however, reducers are presented as a sound solution to these issues.

The report scope is extremely broad in that it attempts to catch missed opportunities from a large range of expected optimisations, some of which are explicitly stated as begin architecture-specific. The report also states that some initial results were found to be false-positive.

Our approach utilises both pre-existing and newly implemented unit tests to determine missed optimisation opportunities. Final IR graphs are compared after all compiler phases have been completed, enabling integrated testing of our modifications.

3.3   Redundancy Elimination Revisited

Cooper, *et al.* [13] present improvements to the method of redundancy elimination through arithmetic re-association. The re-association methods proposed utilise hash-based value numbering to identify equivalent sub-expressions. A method of scalar- replacement is also given, however, this review will focus solely on re-association. The report expresses concerns with previous methods of ranking expressions by operators or variable names, and proposes the use of a frequency-based affinity instead. Sub-expressions are ranked by their frequencies within a program source and stored in an undirected affinity graph, where redundant sub-expressions are found as *maximal-cliques*. The report expresses that finding this is not entirely easy, though many studies are available on the relevant algorithms. The method is finalised with a method to identify the found redundant sub-expressions.

The approach proposed in this report adopts a simpler method of identifying redundant operands. Instead of using affinity graphs with complicated logic, a simple list is used to ease both re-association and final expression simplifications. The method proposed by Cooper, *et al.* has a time complexity of $O(n^2)$, whereas our method comes closer to that of amortised $O(log(n))$.

Chapter 4

Methodology

4.1 Approach

In order to re-associate an arithmetic expression as described in Section 2.1, there are at least four approaches one can take. First, one could approach the expression in the form of a tree and attempt to re-associate each applicable node by swapping a node's operands where appropriate, as it is implemented currently in GraalVM [7]. This approach is by no means exhaustive; it is limited to the context of only the individual node that it is re-associating.

Alternatively, one could consider all expressions within a program at once, where operands are grouped by common factors consequently separating constants from variables, as described in Section 3.1. This can lead to an excess of computational complexity.

Continuing a heuristic-based approach, one could re-associate each expression individually, and rank each operand by their frequencies within a program. One would perform re-association via an affinity graph as outlined in Section 3.3, increasing the complexity of the implementation further.

The fourth approach, and the one used for this investigation, transforms individual expressions from a tree of operands to a simple list. This approach views each expression in its entire context and allows it to be re-associated and simplified by swapping and removing elements respectively. This approach is detailed in Appendix 1 and its individual components will be further discussed in the sections to follow.

4.2 Graph Preparation

To maximise the effectiveness of the *ReassociationPhase*, the IR graph is altered prior to applying any re-association techniques. Arithmetic operator nodes within

the graph may have been altered by a previous compiler phase. As it may allow further re-association, these operators should be reverted to their original representation. The multiplication operator is often converted to a left-shift operator, a technique discussed in Section 2.2.1. These transformations prohibit our approach because a multiplication operator is associative whereas a left-shift operator is not. Currently, the GraalVM compiler applies this reversion, however an extension is proposed.

With our approach, expressions of the form `x - y` are altered to the form `x + (-y)`, allowing the compiler to recognise the expression as a chance for re-association where it previously could not. For the same reason, when the compiler is attempting to re-associate a multiplication-expression, operands of the form `-x` are transformed to sub-expressions of the form `x * -1`.

4.3   Expression Tree Flattening

To make re-association as simple a process as possible, arithmetic expressions need to be structured in a way such that both operand re-ordering and operand removal are easily and efficiently applicable. Our approach represents expressions as *lists of operands*. Arithmetic expression graphs are traversed for applicable operands, as discussed in Section 2.3.1. A breadth-first traversal of the graph is conducted, during which nodes that require further traversing are maintained in a *work-list* whilst the remaining nodes form the resulting list of operands. Within this process, shown in Appendix A.2, the original order of the operands is maintained to ensure deterministic compilation.

4.4   Operand Ranking

Operand ranking plays a vital role in facilitating the re-association of arithmetic expressions. It serves as a pre-processing step that enables the grouping of similar terms within an expression, as discussed in Section 2.1. The approach proposed in this report assigns numerical ranks to operands by identifying relationships between terms, even when they appear within unary or binary arithmetic operators. This heuristic ranking scheme, detailed in Appendix 4, allows for re-association through a simple sorting of the list of operands. This re-association ensures that similar terms are positioned adjacently. This operand ranking and sorting approach lays the foundation for the compiler to easily identify potential expression simplifications in the subsequent optimisation step.

4.5 Expression Optimisation

The expression optimisation step aims to reduce the complexity of an arithmetic expression while preserving its semantic value. The proposed approach begins by applying constant folding, evaluating and simplifying static numerical terms within the expression. For addition-expressions, the optimisation process identifies and eliminates redundant operands, such as negations of identical terms (*e.g.* `x - x`). Following these initial simplifications, our approach leverages the distributive and commutative properties of the addition operator to perform strength reduction. This involves consolidating operands with common factors, transforming additive forms into multiplicative, and combining coefficients when possible. The re-associated operands, resulting from the previous ranking and sorting step, enable the identification of these simplification opportunities through efficient linear scans. By systematically applying these optimisation techniques, complex arithmetic expressions are transformed into their simplified counterparts, leading to improved program performance and reduced computational overhead.

4.6 Expression Tree Reconstruction

After applying the expression optimisation techniques discussed in Section 4.5, our approach must reconstruct the original expression to reflect the simplifications and re-associations performed. The expression tree reconstruction algorithm, detailed in Appendix A.3, takes the re-associated and optimised list of operands and rebuilds the binary expression tree in a bottom-up manner. The algorithm iteratively processes pairs of operands until the entire optimised tree is reconstructed. This approach ensures that the resulting expression tree can be seamlessly integrated back into the program's graph for further compilation phases.

4.7 Testing

To ensure the effectiveness of the proposed arithmetic re-association approach, a *test-driven development* (TDD) methodology was employed throughout the project. TDD involves creating test cases before implementing the corresponding functionality, allowing for incremental development and continuous validation of the system's behaviour. This approach was adopted to gauge the progression of the project and maintain code quality at each stage of development.

Unit testing formed the core of the testing strategy, focusing on verifying the correctness of individual components and algorithms. The existing test suite of the GraalVM compiler was leveraged, and additional unit tests were created to cover

specific scenarios related to redundant operand elimination and complex expression simplification. These new test cases can be found in Appendix B.1 and Appendix B.2 respectively.

The unit tests were designed to validate the expected behaviour of the re-association and optimisation algorithms. Each test case consisted of an input expression and its corresponding expected optimised form. By comparing the actual resulting graph structure of the implemented algorithms with the expected graph structure, the correctness of the re-association and simplification processes could be verified.

The test-driven development approach allowed for the early detection and resolution of issues, ensuring that the implemented functionality aligned with the desired outcomes. The iterative nature of TDD facilitated the refinement of the algorithms and the handling of edge cases, resulting in a more robust and reliable implementation. Furthermore, the integration of the new unit tests into the existing GraalVM test suite provided a means to validate the compatibility of the proposed approach with the overall compiler infrastructure. This integration testing helped identify any potential conflicts or regressions introduced by our modifications to the GraalVM compiler.

By employing the use of integrated unit tests through the principles of test-driven development, the development process aimed to deliver a well-tested arithmetic re-association approach that seamlessly integrates with the current GraalVM compiler.

4.8   Problems

Throughout the implementation process some problems arose. It is no small task to modify an already large code base, especially one like GraalVM where in places documentation is scarce. This made it difficult to know the most optimal method of completing actions relating to IR graph manipulation. This was overcome by observing and mimicking the styles of code already present.

Another problem came from utilising the proposed operand ranking scheme. Although IR nodes may be equivalent in value, it is not certain that they will adopt the same GVN identifiers. This may seem contradictory to the nature of GVN, however it may be possible that this is a valid approach to ensure values stored within variables are queried appropriately in concurrent environments. This was overcome by comparing variable node's locations in memory to determine node equivalence.

The final problem was found when utilising the unit tests within the GraalVM compiler. The tests ensure exact equivalence of resulting IR graphs, producing some false negative results. This is because our proposed approach eagerly re-associates expressions, even when no further simplification may be found. To temporarily overcome this issue, tests with negative results were manually checked for real graph differ-

ences. To appropriately fix this issue, the testing methods should be relaxed to only test the resulting value of the expression, and not its entire graph structure. This may introduce further problems in regards to the determinism of the compiler.

Chapter 5

Results and Discussion

Analysis of the GraalVM compiler's current re-association method compared to our proposed list-based implementation reveals significant improvements in optimisation capabilities. The results demonstrate the effectiveness of flattening arithmetic expression trees into linear lists of operands prior to re-association and simplification. Operand Elimination Tests, shown in Fig. C.1 through to Fig. C.5, illustrate that our approach consistently produces more optimal graph structures whilst maintaining semantic equivalence. The current GraalVM implementation, when processing expressions such as `x + y - x + x - y + z`, produces graphs containing redundant arithmetic operations. In contrast, our list-based implementation successfully identifies and eliminates these redundancies, producing the simplified expression `x + z`. This transformation is evidenced by the reduced size of the resulting graphs.

Multiplication simplification tests, presented in Fig. C.6 through to Fig. C.8, further demonstrates the advantages of our approach. The list-based implementation successfully identifies opportunities for strength reduction and term consolidation that are not captured by the current graph-based method. For instance:

1. When processing the expression `(x * 3) + (x * 4)`, our implementation consolidates the common factor `x` and combines the constant terms, yielding `x * 7`.

2. More complex expressions containing both variables and constants, such as `(x * 3) + (x * y) + (x * 5)`, are transformed into `x * (y + 8)`, demonstrating effective factorisation.

3. The implementation handles nested expressions effectively, as shown in Fig. C.8, where `(x * y) + (x * z) + (x * 6) + w` is optimised to `x * (y + z + 6) + w`.

   In all test cases shown, the proposed implementation produces intermediate representations with reduced complexity compared to the current GraalVM com-

14

piler. This reduction in graph complexity not only indicates more effective optimisation but also suggests potential improvements in compilation efficiency. These results provide evidence that representing arithmetic expressions as sorted lists of ranked operands facilitates the identification and application of optimisations that are missed by the current approach.

Chapter 6

Conclusions

In summary, this investigation has demonstrated that a list-based approach to arithmetic re-association within the GraalVM compiler offers significant advantages over the current graph-based implementation. The proposed method successfully identifies and applies optimisations that were previously unattainable, whilst maintaining the semantic correctness of the transformed expressions.

The effectiveness of our approach stems from its ability to view arithmetic expressions in their entirety, rather than being limited to individual nodes within a graph-structure. By flattening expression trees into linear lists of ranked operands, the implementation can more readily identify opportunities for expression simplifications.

The success of this list-based implementation suggests that similar approaches may be beneficial in other areas of compiler optimisation where current graph-based methods face limitations. This work contributes to the ongoing development of the GraalVM compiler's optimisation capabilities and demonstrates the value of exploring linear-based approaches in compiler design.

6.1 Future Work

Further research opportunities exist in extending the proposed approach to implement re-association and simplifications for more arithmetic operators than just multiplication and addition.

The formal verification of our proposed algorithms, through automatic theorem proving provides a crucial direction for future work, particularly given recent successes in verifying GraalVM's graph transformations [2].

Finally, the evaluation of this approach on larger code bases, especially those containing complex numerical computations, would provide valuable insights into the proposed implementation's real-world impact on compilation speed and program

runtime performance. This comprehensive testing would more accurately quantify the benefits of reduced graph complexity and may identify potential scalability constraints of the proposed list-based approach.

Appendix A

Algorithm Listings

A.1   Expression Re-Association Algorithm

Algorithm 1 Expression Re-Association

1: **procedure** REASSOCIATEEXPRESSION(G)
2:      **for all** Node $n \in G$.nodes **do**
3:          **if** n is not (AddNode or MulNode) **then**
4:              continue
5:          **if** (n is not alive) or (n has no usages) **then**
6:              continue
7:          $expTree \leftarrow$ empty list
8:          $expOps \leftarrow$ empty list
9:          FlattenExpressionTree(n, $expTree$, G)
10:         **for all** RankedValue $rv \in expTree$ **do**
11:            **if** rv.rank $= 1$ **then**
12:               $newRank \leftarrow$ ComputeOperandRank(rv.value)
13:               $expOps$.Append(RankedValue(rv.value, $newRank$))
14:            **else if** rv.rank > 1 **then**
15:               **for** $[1 \ldots rv.\text{rank}]$ **do**
16:                  $newRank \leftarrow$ ComputeOperandRank(rv.rank)
17:                  $expOps$.Append(RankedValue(rv.value, $newRank$))
18:       Sort($expOps$)
19:       $newVal \leftarrow$ OptimiseExpression(n, $expOps$)
20:       **if** $newVal$ is not $null$ **then**
21:          **if** $newVal = n$ **then**
22:             continue
23:          $newVal \leftarrow G$.AddOrUnique($newVal$)
24:          G.ReplaceAllUsages(n, $newVal$)
25:          continue

26: **if** $expOps$.size $= 1$ **then**
27:      $opLeft \leftarrow expOps[0]$.value
28:      **if** $opLeft = n$ **then**
29:          continue
30:      $opLeft \leftarrow G$.AddOrUnique($opLeft$)
31:      G.ReplaceAllUsages(n, $opLeft$)
32:      continue
33: $expOps$.Reverse()
34: ReconstructExpressionTree(n, $expOps$)

A.2   Expression Tree Flattening Algorithm

Algorithm 2 Expression Tree Flattening

1: **procedure** FLATTENEXPRESSIONTREE(n, $expOps$, G)
2:     $W \leftarrow$ empty list
3:     $expLeaves \leftarrow$ empty map
4:     $expLeafOrd \leftarrow$ empty list
5:     W.Append(RankedValue(n, 1))
6:     **while** W is not empty **do**
7:         $curVal \leftarrow W$.TakeFirst()
8:         $curOp \leftarrow curVal$.value
9:         **for all** Node $in \in curOp$.inputs **do**
10:            $weight \leftarrow curOp$.rank
11:            **if** in is BinaryArithmeticNode **then**
12:                **if** in.nodeType $= n$.nodeType **then**
13:                    W.Append(RankedValue(in, $weight$))
14:                    continue
15:            **if** $in \notin expLeaves$ **then**
16:                **if** in.usageCount $\neq 1$ **then**
17:                    $expLeafOrd$.Append(in)
18:                    $expLeaves[in] \leftarrow weight$
19:                    continue
20:            **else**
21:                $prevWeight \leftarrow expLeaves[in]$
22:                $expLeaves[in] \leftarrow prevWeight + weight$
23:                **if** in.usageCount $\neq 1$ **then**
24:                    continue
25:                $weight \leftarrow expLeaves[in]$
26:                remove in from $expLeaves$
27:            **if** n is MulNode and in is NegateNode **then**
28:                $inVal \leftarrow in$.value
29:                $newMul \leftarrow G$.AddOrUnique(MulNode($inVal$, -1))
30:                G.ReplaceAllUsages(in, $newMul$)
31:                W.Append(RankedValue($newMul$, $weight$))
32:                continue
33:            $expLeafOrd$.Append(in)
34:            $expLeaves[in] \leftarrow weight$

35:　　**for all** Node $node \in expLeafOrd$ **do**
36:　　　　**if** $node \in expLeaves$ **then**
37:　　　　　　$weight \leftarrow expLeaves[node]$
38:　　　　　　$expOps$.Append(RankedValue($node$, $weight$))

39:　　**if** $expOps$ is empty **then**
40:　　　　**if** n is AddNode **then**
41:　　　　　　$identity \leftarrow G$.AddOrUnique(ConstantNode(0))
42:　　　　**else if** n is MulNode **then**
43:　　　　　　$identity \leftarrow G$.AddOrUnique(ConstantNode(1))
44:　　　　$expOps$.Append(RankedValue($identity$, 1))

A.3   Expression Tree Reconstruction Algorithm

Algorithm 3 Expression Tree Reconstruction

Require: $expOps > 1$

 1: **procedure** RECONSTRUCTEXPRESSIONTREE(n, $expOps$)
 2:     $op \leftarrow n$
 3:     $notRewritable \leftarrow$ empty list
 4:     **for all** RankedValue $rv \in expOps$ **do**
 5:         $notRewritable$.Append(rv.value)
 6:     $i \leftarrow 0$
 7:     **while** true **do**
 8:         **if** $i + 2 = expOps$.size **then**
 9:             $newLHS \leftarrow expOps[i]$.value
10:             $newRHS \leftarrow expOps[i + 1]$.value
11:             $oldLHS \leftarrow op$.lhs
12:             $oldRHS \leftarrow op$.rhs
13:             **if** $newLHS = oldLHS$ and $newRHS = oldRHS$ **then**
14:                 break
15:             **if** $newLHS = oldRHS$ and $newRHS = oldLHS$ **then**
16:                 op.lhs $\leftarrow newRHS$
17:                 op.rhs $\leftarrow newLHS$
18:                 break
19:             **if** $newLHS \neq oldLHS$ **then**
20:                 op.lhs $\leftarrow newLHS$
21:             **if** $newRHS \neq oldRHS$ **then**
22:                 op.rhs $\leftarrow newRHS$
23:             break
24:         $newRHS \leftarrow expOps[i]$.value
25:         **if** $newRHS \neq op$.rhs **then**
26:             **if** $newRHS = op$.lhs **then**
27:                 $oldRHS \leftarrow op$.rhs
28:                 op.lhs $\leftarrow oldRHS$
29:                 op.rhs $\leftarrow newRHS$
30:             **else**
31:                 op.rhs $\leftarrow newRHS$

32:	**if** (*op*.lhs is AddNode) or (*op*.lhs is MulNode) **then**
33:	**if** *op*.lhs $\notin$ *notRewritable* **then**
34:	$op \leftarrow op.\text{lhs}$
35:	$i \leftarrow i + 1$
36:	continue
37:	$newNode \leftarrow n.\text{CopyWithInputs}()$
38:	$newNode.\text{ClearInputs}()$
39:	$op.\text{lhs} \leftarrow newNode$
40:	$op \leftarrow newNode$
41:	$i \leftarrow i + 1$

A.4   Operand Ranking Algorithm

Algorithm 4 Operand Ranking

1: **procedure** OPERANDRANK(n)
2:     **if** n is ConstantNode **then**
3:         return 0
4:     **else if** n is BinaryArithmeticNode **then**
5:         return (max(OperandRank(n.LHS), OperandRank(n.RHS)) $+1$)
6:     **else if** n is NegateNode **then**
7:         return OperandRank(n.value)
8:     return n.id $\triangleright$ return n's unique GVN ID

Appendix B

Test Code Listings

B.1 Operand Elimination Tests

```
1  // Variable Elimination 1
2  public static int
3  testVariableElimination_Actual()
4  {
5      return (x + y * 0 + z - z);
6  }
7
8  public static int
9  testVariableElimination_Expected()
10 {
11     return (x);
12 }
13
14 // Variable Elimination 2
15 public static int
16 testVariableElimination2_Actual()
17 {
18     return ((x - y) + (y - z) + z);
19 }
20
21 public static int
22 testVariableElimination2_Expected()
23 {
24     return (x);
25 }
26
```

```
27  // Variable Elimination 3
28  public static int
29  testVariableElimination3_Actual()
30  {
31      return (x + y - x + x - y + z);
32  }
33
34  public static int
35  testVariableElimination3_Expected()
36  {
37      return (x + z);
38  }
39
40  // Variable Elimination 4
41  public static int
42  testVariableElimination4_Actual()
43  {
44      return ((5 * x) - x);
45  }
46
47  public static int
48  testVariableElimination4_Expected()
49  {
50      return (4 * x);
51  }
52
53  // Variable Elimination 5
54  public static int
55  testVariableElimination5_Actual()
56  {
57      return ((5 * x) + x + x);
58  }
59
60  public static int
61  testVariableElimination5_Expected()
62  {
63      return (7 * x);
64  }
65
```

Code Snippet B.1: Operand Elimination Tests

B.2   Multiplication Simplification Tests

```
1  // Simplify Multiplication Expression 1
2  public static int
3  testMulAddSimplify_Actual()
4  {
5      return ((x * 3) + (x * 4));
6  }
7
8  public static int
9  testMulAddSimplify_Expected()
10 {
11     return (x * 7);
12 }
13
14 // Simplify Multiplication Expression 2
15 public static int
16 testMulAddSimplify2_Actual()
17 {
18     return ((x * 3) + (x * y) + (x * 5));
19 }
20
21 public static int
22 testMulAddSimplify2_Expected()
23 {
24     return (x * (y + 8));
25 }
26
27 // Simplify Multiplication Expression 3
28 public static int
29 testMulAddSimplify3_Actual()
30 {
31     return ((x * y) + (x * z) + (x * 6) + w);
32 }
33
34 public static int
35 testMulAddSimplify3_Expected()
36 {
37     return (x * (y + z + 6) + w);
38 }
39
```

Code Snippet B.2: Multiplication Simplification Tests

Appendix C

Result Listings

Please note, all results are shown with that of the current GraalVM compiler on the left-hand side, and the proposed compiler on the right-hand side.

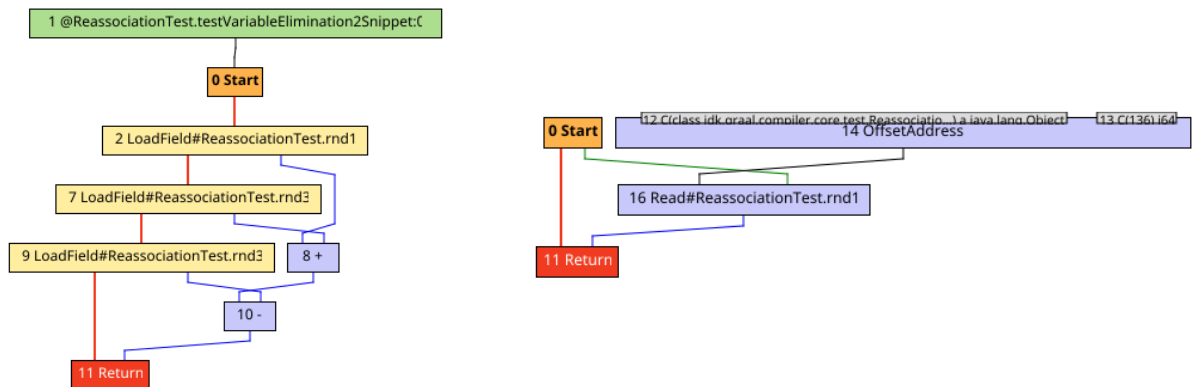## C.1 Operand Elimination Results

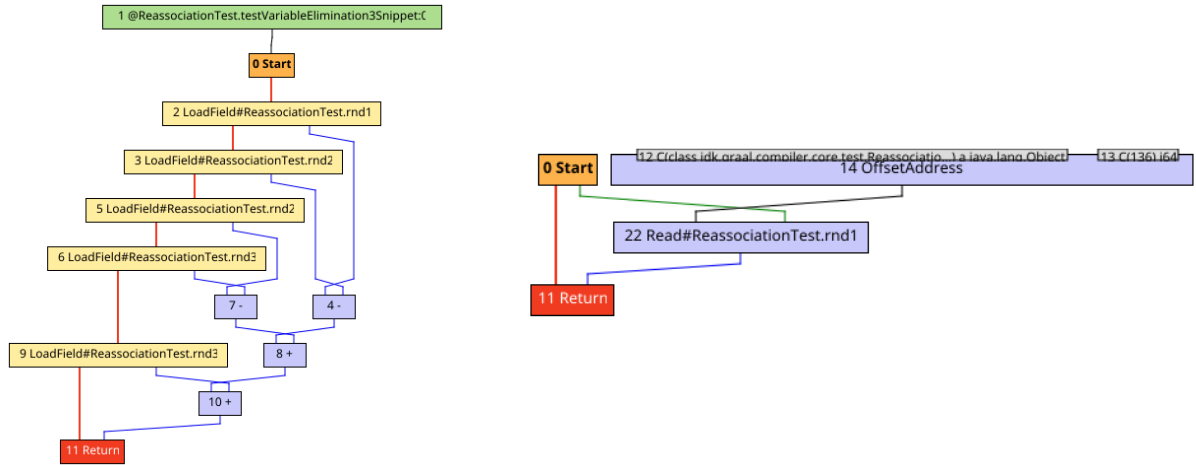Figure C.1: testVariableElimination Resulting Graphs
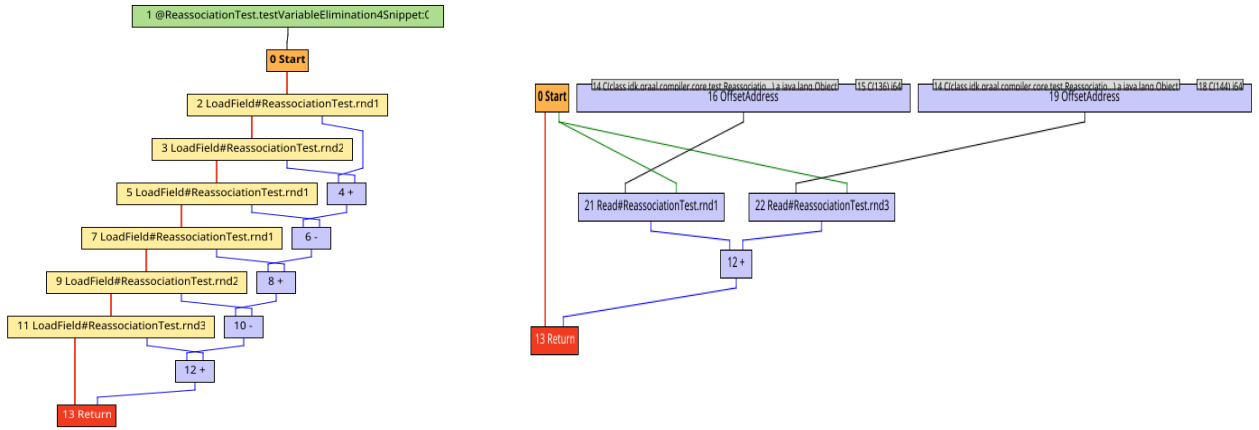
Figure C.2: testVariableElimination2 Resulting Graphs

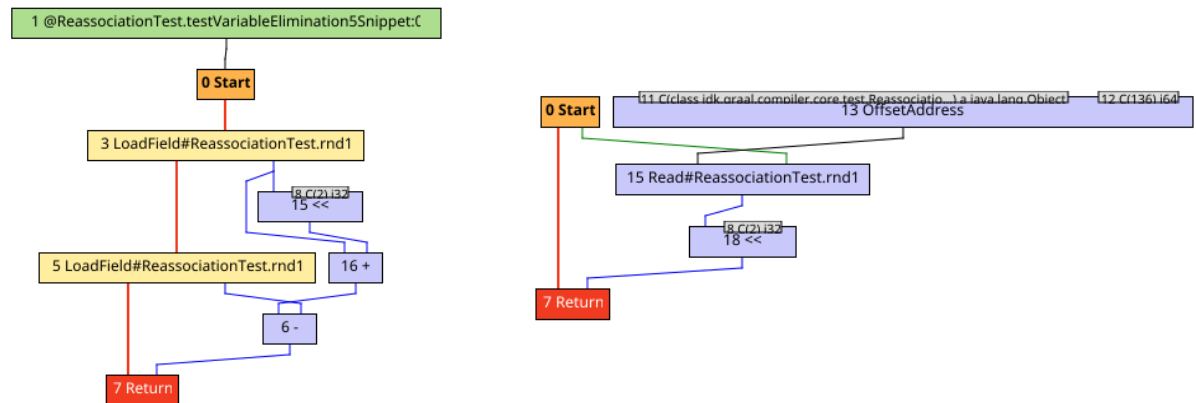Figure C.3: testVariableElimination3 Resulting Graphs

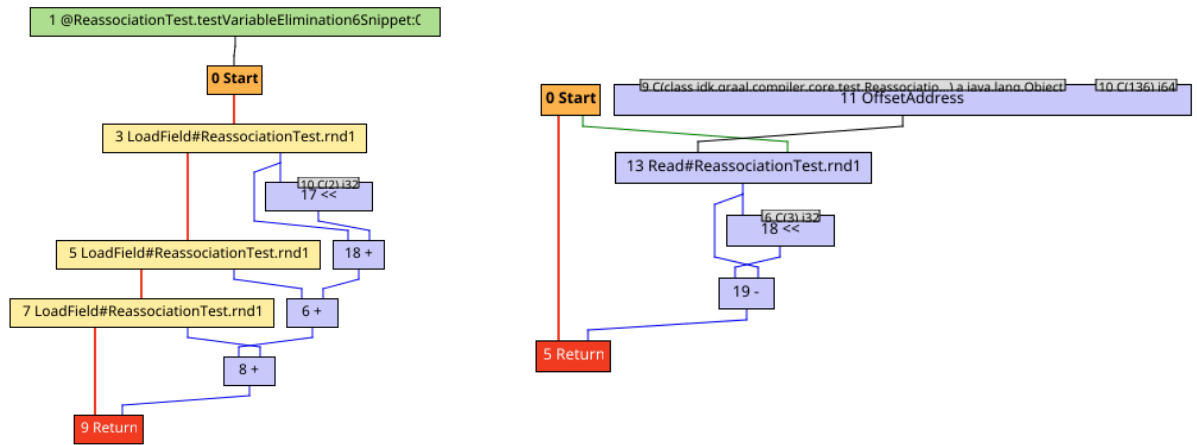Figure C.4: testVariableElimination4 Resulting Graphs

Figure C.5: testVariableElimination5 Resulting Graphs

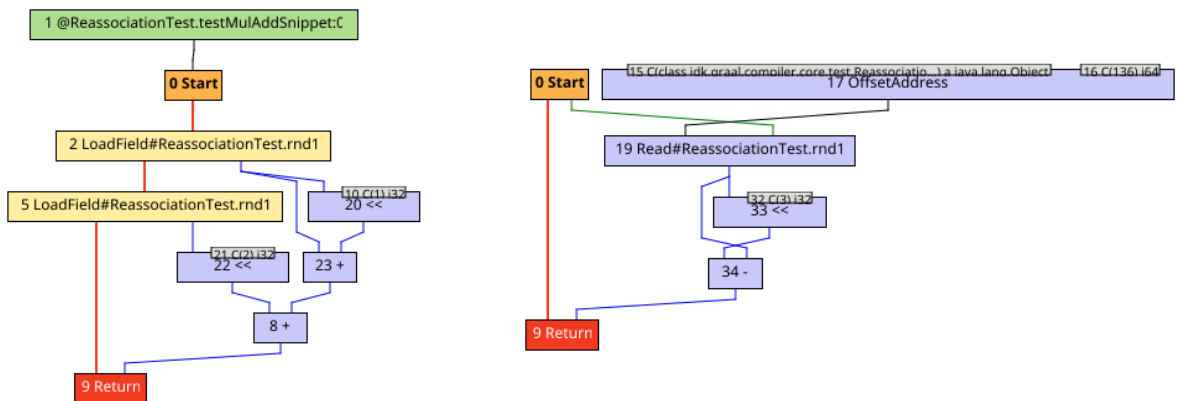# C.2   Multiplication Simplification Results
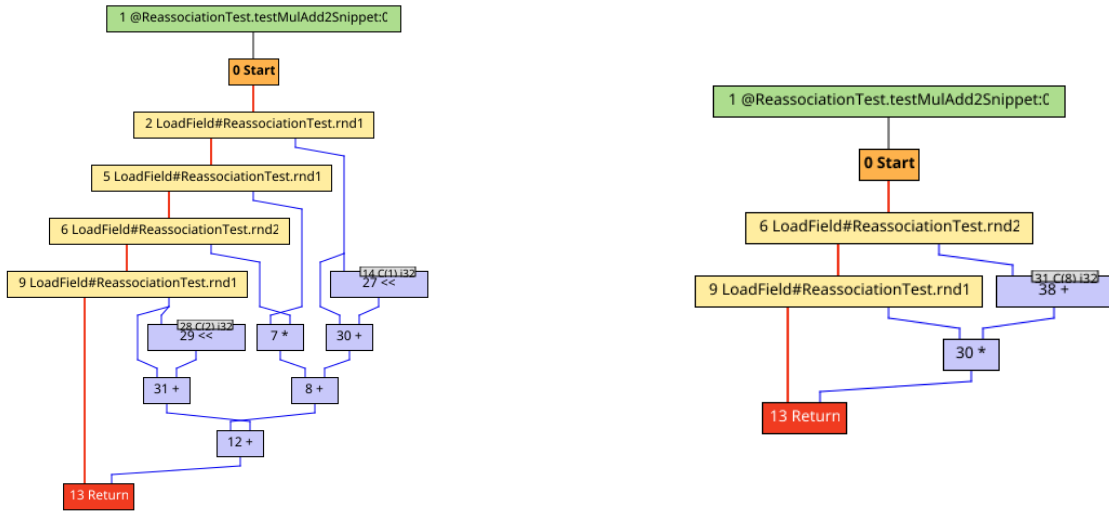

Figure C.6: testMulAddSimplify Resulting Graphs
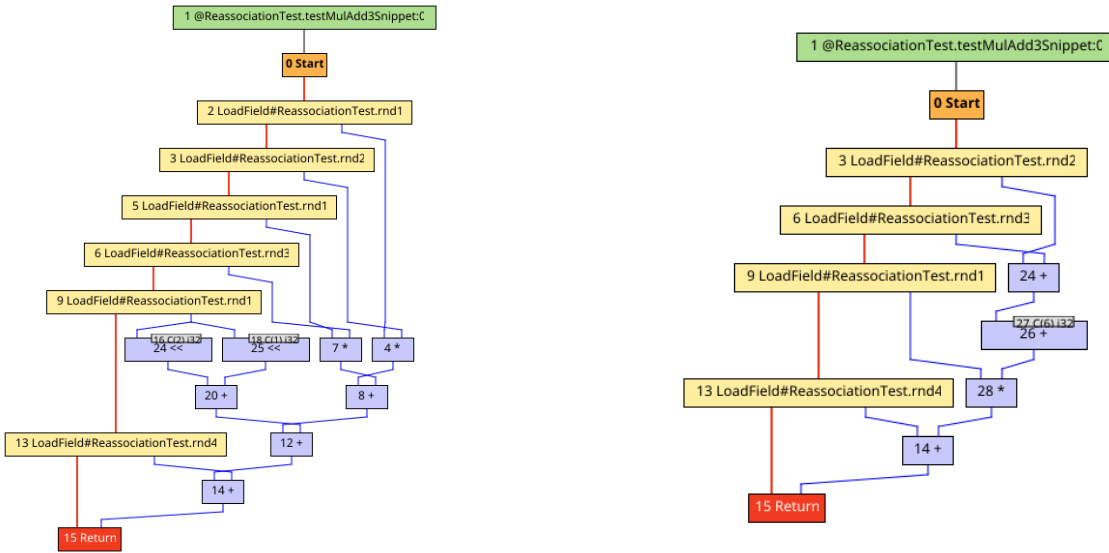
Figure C.7: testMulAddSimplify2 Resulting Graphs

Figure C.8: testMulAddSimplify3 Resulting Graphs

Bibliography

[1] Oracle. 2024. *GraalVM: An advanced JDK with ahead-of-time Native Image compilation.* https://github.com/oracle/graal (accessed Mar. 9, 2024)

[2] Brae J. Webb, Ian J. Hayes, and Mark Utting. 2023. *Verifying Term Graph Optimizations using Isabelle/HOL.* In Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3573105.3575673

[3] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, Hanspeter Moessenboeck. 2013. *Graal IR: An Extensible Declarative Intermediate Representation.* https://api.semanticscholar.org/CorpusID:52231504

[4] OpenJDK Community. 2012. *Graal Project.* https://openjdk.org/projects/graal (accessed Mar. 18, 2024)

[5] Cliff Click, Michael Paleczny. 1995. *A Simple Graph-Based Intermediate Representation.* https://www.oracle.com/technetwork/java/javase/tech/c2-ir95-150110.pdf

[6] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Chapter 12. ISBN 1-55860-320-4

[7] Oracle. 2024. *Oracle GraalVM Enterprise Edition 21.* https://docs.oracle.com/en/graalvm/enterprise/21/docs/reference-manual/java/compiler/#graal-compiler (accessed Mar. 17, 2024)

[8] Preston Briggs and Keith D. Cooper. 1994. *Effective partial redundancy elimination.* SIGPLAN Not. 29, 6 (June 1994), 159–170. https://doi.org/10.1145/773473.178257

[9] M. Sipek, B. Mihaljevic, A. Radovan. 2021. *Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM.* https://doi.org/10.23919/MIPRO.2019.8756917

[10] Gergoe Barany. 2018. *Finding Missed Compiler Optimizations by Differential Testing.* Compiler Construction (CC'18). ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179521

[11] Chris Seaton. 2017. *Understanding How Graal Works - a Java JIT Compiler Written in Java.* https://chrisseaton.com/truffleruby/jokerconf17/ (accessed Mar. 18, 2024)

[12] Martyn J. Corden, David Kreitzer. 2018. *Consistency of Floating-Point Results using the Intel Compiler or Why doesn't my application always give the same answer?.* Software Services Group, Intel Corporation. https://www.intel.com/content/dam/develop/external/us/en/documents/pdf/fp-consistency-121918.pdf

[13] Keith Cooper, Jason Eckhardt, Ken Kennedy. 2008. *Redundancy Elimination Revisited.* PACT'08, October 25–29, 2008, Toronto, Ontario, Canada. https://cscads.rice.edu/p12-cooper.pdf

[14] Oracle. 2015-2022. *GraalVM Reassociation Phase Source.* https://github.com/oracle/graal/blob/master/compiler/src/ jdk.graal.compiler/src/jdk/graal/compiler/phases/common/ReassociationPhase.java (accessed Mar. 15, 2024)

[15] David Monniaux, Cyril Six. 2022. *Formally Verified Loop-Invariant Code Motion and Assorted Optimizations.* ACM Transactions on Embedded Computing Systems (TECS). ff10.114/3529507ff.hal03628646

[16] Douglas Thain. 2020. *Introduction to Compilers and Language Design Second Edition.* Chapter 5. ISBN 0-35913-804-7