Garbage collection: Reference-Sweep

Hannes Brinklert
D19, Lund University, Sweden
ha7075br-s@student.lu.se

Christoffer Fjällborg Rinaldo
D20, Lund University, Sweden
ch8602fj-s@student.lu.se

Abstract

The stack and the heap are two different parts of the memory which a programming language usually requests memory from. Both of them are used for reading and writing data, but they are used for different purposes. The goal of this paper is to describe the work of extending a compiler built at a university course that only supports stack allocation, to support arrays with heap allocation. A further goal is to implement a garbage collection algorithm Reference-Sweep, which is a combination of mark-and-sweep and reference counting, and evaluate its performance.

It was a successful implementation of heap-allocated arrays with manual memory allocation and the garbage collection algorithm Reference-Sweep. The evaluation shows that when garbage collection is enabled in the SimpliC compiler it is slower than the same program with manual memory allocation. SimpliC with garbage collection's performance was worse than a similar Go program and as expected the same was true when compared against a similar program written in C. The paper creates a framework for future implementation of other garbage collection algorithms and the implemented Reference-Sweep algorithm can be evaluated against other garbage collection algorithms.

1 Introduction

The main task of the Compilers course at Lund University is to build a compiler from the ground up. This includes scanning, parsing, name analysis, and type checking. At the end of the course, the built compiler generates assembly code. This paper uses the compiler built during the course and extends it.

The language supported by the built compiler is SimpliC which is created for educational purposes and the syntax of SimpliC is similar to the programming language C, but simplified. The specification of SimpliC states that it supports control flow, binary expressions, functions, integer variables, and basic I/O.[17].

The first goal of this paper is to use the previously built compiler that only supports stack allocation and extend it to support arrays with heap allocation and freeing of memory. The array should support nested arrays as well. The second goal is to implement a new garbage collection(GC) algorithm which is a combination of mark-and-sweep and reference-counting. The bubble sort algorithm was implemented in the programming languages below to assess the performance of the implementation.

- SimpliC with GC disabled and enabled
- C compiled by GCC with no optimizations
- Go

2 Background

2.1 Attribute Grammar

Given production rules from a formal grammar that takes a string and constructs a tree of nodes (also called "derivation tree"[13]), the question becomes how one can further assign more metadata and meaning to said tree. One such method is through the use of attribute grammars. Here, the nodes can have function-defined attributes, where each function belongs to one of the grammar's productions. By introducing the attributes, information about the string can be extracted. Knuth [13], introduces two new types of attributes, synthesized and inherited. A synthesized attribute value is calculated with information from nodes below the node itself and inherited attributes with information from nodes above it [13].

Hedin [10] extended attribute grammar, by allowing attributes to point to other nodes, called reference attributed grammar(RAG). Attributes of the pointed node can be read from the node that points to it and the data does not have to be sent throughout the tree, it can be sent directly between the two nodes[10].

2.2 JastAdd

A compiler can be created using JastAdd[11] and it is an example of a meta-compiler[2]. An important part of JastAdd is "an object-oriented representation of the abstract syntax tree"[11]. JastAdd works by introducing new functionality to the classes of the tree in different aspects and it supports RAG.

2.3 x86 assembly

For our code generation, we target x86 assembly, relying on existing assemblers and linkers to produce machine code. We textually generate the assembly following the AT&T syntax.

Assembly refers to the textual representation of machine code, or the instructions that are run by the host machine. The assembly is target-dependent and varies between different computer architectures. There exist two large groupings of machine code in use today, CISC and RISC, of which x86 belongs to the CISC family [12].

Course paper, EDAN70, Lund University, Sweden
January 16, 2024.

2.4 Memory management and Garbage Collection

To be able to perform any interesting computation you need some way to manage and store data. For most programming languages this is done using two areas of our memory called the "stack" and "heap". The stack is reserved for any local data used during the execution of a function call. The heap is better suited for dynamic data.

For many programming tasks simply using the stack is enough for our memory needs. We know ahead of time how much memory we require and can therefore statically make space for it on the stack. There are cases however in which the required memory is not known ahead of time, and for these situations, a programming language requires dynamic memory management of some sort.

There are several ways in which a programming language can decide to provide this facility but the most common techniques are garbage collection and manual memory management. Some examples of languages with garbage collection include Python [4], Java [7], and Go [1]. There exists other languages like Rust which exhibit much of the characteristics of a language with manual memory management but where memory is automatically managed. [5].

Garbage collection is defined as the automatic reclamation of memory [15]. It is then an implementation detail how this is achieved. The programmer requests memory from the language runtime which at some point will try to check if that memory is still used or whether it should be retrieved. There are multiple algorithms for how this checking and retrieval should occur. We are going to look at two of them: "Reference Counting" and "Mark & Sweep".

2.4.1 Reference counting

The core idea of garbage collection is that the memory associated with some object should be reclaimed when there is no way to access that memory. In the case of reference counting this means that we keep track of how many references a certain object has. When we take a reference to it, the count is increased, and when the reference disappears the count is decreased. This happens either by the reference falling out of scope or by the object containing it being reclaimed. This can in turn trigger further decrements of other reference counts. When the count further reaches zero, we know that no one is referencing the object, and it can be reclaimed. [15]

An advantage of this approach is that most operations done by the garbage collector are inter-weaved with ordinary program execution, with each operation only managing a small amount of resources. This means that any latency perceived will also be small. [15]

A disadvantage with reference counting is that its criteria for liveliness is only local. Reference counting only checks if there exists a neighbor in the reference graph and not if that neighbor is reachable e.g from the stack. Any construction of any "circular references" [15] are therefore prone to cause memory leaks. This means the memory used by the objects will never be reclaimed which could cause our program to run out of memory.

2.4.2 Mark & Sweep

Another approach used by many garbage collectors is called Mark & Sweep. Here the GC periodically halts the execution to perform a marking phase during the still live objects are marked. This can be done by walking down the stack, identifying each reference into the heap, following that reference, and then "marking" it (e.g setting some 'live' bit to 1). The GC then iterates over the items on the heap which it has allocated memory for and checks to see whether they are still live or not. If they are not, they are reclaimed. [15]

One disadvantage of the Mark & Sweep approach is the linear nature of how we determine liveliness and how we reclaim memory. When counting live references from the stack the cost of doing such grows with the stack size. The same holds true for the reclamation phase. [15]

2.5 Advantages of GC over manual memory management

Garbage collection eliminates the need for the programmer to manually keep track of the liveliness of their objects. This has a number of benefits both in terms of security and development costs.

One advantage is that more mental effort can be placed on the business logic rather than keeping track of when to reclaim an object's resources. In most development tasks this probably increases development speeds and reduces cost.

Another advantage is that garbage collection practically exterminates one of the largest sources of security vulnerability-inducing bugs. An often-cited report from Microsoft states that around 70% of the yearly CVEs reported are memory safety issues. [3]

3 Solution

3.1 Grammar

The grammar of SimpliC was extended to support arrays and the array variable holds a memory address. It was done by extending the abstract grammar of identifier declaration to hold an integer value PointerLevel, which indicates the dimensions of an array. An integer variable has a PointerLevel of zero.

3.1.1 Name and type analysis

The name analysis from the initial SimpliC implementation worked with arrays, due to the extension of identifier declaration. The name analysis checks that a variable or an array is declared before it is used.

A new type "Array" was introduced and it contains an integer called PointerLevel, same functionality as in the identifier

```java
public class Heap {

    private Map<Integer, int[]> heap = new HashMap<>();

    public int calloc(int size) { }

    public void free(int address) { }

    public void write(int address, int index, int value) { }

    public int read(int address, int index) { }
}
```

Figure 1. Interpreter heap interface

declaration. The type analysis had to be extended with support for this new type and previous assumptions of SimpliC had to be removed. A variable's type is associated with the type of its declaration and therefore every variable has a reference to its declaration. The type analysis is done by checking that the type of a variable is compatible with the type of the expression.

3.2 Interpreter

For the Interpreter our addition of arrays requires us to introduce something equivalent to a "heap". We want some way to request enough memory to hold a given number of elements and an API through which we can interact with that memory. We also want the API between the internals of the interpreter and our heap to only require knowledge of language-specific constructs rather than Java-specific ones (evaluation of e.g retrieving an element from a SimpliC array should not require knowing that the array is backed by a Java array).

Our heap implementation consists of a Java class which is then propagated throughout the interpreter, see Figure 2. Any interpreter internals that interact with this heap only communicate with it through the constructs of indices, values, and addresses without knowing how any of the memory is backed. This achieves loose coupling.

3.3 Heap allocation using manual memory management

We implement heap allocation by wrapping the C functions *calloc* and *free*. This is done by creating a C file with two functions called *yoink* and *yeet* which wrap the calloc and free function calls. The signature of our yoink function is simplified such that element size is omitted with the only required argument being the number of elements. This allows us to simplify the language semantics with an array of numbers being of equal size to an array of addresses. It is decided that each position takes 64 bits(8 bytes). The wrapper function calls calloc using the provided number of elements and an address is returned. When calling the wrapper function around free, the address to be freed is sent as an argument, and nothing is returned.

When assembling and linking the final binary, a few changes needed to be made for a valid program to be created. Since we now link with glibc we needed our build step to generate an entry point into the program that would dynamically load the required library functions. This means that we could no longer generate a *_start* label ourselves in the assembly generation step of our compiler (which would result in a conflict). The start label was removed and instead, the main label was declared as global.

The following tutorial[9] was used to connect our C library and the compiler-generated assembly code. The C library is linked to the compiler's generated assembly code and run using *gcc* with the below commands.

```
$ gcc our_c_lib.c compiler_output.s -o program
$ ./program
```

Linking to C code also required us to adhere to its calling convention. The calling convention of SimpliC is that arguments are pushed on the stacked in reverse order. However, this is not the same as when calling C functions with arguments, which was learned from the tutorial. The tutorial states that the first argument should be placed in the *rdi* register, the second argument in another register, and so on [9].

3.3.1 Stack alignment

When calling a C function, the stack has to be aligned, which means that the value of the stack pointer must be equal to $16 \cdot n + 8$ for some n [9]. To guarantee that this convention is upheld when calling the C functions in our library we implement two assembly procedures that check the value of the stack pointer, altering it if necessary. Before and after calling the free and malloc wrapper functions these procedures are run. The first one checks if the stack pointer has the correct value and if it is not correct, it subtracts 8 bytes from the stack pointer. If the stack pointer was updated the register *r15* is updated with a 1. Later on, after the wrapper function call is complete the second function checks if there is a 1 in the *r15* register, if so then 8 bytes are added to the stack pointer and *r15* is updated to a 0, otherwise not.

3.4 Garbage Collection

The garbage collection algorithm that is implemented in the paper uses elements borrowed from both reference counting and mark-and-sweep, which we named Reference-Sweep. The incremental marking of reachable objects from Reference Counting and the liveliness check and reclamation pass from Mark and Sweep are combined into a GC algorithm that allows for a simpler implementation. Firstly, during program execution, the garbage collection algorithm counts the number of pointers from the stack to an object on the

```
struct array_t
{
    int is_integer;
    int live;
    unsigned int stack_counter;
    int length;
};
```

Figure 2. Structure of one instance in heap

heap, i.e. stack reference counting. Only the direct pointers are counted. This means that only array declarations and reassignments are counted.

When the program tries to request memory, but there isn't enough free memory left, the second phase of the GC algorithm occurs. At this point, the GC already knows which objects on the heap are reachable from the stack and only needs to check which heap objects are reachable from other heap objects. The GC algorithm can immediately iterate through all the heap objects reachable from the stack and then recursively calculate which other objects are reachable. This phase is called the mark phase. The third phase is the sweeping phase, where the algorithm sequentially goes through the objects on the heap and collects all object that is not live. The garbage collection is complete.

3.4.1 Structure of heap

The garbage collection is implemented in the same C file as *yeet* and *yoink*, extended with multiple functions. Furthermore, *yoink* was extended to include the *PointerLevel* as an argument, in order to keep track of whether the array contains memory addresses or integer values.

Using conditional compilation using a C macro, the behaviour of *yoink* is different between when garbage collection is enabled versus when it is not. When garbage collection is enabled and the program requests memory, an additional *array_t* struct as seen in Figure 2 is allocated on the heap in addition to any memory requested by the program. Using pointer arithmetic a pointer to the requested memory is constructed and returned back to the program. This address is also saved in a global array for later reference by the mark and sweep phases. Using two additional functions that we implemented, this conversion from an array pointer into a pointer to requested memory and vice versa can easily be performed.

The metadata struct allows us to store bookkeeping details right next to the related memory. The is_integer member in Figure 2, is 1 if the array stores integer values and it is used to distinguish arrays containing references to other arrays and arrays containing integers. It is used to stop the recursive reachable check, in the mark phase of garbage collection. Live is also used during the marking phase, to indicate if the array is reachable from the stack. It also serves as a check to whether we have already visited it during another recursive step. The stack_counter counts how many pointers there are from the stack to some array on the heap is updated during program execution. The member length is for indexing the array and is used when finding array objects referenced from other array objects.

3.4.2 Stack reference counting & depth

To be able to count the stack reference, the garbage collection algorithm has a function called *update_counter*, which decrements and increments the stack pointer of the old and new address respectively. Given an address that is zero (NULL pointer), the address is ignored. The previously mentioned function is called when a variable of type array is declared or assigned a value. To support repeated re-declarations of a variable in a loop, the initialization of local variables on the stack had to be changed from subtracting the stack pointer to pushing value 0 to the stack.

To correctly increment and decrement the references made during program execution, we introduced a notion of "function depth". We keep track of and associate each reference made from the stack with what essentially is the number of function frames constructed so far. We do this by implementing a linked list that contain both the reference (memory address) and the depth at which it was made. Before a function is called, and before its arguments are evaluated, the depth is increased. After the function returns, the depth is decreased and all the references that were made during the previous depth level have their stack counter decremented. This means that we then know all references that were made during that specific function invocation, which now should be decremented. We also simplify the handling of functions returning references (e.g a function returning the address to an array that it has constructed) by disallowing them in context were the value is not immediately bound to a variable.

3.4.3 Reclaiming memory

The garbage collection is potentially run during a SimpliC calloc call, if the current size of the heap and the requested allocated space exceeds the arbitrary heap max size. The first part of reclaiming memory is the marking reachable object as live. The algorithm goes through the array of memory addresses and calls a recursive function *mark_reachable* if the stack counter in the associated *array_t* struct is larger than 0. The function marks the object as live and recursively calls itself on the reachable objects. The recursion stops if the object are already marked as live or if the array contains integer values(*PointerLevel* equal to 0).

The last phase of "sweeping" is done by sequentially going through the array of memory addresses again and freeing the memory if is not marked as live. The linked list of references

with their depth levels is then iterated through, and any node that now points to invalid memory is freed and removed as well. The live objects are updated to not being live anymore, to prepare for the next garbage collection.

4 Evaluation

The figures 3 and 4, show the heap memory usage over the number of instructions with the same program using manual memory allocation and garbage collection respectively. The program sorts an array of 500 values in reversed order using bubble sort. The garbage collection algorithm has an arbitrary heap maximum size which in Figures 3 and 4 was set to 15 kB. This amount was chosen to control the number of times the garbage collection is run.

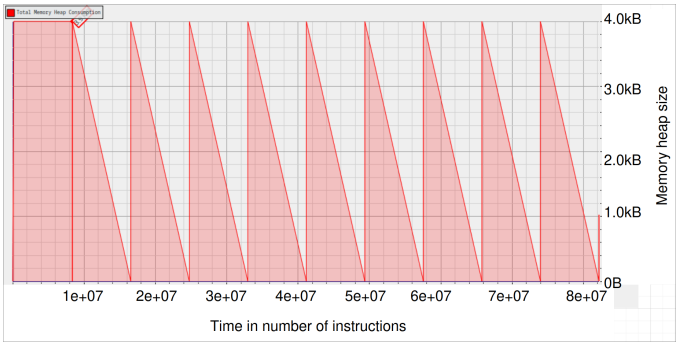

Figure 3. Bubble sort with manual memory allocation run 10 times

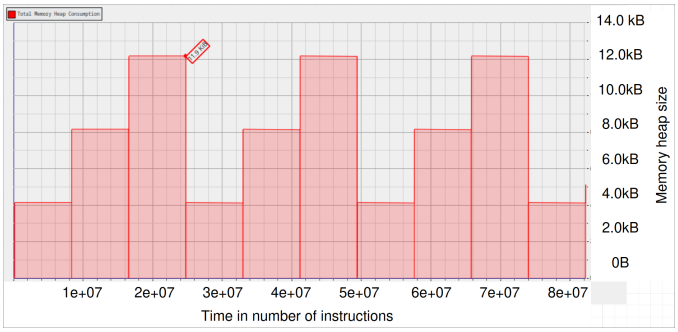

Figure 4. Bubble sort with garbage collection run 10 times

Figures 3 and 4 show a successful implementation of manual memory management and the garbage collection algorithm Reference-Sweep.

To evaluate the performance of our SimpliC implementation, we implement a bubble sort algorithm in SimpliC with GC disabled and with GC enabled. We also implement an as equivalent as possible of an implementation in C and Go (using. This is such that we are able to compare SimpliC's performance with other programming languages with manual memory management and languages with garbage collection.

The bubble sort programs read data from stdin. First, they read how many times they should sort the input, followed by the length of the input. Then, in a loop, the code reads the values to be sorted and populates them in an array. After that, copies the input into another array, sorts that array using bubble sort, and lastly, prints the sorted array.

On top of this, an evaluation script that executes each program M times for varying input sizes (N) is implemented. This allows us to retrieve the average execution times per program and by extension, per sort of the input array for many different lengths of the arrays.

We compare the execution speed of one sort attempt in our SimpliC implementation of bubble sort against itself (with and without GC) and equivalent implementations in C and Go (using go version 1.21.6). The choice of these languages allow the benchmarking methods to be identical between languages. This can be seen in Figure 5

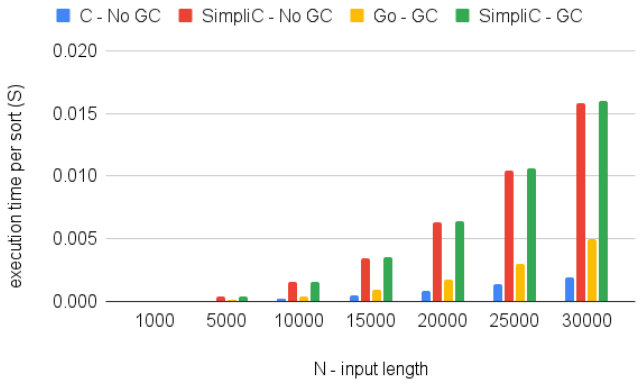

Figure 5. Different implementations of bubble sort in languages with and without automatic memory management

5 Related work

5.1 ZGC

A modern implementation of a GC that is also used in production is the ZGC garbage collector. ZGC is a new garbage collection algorithm in OpenJDK[16]. It uses three pauses during one clean-up. During the first pause, it recognizes the current objects and will not consider new objects allocated after this pause during this clean-up, and pushes the root pointers to a stack. After the pause the marking begins, it is done by multiple threads and they mark objects live. Then, a second pause is used to check that the marking is completed by the threads. After this, the algorithm selects pages in the heap that do not contain a lot of live objects, so-called evacuation candidates(EC) set. Now is the last pause and it consists of marking all pointers as bad. During the pause,

it also moves objects pointed by root pointers from the EC set to other pages and marks all root pointers as good. Then the final part begins, threads concurrently move live objects from the EC set to other pages. The pages in the EC set are now cleared and the clean-up is complete[16]. As previously mentioned under 2.4.2, one drawback with mark and sweep is that the phase of reclaiming memory grows with the size of the heap. However, Yang and Wrigstad[16] explain that as the heap grows larger, ZGC tries to maintain short pauses when garbage collecting.

5.2 Boehm GC

An often-cited modern C implementation of a GC is the Boehm-Demers-Weiser conservative garbage collector. [8] This GC is written to be used as a replacement for the POSIX memory management functions. The library is used not only to implement programming languages with a GC but also for its improvement in ergonomics. Application authors who are programming in C but who want the convenience of a GC can use the library over the ordinary malloc and free. An example of a project that has chosen to go this route is Inkscape. [6]

5.3 Silver

Another attribute grammar framework is Silver, developed at the University of Minnesota[14]. Silver is "an extensible attribute grammar specification system"[14]. Van Wyk et al.[14] explain that JastAdd and Silver have implemented different concepts in attribute grammar.

6 Conclusion

In this report, the extension of SimpliC with heap allocation and automatic memory management is described. An introduction to relevant areas and technologies is given. A garbage collector using elements from both reference counting and mark and sweep is successfully implemented and its performance in comparison to SimpliC with manual memory management as well as other languages with and without GC is done.

We find that our implementation of bubble sort performs worse than an equivalent implementation in C and Go, which was anticipated. A surprising find is that the difference between SimpliC with and without garbage collection is not that large, though this could be because of limitations in our benchmarks rather than actual performance characteristics.

A reasonable explanation for the (relative) poor performance of our GC implementation can be found in how we interface with the C library, as well has the ad-hoc generation of assembly code. For every interaction with the C library, the assembly code is required to call a routine that makes sure the stack is aligned. The overhead of this has not been measured but it is reasonable to say that a large portion of the executed instructions for any SimpliC program with be from these routines. Some of this could be mitigated by better analysis (such as computing stack alignment statically).

Another explanation for the poor performance of the GC in particular is in our choice to combine Reference counting with Mark and Sweep. While our approach does give us some nice properties (with the most important one probably being how it simplifies our implementation) we are essentially paying for both the cons of each approach simultaneously. The incremental work done by the reference counting part of our GC introduces a large constant on top of any other operation. While this somewhat speeds up the mark and sweep phase the cost of this step is still high.

6.1 Future work

The work of implementing a garbage collection for SimpliC creates a basic framework for the implementation of other garbage collectors, with methods such as generational and copying garbage collection. This enables the comparison of the performance of the implemented Reference-Sweep garbage collection with the performance of other garbage collectors in the same programming language of SimpliC. Another future implementation would be to extend the SimpliC language with the support of heap-allocated structs. Then, a garbage collection algorithm's performance could be compared between structs and arrays.

Acknowledgments

We would like to give a big thanks to our supervisor *Anton Risberg Alaküla* for all the feedback and the discussions regarding the project. We would also like to thank the other supervisors as well as those responsible for the course.

References

[1] 2023. A Guide to the Go Garbage Collector. (2023). https://tip.golang.org/doc/gc-guide
[2] 2023. JastAdd.org. (2023). https://jastadd.cs.lth.se/web/
[3] 2023. A proactive approach to more secure code. (2023). https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/
[4] 2023. Python Garbage Collector. (2023). https://devguide.python.org/internals/garbage-collector
[5] 2023. Rust Memory Management. (2023). https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
[6] 2023. Tracking Dependencies. (2023). https://wiki.inkscape.org/wiki/Tracking_Dependencies
[7] 2023. The Z Garbage Collector. (2023). https://wiki.openjdk.org/display/zgc/Main
[8] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly parallel garbage collection. *Proceedings of the ACM SIGPLAN 1991 Conference: Programming Language Design &; Implementation* (1991), 157 – 164. https://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true& AuthType=ip,uid& db=edb& AN=84094616& site=eds-live& scope=site
[9] Andrew Clifton. 2019. Calling C Functions; Floating-point instructions. (2019). https://staffwww.fullcoll.edu/aclifton/cs241/lecture-asm-to-c-interop.html

[10] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.
[11] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58.
[12] Tariq Jamil. 1995. RISC versus CISC. *Ieee Potentials* 14, 3 (1995), 13–16.
[13] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.
[14] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75, 1-2 (2010), 39–54.
[15] Paul R Wilson. 1992. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*. Springer, 1–42.
[16] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 44, 4 (2022), 1–34.
[17] Jesper Öqvist, Görel Hedin, Niklas Fors, and Christoff Bürger. 2023. Lund University compilers EDAN65 2023-09-05 - fileadmin.cs.lth.se. (2023). https://fileadmin.cs.lth.se/cs/Education/EDAN65/2023/assignments/A2.pdf