Contents

1 Compiler & Runtime System

Runtime System A Runtime System supports the application execution using software and hardware mechanism.

Architecture

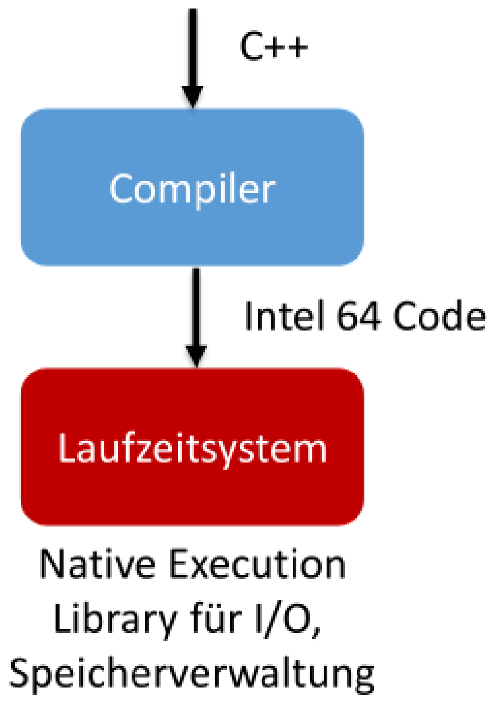


Figure 1: C++ Architecture

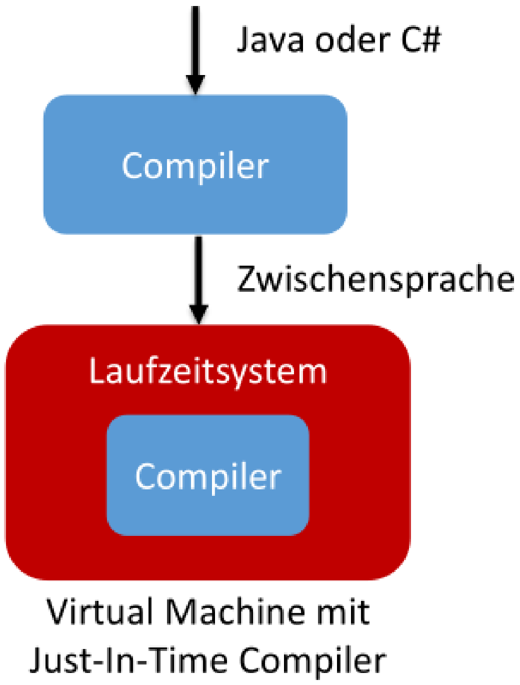


Figure 2: Java / C Architecture

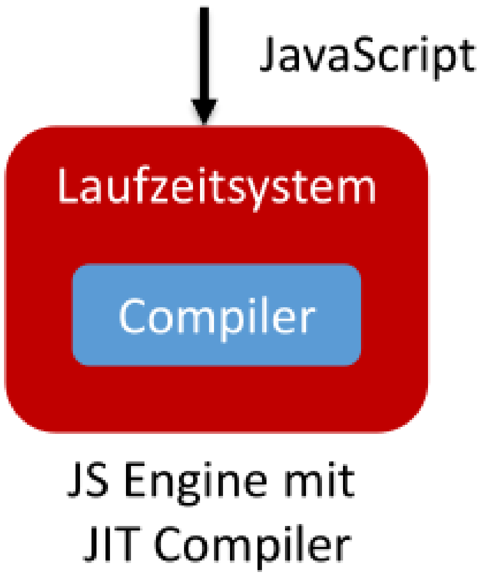


Figure 3: JS Architecture

Compiler Architecture A compiler consists of the following parts:

- Lexer
- Parser
- Semantic Checker
- Optimization
- Code Generator

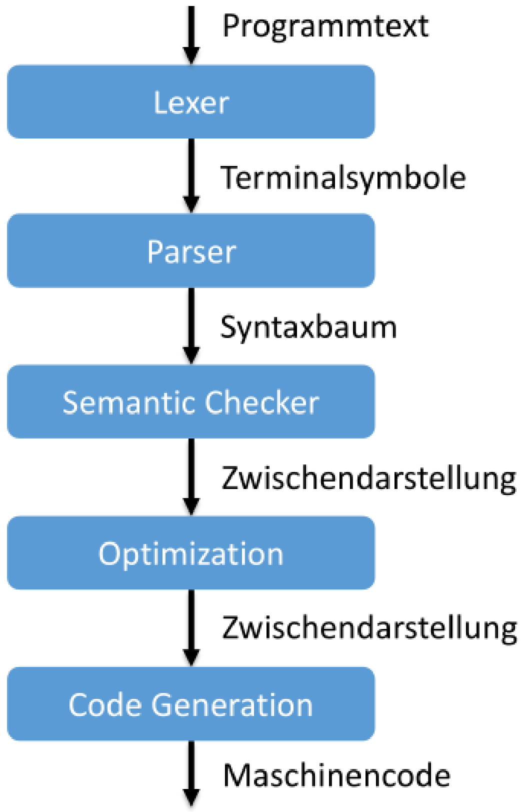


Figure 4: General Compiler Architecture

Runtime Architecture A Runtime System consists of the following parts:

- Loader
- Interpreter
- Metadata, Heap, Stack
- optional Garbage Collection
- optional Just-In-Time Compiler

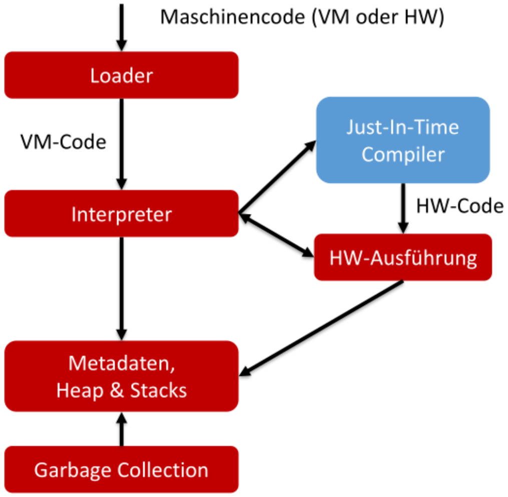


Figure 5: Components of a Runtime System

EBNF The syntax of a language can be expressed using the (E)BNF. The EBNF has some convenient features but can not express more than the BNF.

Syntax vs. Semantic The syntax defines the structure of the program while the semantic describes the meaning behind the syntax. The syntax is often described using the EBNF. The semantic is often described in prose.

2 Lexer

Lexer A Lexer takes a string of chars as input (source code) and produces a stream of terminal symbols (tokens). You want to use a lexer because it will help to parse the input.

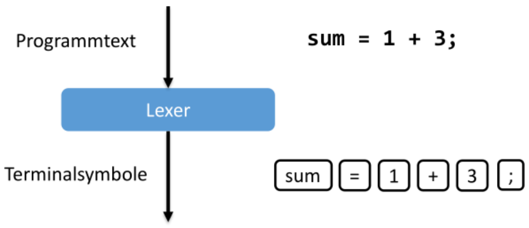


Figure 6: Input and output from a Lexer

A lexer does support only Regular Language. That means, languages which can expressed using EBNF without recursion. Therefore, a look ahead of one char is enough to implement a lexer.

```
Integer = Digit { Digit }. // regular
Digit = "0" | ... | "9".

Ausdruck = [ "(" Ausdruck ")" ]. // not regular
```

Chomsky Hierarchy The Chomsky Hierarchy describes what is required to read and understand a specific language. Lexer, Parser and Semantic Checker reads the Language while the DEA, Push down Automata and Bounded Turing Machine runs the application.

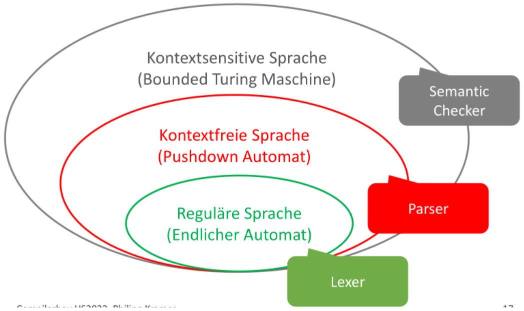


Figure 7: Chomsky Hierarchy

3 Parser

Parser A Parser takes the output of a Lexer (token stream) and checks if the input follows the given syntax. A parser can read only context free languages (see 2).

The output from a parser is normally an Abstract Syntax Tree or Concrete Syntax Tree. If you write the parser by your self you normally want to create an AST.

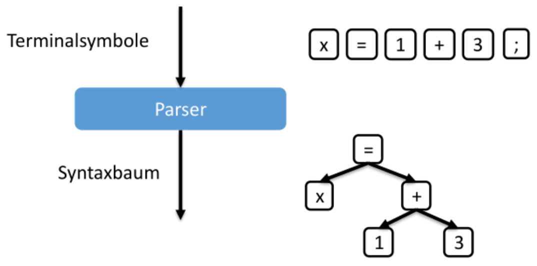


Figure 8: In- and Output for Parser

Pare Tree A parse tree a.k.a. Concrete Syntax Tree is a derivation of the syntax rules as tree. A Concrete Syntax Tree contains everything, even the not required parentheses. The order of the evaluation is implicitly given through the data structure.

The Abstract Syntax Tree is a more minimal version of the Concrete Syntax Tree.

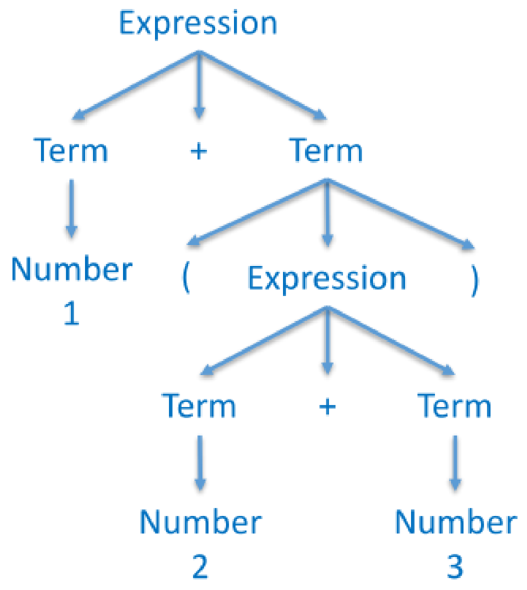
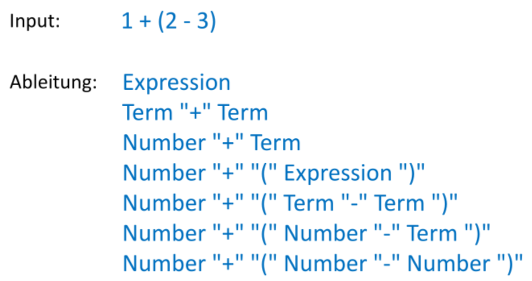


Figure 9: Concrete Syntax Tree

AST The Abstract Syntax Tree is a possible output from the Parser. The Abstract Syntax Tree contains only the required information. For example parentheses are not stored for a Java AST.

Top-Down Parsing A Top-Down Parser will start with the Start Symbol form the EBNF and performs production on the input. The Top-Down parser will expand from the start symbol to the terminal symbol.

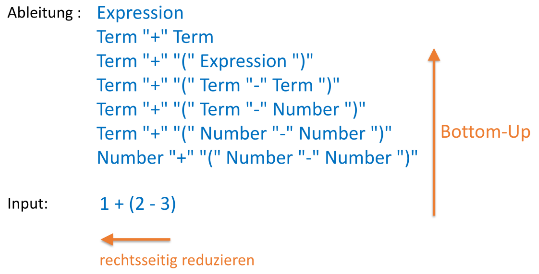


Input: 1 + (2 - 3)

Ableitung: Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"

Figure 10: Top-Down Parsing

Bottom-Up Parsing A Bottom-Up parser starts with the input stream (tokens). As soon as the parser can match the input on a rule it will reduce it.



Ableitung : Expression
Term "+" Term
Term "+" "(" Expression ")"
Term "+" "(" Term "-" Term ")"
Term "+" "(" Term "-" Number ")"
Term "+" "(" Number "-" Number ")"
Number "+" "(" Number "-" Number ")"

Input: 1 + (2 - 3)

Figure 11: Bottom-Up Parsing

Recursive Descent Parser A recursive descent parser is a Top-Down Parser. For each non-terminal symbol in a EBNF you will implement a function. During parsing, it will call the function `parseExpression` which will call `parseTerm`. The `parseTerm` function will call again `parseExpression` (recursion).

The Recursive Descent Parser is a Push-Down Automata which uses the call stack as its stack.

```
Expression = Term { ( "+" | "-" ) Term }.
Term = Number | "(" Expression ")".
```

```
void parseExpression() {
    parseTerm();
    while (is(Tag.PLUS) || is(Tag.MINUS)
        ) {
        next();
        parseTerm();
    }
}

void parseTerm() {
    // ...
    parseExpression();
    // ...
}
```

Parser Categories

- first letter

  L read from left to right
  R read from right to left

- second letter

  L Left-most expansion (top-down parser)
  R right-most reduction (bottom-up parser)

- number in parentheses
  - number of symbols look ahead

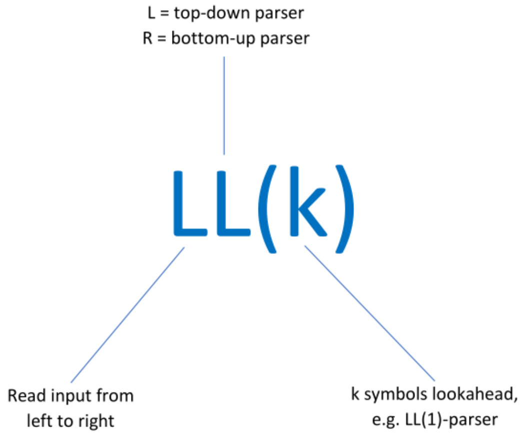

L = top-down parser
R = bottom-up parser
LL(k)
Read input from left to right
k symbols lookahead, e.g. LL(1)-parser

Figure 12: Parser notation

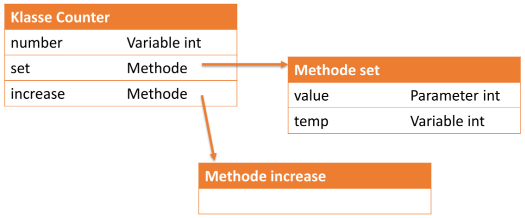
4 Semantic Checker

Semantic Checker A Semantic Checker takes the syntax tree (AST / Concrete Syntax Tree) from the Parser and returns the syntax tree with a Symbol Table. The semantic checker performs context-sensitive checks as:

declaration is every identifier uniquely declared
types are the type rules satisfied
methoden calls are arguments and parameters compatible
and more (no cyclic inheritance, single main, ...)

To perform this checks the checker requires all declarations (variables, methods, classes) and all types (predefined, user defined, array, type polymorphism).

Symbol Table The Symbol Table is a data structure produced by the Semantic Checker. The Symbol table is used to manage all declaration in a program source code.
The symbol table contains a mapping from *Symbol* to the *AST* node (`node = symbolTable.getDeclarationNode(symbol);`)



Klasse Counter	
number	Variable int
set	Methode
increase	Methode

Methode set	
value	Parameter int
temp	Variable int

Methode increase

Figure 13: Example Structure for a Symbol Table

Symbol Table Construction The Semantic Checker performs the following steps:

1. construct the symbol table
   - traverse AST
   - start with global scope
   - insert symbol in parent scope
   - do not resolve type name and designator yet
   - do not forget Built-in values / types
2. resolve all types in table
   - set the type in symbol
   - see Figure 14
3. resolve all declarations in AST
   - see Figure 15
4. resolve types in AST

- set type for each expression
- see Figure 16

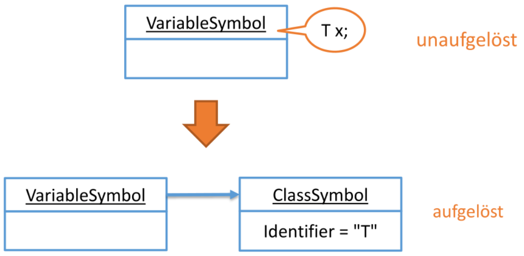


Figure 14: Resolve Types in Symbol

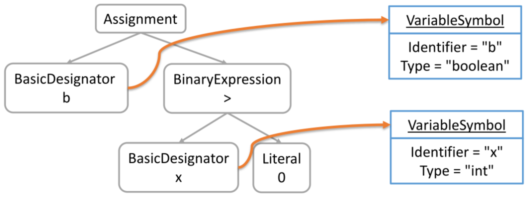


Figure 15: Resolve Declaration in AST

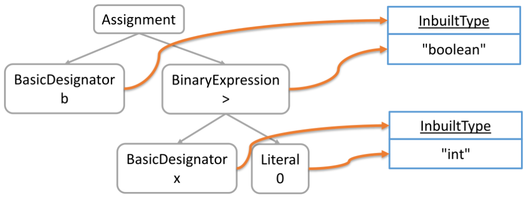


Figure 16: Resolve Types in AST

5 Code Generation

Code Generator The Code Generator takes the Symbol Table and the AST as input and generates machine code (Intel, ARM, JVM, ...). Normally you want to separate the Code Generator from the Parser, Semantic Checker. If you do so, you can use the same backend to produce code for various different platforms.

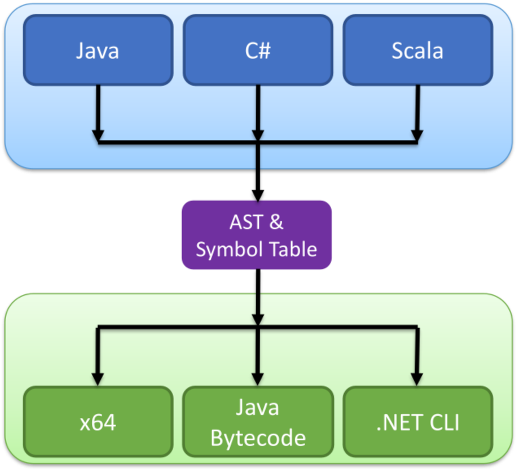
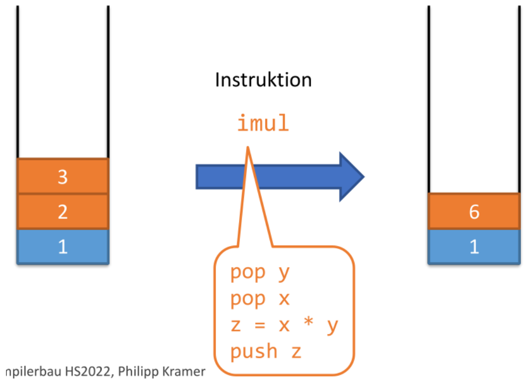


Figure 17: Seperate Backend / Frontend

Evaluation Stack In a VM (e.g. JVM - Java) you may use an evaluation stack to run interpret your instructions. The evaluation stack is an alternative to using registers (what you normally do on real processors). For example the `imul` instruction will:

1. pop y
2. pop x
3. perform z = x * y
4. push z

Each function call has its own evaluation stack and should be empty at the beginning and at the end.



npilerbau HS2022, Philipp Kramer

Figure 18: Example Evaluation Stack

6 Virtual Machine

VM A VM is a piece of software which emulates a processor. This processor can have a custom instruction set.
The benefit of a VM is:

- multiplatform
- multilanguage (multiple programming languages)
- security

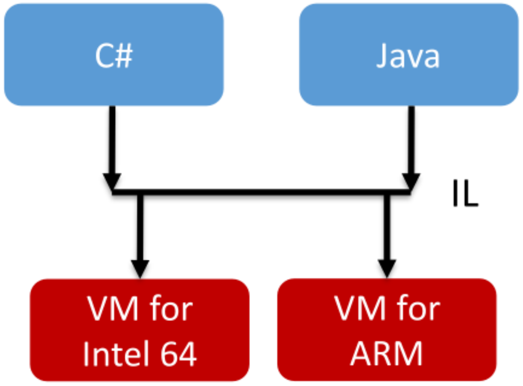


Figure 19: VM Example

Loader The loader

- reads the application binary into memory
- allocate memory
- defines memory layout
- perform address relocation
- starts program execution

Descriptor A descriptor contains the runtime information for types & methods:

- types: classes, arrays or base type
- classes: field types
- methods: parameter and local types, return value, byte code

Ancestor Table The ancestor table is used to check if a class inherits from another in constant time. In Figure 20 you see an example how such an ancestor table is constructed. The last element in the table for each class is the reference to itself. However, this only works for **Single-Inheritance**.

In languages like Java you don't have to allocate memory for the `object` class, because every class inherits from it (in the ancestor table this would be -1).

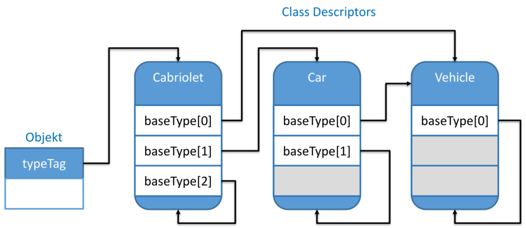


Figure 20: Ancestor Table Example

Type Cast / Checking You have a target type (you want to cast into this type) and a source type (you want to cast from this type).

1. Get the ancestor level from the target type (for example 0). Call the level **x**
2. Check in the ancestor table of the source type if the ancestor at level **x** is the same as the target class

```
private boolean typeTest(Pointer
    instance, ClassDescriptor targetType)
     {
    ClassDescriptor sourceType = heap.
        getDescriptor(instance);
    if (sourceType == targetType) {
        return true;
    }
    var level = targetClass.
        getAncestorLevel();

    if (sourceClass.getAncestorTable().
        length > level) {
        return sourceClass.
            getAncestorTable()[level] ==
            targetClass;
    } else {
        return false;
    }
}
```

Virtual Table Each type has a Virtual Method Table (vtable). The vtable describes which virtual method should be called when dynamic dispatch is performed. In a VM the vtable often points not directly to the function but to the method descriptor.

However, this only works with **Single-Inheritance**.

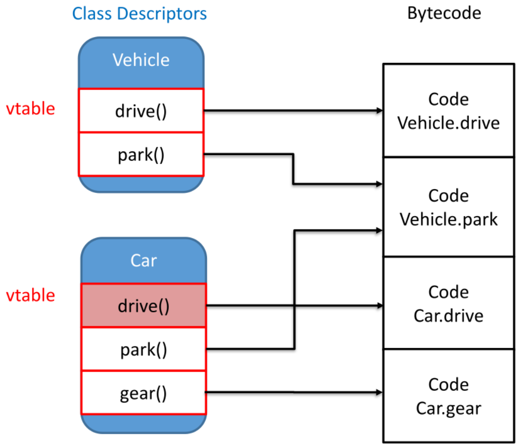
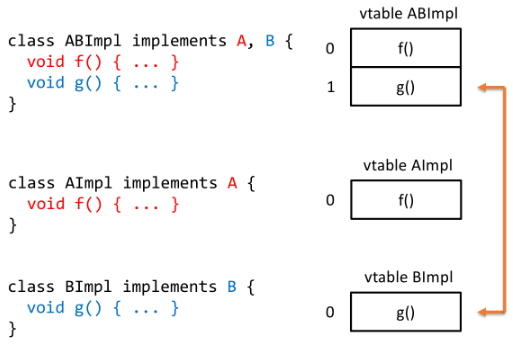


Figure 21: VTable example

Interface Support To implement dynamic polymorphism using Interfaces you can't use the same approach as using a vtable in inheritance. Because you would end up with inconsistent numbering (see Figure 22).



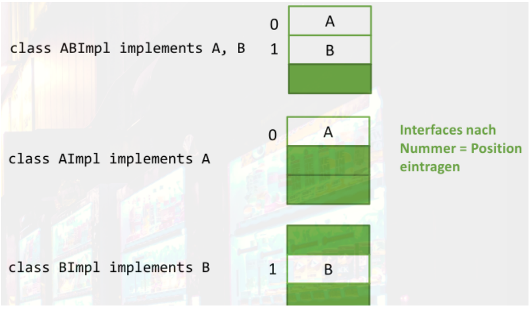
```
class ABImpl implements A, B {
  void f() { ... }
  void g() { ... }
}

class AImpl implements A {
  void f() { ... }
}

class BImpl implements B {
  void g() { ... }
}
```

Figure 22: Bad Interface Implementation Attempt

To support interfaces you have to globally numbering all interfaces. For each class you also have to generate an interface table (itable). This table contains only the implemented interfaces at the same index as they are globally numbered (see Figure 23).



class ABImpl implements A, B
0
A
1
B
class AImpl implements A
0
A
Interfaces nach Nummer = Position eintragen
class BImpl implements B
1
B

Figure 23: Interface Support

To safe memory you can generate a single itable with different offsets.

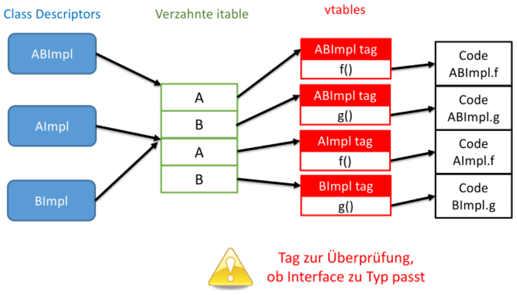


Figure 24: Total Interface Support

7 GC

Dangling Pointer A dangling pointer is a pointer which points to a location in memory which is already freed. Reading / Writing to such a pointer can / will cause problems (segfault).

Memory Leak We talk about memory leak if an object is not accessible anymore from the application but are not getting cleaned up. This is not as dangerous as a Dangling Pointer, but it will fill your memory with useless data.

GC The Garbage Collector is a piece of software which automatically cleans unused memory. The usage of a Garbage Collector is memory safe. That means:

- no Dangling Pointer
- no Memory Leak
- an overall simplification for programming

Reference Counting Reference counting is a mechanism to store the number of pointers to a reference. Every time a pointer starts to point to a resource the counter is incremented. As soon as one pointer does not point anymore to this location the counter is decremented.

RC Cyclic Graph As soon as you get a cyclic object structure the counter for each object can not reach zero. Therefore, the Garbage Collector will never recognize it as garbage and a Memory Leak occurs.

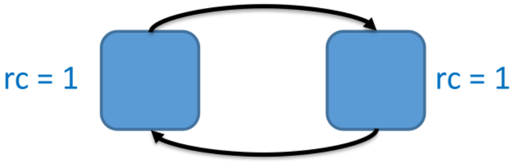


Figure 25: Cyclic Object Structure

Mark & Sweep The Mark And Sweep Algorithm can be used to inside a Garbage Collector to find and free garbage. In the first phase (mark phase) you perform a DFS staring from a root set. Every time you visit a node you mark it.
The root set consists of:

- all references in static variables
- all references in all activation frames on the call stack
- all references in the registers

In the second phase (sweep phase) you traverse the whole Heap linearly. If the node is marked you clear the mark and look at the next node. If the node is not marked you will free it.

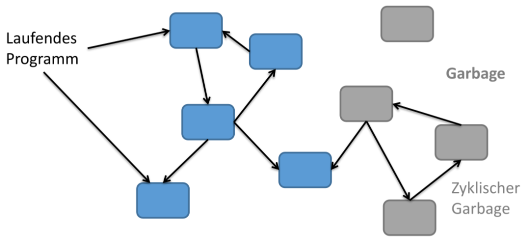


Figure 26: Heap Graph of Nodes

GC Execution The Garbage Collector is executed at the latest when the Heap is full. However, most common Garbage Collector are executed earlier.
Depending on the implementation the application can not run, while the Garbage Collector is working (Stop & Go).

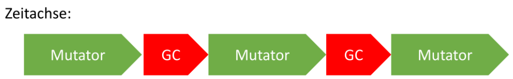


Figure 27: Stop Go GC

Finalizer A Finalizer is a method which is run just before the objects get deleted. These functions are normally called by the GC but not during normal GC phases.
Finalizers are executed in an arbitrary order and time. They run concurrent to the main program. In Java the finalizer is **NOT** executed again after Resurrection.

```
class Block {
    @Override
    protected void finalize() {
        //...
    }
}
```

Finalizer Execution: Time A finalizer is called from the GC but not during the normal GC phases. Reasons for this are:

- finalizer might run forever => would block GC
- finalizer migth create new objects => corrupts GC
- programming errors in finalizer => GC might crash
- finalizer can revive objects => Resurrection

Finalizer Execution: How The GC has a **finalizer set** where all finalizers are registered. In the **pending queue** are all finalizers which are not executed yet but has to. **Attention**: as soon as you insert the finalizer into the pending queue the object the revived (Resurrection). An additional GC phase is required.

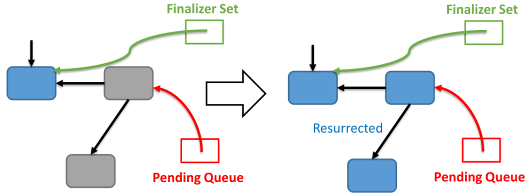


Figure 28: Pending Queue

Resurrection A finalizer can revive objects (no garbage anymore). This can happen not only for the current object but also for other objects.

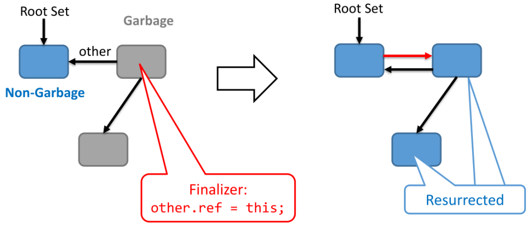


Figure 29: Resurrection Example

Weak References A Weak Reference are references which are not counted for Reference Counting. In a GC context this can be used for object caches. Weak References must be set to `NULL` after the target object is freed (Dangling Pointer).

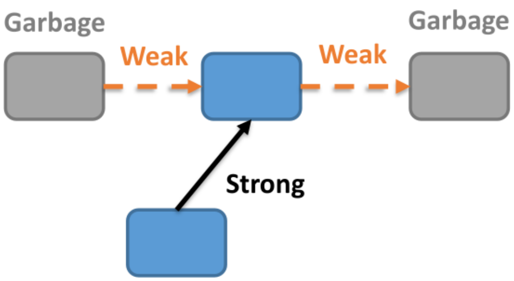


Figure 30: Weak Reference Example

Compacting GC A compacting GC always allocates at the end of the Heap right after the last object. During the GC Phases (here mark and copy) the marked objects (non-garbage objects) are copied to the beginning.

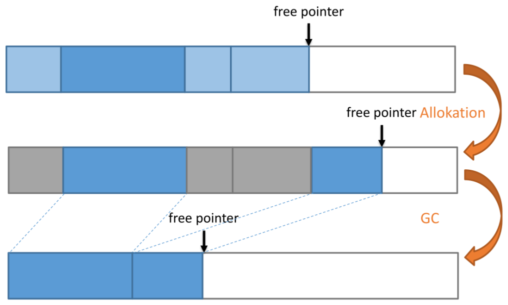


Figure 31: Compacting GC Example

Incremental GC An incremental GC runs **quasi parallel** to the mutator. The main execution is interrupted only a very short time (No Stop & Go GC).

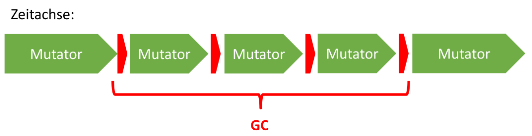


Figure 32: Incremental GC

Generational GC A Generational GC is an incremental GC because it often does not clean up the whole heap but in generations. For example, you have 3 generations as describe in Table 1

age	generation	GC frequency	GC break
new	G0	high	short
middle	G1	middle	middle
old	G2	low	long

Table 1: Example for generations

If you want to clean up *Gn* you must extend your root set with all references from the latter generations which are pointing into *Gn*. If you want to clean up *Gn* you also have to clean up all previous generations.

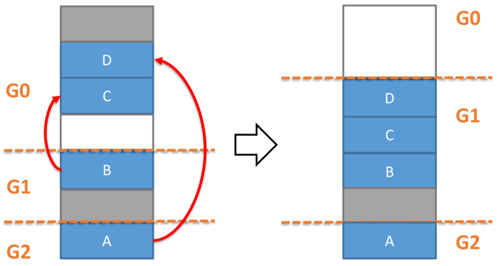


Figure 33: GC of G0

Partitioned GC A Partitioned GC is an incremental GC. The Heap is partitioned. The GC will focus on the partition with the most garbage. The non-garbage objects are moved to a new partition before cleaning up the partition (see Figure 34).

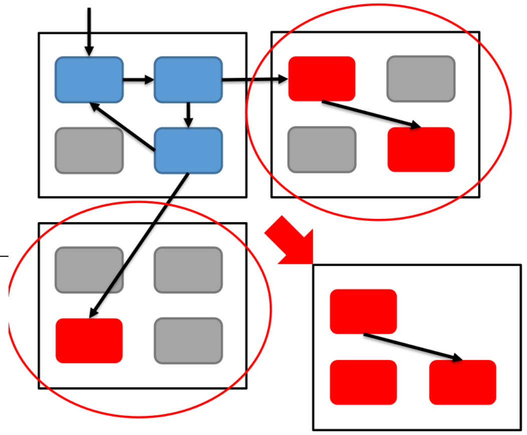


Figure 34: Partitioned GC

8 JIT

JIT A Just-in-Time compiler is used in interpreted languages to compile the intermediate code (e.g. java byte code) to native machine code. This process speeds up the execution. Because JIT compiling takes time you normally don't want to compile everything but only so-called **Hot Spots**. Hot Spots are code sections which are executed more often.

Registers In the original registers you have to specify another name. For detail information see Figure 35.

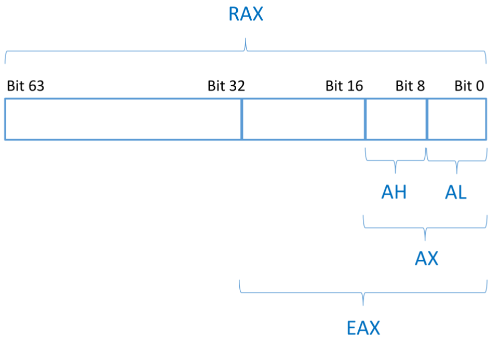


Figure 35: Register sub access

Registers Management The JIT Compiler keeps track of used registers in a Stack (similar to the Evaluation Stack). The stack is updated for each translated byte code instruction.

Register Clobbering For example the *IDVI* assembler instruction overrides the values in *RAX* and *RDX* as side effects. This is called register clobbering. Therefore, you have to save thoses values before you executed *IDIV* (Register Relocation).

9 Code Optimization

Optimizer The Optimizer is a part of the compiler. The optimizer transforms an Intermediate Representation or machine code with the goal to generate a more efficient version. This is often done using multiple steps. Sometimes the same step is performed multiple times.

Strategies

- arithmetic optimization
  - replace `*`, `/`, `%` with bit operations when possible
- algebraic simplification
  - replace const expression with const value
  - `expr / 1 -> expr`
  - `expr * 0 -> ldc 0`
  - `1 + 3 -> ldc 4`
- loop invariant code
  - calculate unchanged expressions during loop outside the loop (see Listing 1 and Listing 2)
- common sub-expressions
  - calculate same sub-expression only once (var are not allowed to change in between) (see Listing 3)
- dead code
  - eliminate dead code (May has to be performed multiple times to eliminate all dead code)
- redundant read and write
  - this may lead to dead code elimination
- constant propagation / constant folding
  - some variables are constant even after some long operations
  - load constant, may lead to dead code (in Listing 4 `ldc 3` has to be performed)
- partial redundancy
  - some expressions are evaluated one or more times (depending on the branching)
  - evaluate only once

```
while (x < N * M) { // N * M does never
   change during loop
  k = y * M;          // k does not change
      during loop
  x = x + k;
}
```

Listing 1: Loop invariant

```
k = y * M;
temp = N * M;
while (x < temp) {
  x = x + k;
}
```

Listing 2: Optimizied Loop

```
x = a * b + c;
// operations, but no changes on a and b
y = a * b + d;

temp = a * b;
x = temp + c;
// operations, but no changes on a and b
y = temp + d;
```

Listing 3: Common sub-expression

```
a = 1;
if (...) {
    a = a + 1; // a = 2
    b = a;     // b = 2
} else {
    b = 2;
}
c = b + 1;    // c = 2
```

Listing 4: Constant Propagation

SSA Static Single Assignment is a method to detect optimization possibilities. In SSA each variable is assigned only once. If a variable is assigned multiple times you rename it. Assignments in different branches also get different indexes. To find the right index at the Phi Function is used.

```
if (...) {
    x_1 = 1;
} else {
    x_2 = 2;
}
y_1 = x_?;
```

Listing 5: SSA with branches

Phi Function The Phi Function is used to check which `x` should be used during SSA after branching. In Listing 6 the phi function performs the following:

- return `x_1` if the `if` branch was taken
- return `x_2` if the `else` branch was taken

```
if (...) {
    x_1 = 1;
} else {
    x_2 = 2;
}
y_1 = phi(x_1, x_2);
```

Listing 6: Phi-Function

SSA and Common Subexpression Using SSA you can detect multiple assignments to the same variable (or the absence of them). Thanks to this, you can detect common subexpressions.

```
x_1 = a_1 * b_1 + c_1;
// some operations
y_1 = a_1 * b_1 + d_1; // no changes on
    a and b otherwise index would be
    incremented
// some operations
z = a_1 * b_2 + d_3;
```

Listing 7: Common sub-expressions in SSA

SSA and Dead Code Thanks to SSA you can detect that `x_1` is never read. Therefore, you don't have to generate code for this expression.

```
x_1 = 1;
x_2 = 2;
y_1 = x_2 + 1;
writeInt(y_1);
```

Listing 8: Dead code detection

Peephole Optimization Peephole Optimization is used only on a small set of instruction. This set of instruction is changed using a Sliding Window. The Optimizations are performed on the sliding window.

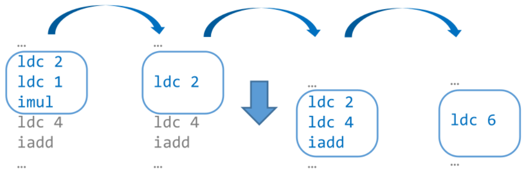


Figure 36: Peephole Optimization Example

10 END