AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN AARHUS UNIVERSITET

ITTAPK-01

Little Virtual Machine

Studienr.	**Navn**	**E-mail**
20105969	Kalle Rønlev Møller	km@post.au.dk
201370050	Kasper Sejer Kristensen	kasper.sejer.kristensen@post.au.dk
201370321	Kristian Mosegaard	kristian.mosegaard@post.au.dk

Indholdsfortegnelse

1 Indledning

Vores oplæg er at implementere en lille virtuel maskine, som kan eksekvere simpelt bytecode (en eksekverbar fil i vores eget format). Arkitekturen af vores maskine vil være stack baseret. Dette er en maskine, hvor alt er centreret omkring en stack (der er derfor ingen registre), hvor forskellige operationer kan mutere stakken. Processeren manipulerer stakken alt efter hvilken opcode (en instruktion), som er givet.

Den virtuelle maskiner vil blive opbygget af to delsystemer: en *compiler* og en *interpreter*. Compileren vil kunne tage en fil med opcodes og "kompilere"denne til bytecode. Bytecode filen kan så køres igennem vores interpreter, som vil udføre programmet i dets helhed.

Alt dette vil implementeres med nogle af de koncepter som er blevet formidlet gennem kurset IT-TAPK.

2 Systembeskrivelse

I dette afsnit vil vi beskrive grundlæggende tanker omkring opbygningen af både interpreteren og compileren.

2.1 Interpreter

Interpreteren vil blive implementeret som en stack maskine. En stack maskine virker ved omvendt polsk notation. Dvs. for en plus instruction ser handlingsforløbet således ud: A B + betyder A + B. Dvs. først skubbes de to værdier vi ønsker at lægge sammen på en stack, derefter udføres en plus operation på de to værdier.

Interpreteren skal virke vha.

- et program område, hvor alle de forskellige opcodes findes.
- en stack, hvor værdierne der arbejdes på, opholder sig.

2.2 Opcodes

Den virtuelle maskine skal kunne afvikle de opcodes, som er specifieret i Tabel 2.1.

Name	**Code**	**Description**	**Before**		**After**	
			Stack	*Ptr*	*Stack*	*Ptr*
`In`	0x00	Read number from stdin	$(...)$	0x00	$(...)(ch)$	0x01
`Out`	0x01	Write char to stdout	$(...)(ch)$	0x00	$(...)$	0x01
`Add`	0x02	Addition	$(...)(b)(a)$	0x00	$(...)(b+a)$	0x01
`Sub`	0x03	Subtraction	$(...)(b)(a)$	0x00	$(...)(b-a)$	0x01
`Mul`	0x04	Multiplication	$(...)(b)(a)$	0x00	$(...)(b \cdot a)$	0x01
`Div`	0x05	Integer Division	$(...)(b)(a)$	0x00	$(...)(b/a)$	0x01
`Mod`	0x06	Modulo Division	$(...)(b)(a)$	0x00	$(...)(b\%a)$	0x01
`Neg`	0x07	Negation	$(...)(a)$	0x00	$(...)(-a)$	0x01
`Inc`	0x08	Increment	$(...)(a)$	0x00	$(...)(a+1)$	0x01

Name	**Code**	**Description**	**Before**		**After**	
			Stack	*Ptr*	*Stack*	*Ptr*
`Dec`	0x09	Decrement	(...)(a)	0x00	(...)($a-1$)	0x01
`And`	0x0A	Bitwise And	(...)(b)(a)	0x00	(...)($b \wedge a$)	0x01
`Or`	0x0B	Bitwise Or	(...)(b)(a)	0x00	(...)($b \vee a$)	0x01
`Not`	0x0C	Bitwise Negation	(...)(a)	0x00	(...)($\sim a$)	0x01
`Xor`	0x0D	Bitwise Exclusive Or	(...)(b)(a)	0x00	(...)($b \oplus a$)	0x01
`Shl`	0x0E	Left Shift	(...)(b)(a)	0x00	(...)($b \ll a$)	0x01
`Rhl`	0x0F	Right Shift	(...)(b)(a)	0x00	(...)($b \gg a$)	0x01
`Push`	0x10	Push	(...)	0x00	(...)(*0x01)	0x02
`Pop`	0x11	Pop	(...)(a)	0x00	(...)	0x01
`Dup`	0x12	Duplicate	(...)(a)	0x00	(...)(a)(a)	0x01
`Swp`	0x13	Swap	(...)(b)(a)	0x00	(...)(a)(b)	0x01
`Ovr`	0x14	Duplicate Over	(...)(b)(a)	0x00	(...)(b)(a)(b)	0x01
`Load`	0x15	Load from address	(...)(addr)	0x00	(...)(*addr)	0x01
`Stor`	0x16	Store to address	(...)(data)(addr)	0x00	(...)	0x01
`Jmp`	0x17	Jump to address	(...)(addr)	0x00	(...)	addr
`Je`	0x18	Jump If Equal	(...)(b)(a)(addr)	0x00	(...)	0x01 or addr if $a == b$
`Jne`	0x19	Jump If Not Equal	(...)(b)(a)(addr)	0x00	(...)	0x01 or addr if $a! = b$
`Jg`	0x1A	Jump If Greater	(...)(b)(a)(addr)	0x00	(...)	0x01 or addr if $a > b$
`Jge`	0x1B	Jump If Greater Or Equal	(...)(b)(a)(addr)	0x00	(...)	0x01 or addr if $a >= b$
`Jl`	0x1C	Jump If Less	(...)(b)(a)(addr)	0x00	(...)	0x01 or addr if $a < b$
`Jle`	0x1D	Jump If Less Or Equal	(...)(b)(a)(addr)	0x00	(...)	0x01 or addr if $a <= b$
`Nop`	0x1E	Do nothing	(...)	0x00	(...)	0x01
`Halt`	0x1F	Terminate Program	(...)	0x00	(...)	0x01

Tabel 2.1: Opcodes

2.3 Compiler

Compileren er ansvarlig for at tage et program (skrevet af en bruger) og oversætte dette til byte-code, som interpreteren kan afvikle. Derfor er det nødvendigt at sætte nogle regler op for, hvordan et program skal se ud.

Følgende regler skal være opfyldt:

- Opcodes står på en linje for sig, der må kun anvendes en opcode pr. linje.
- Semikolon (;) betyder, at alt efter semikolonet er en kommentar.
- Kolon (:) er en label deklaration som bruges i forbindelse med jumps.
- Snabel A (@) er en label reference som indsætter det som label referere til.
- En tal værdi skrives blot i programmet på en linje for sig selv.

Programmer

I Listing 2.1 er et eksempel på et program.

```
IN   ; read integer "A" from standard input
IN   ; read integer "B" from standard input

:GCD ; this is a label
DUP  ; if B is 0 then A is the gcd
0    ; (immediate values get pushed on the stack)
@END ; (this is how you put the address of a label onto the stack)
JE   ; (this will jump to the address at top of stack if the preceding two values are equal)
SWP  ; if B is not 0 then the result is gcd(B, A modulo B)
OVR
MOD
@GCD
JMP  ; recursion!

:END
POP  ; remove 0 from top of stack
OUT  ; now the result is at the top, print it.
```

Listing 2.1: Eksempel på program

Bytecode

Når programmet er kompileret, er det blevet til bytecode. I Listing 2.2 er der et eksempel på et program, der lægger to tal sammen.

```
00000000  12 00 00 00 30 00 00 00  12 00 00 00 01 00 00 00  |....0...........|
          PUSH        '0'          PUSH        1

00000004  02 00 00 00 01 00 00 00  20 00 00 00              |............|
          ADD         OUT          HALT
```

Listing 2.2: Eksempel på indholdet af den eksekverbare fil

Læg mærke til at byte nr. `0x03` og byte nr. `0x05` har den samme værdi. Så hvis programpointeren rammer `0x03` så vil den antage at det er en OUT kommando.

3 Design

I dette afsnit vil vi kommentere på design valg i henholdsvis interpreter og compileren.

3.1 Interpreter

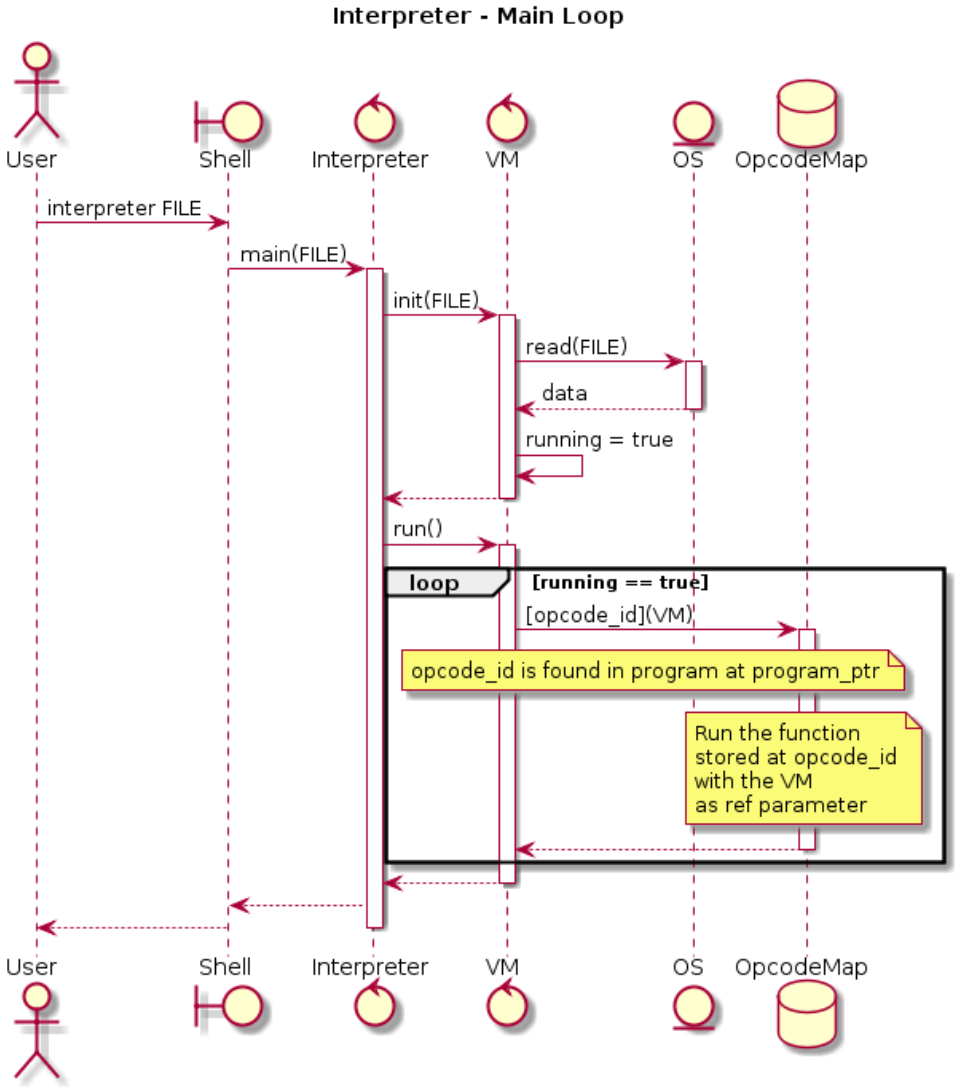
Interpreteren er i sig selv ret simpelt bygget op. Den indeholder et VMSystem.

VMSystem er opdelt i:

- Program (array, vector, list)
- Memory (array, vector, list)
- Stack (array, stack)
- OpcodeMap (array, vector)

Idéen er at Program, Memory, Stack er template parametre. Derved kan man selv vælge om man ønsker en statisk stack (array) eller en dynamisk stack (stack). Compileringen sørger selv for at bruge de mest optimale funktioner til den givne stack. Det samme er gældende for Program og Memory. Interpreteren kan derfor også kompileres til forskellige type størrelser int64, int32 osv.

VMSystemets main funktion er at læse opcodes fra et program, slå op i et opcode map og køre den associeret funktion. Funktionen er ansvarlig for at flytte program-pointeren. Denne proces gentages indtil halt funktionen bliver kaldt. I Figur 3.1 ses der et sekvensdiagram, som afbilleder forløbet.



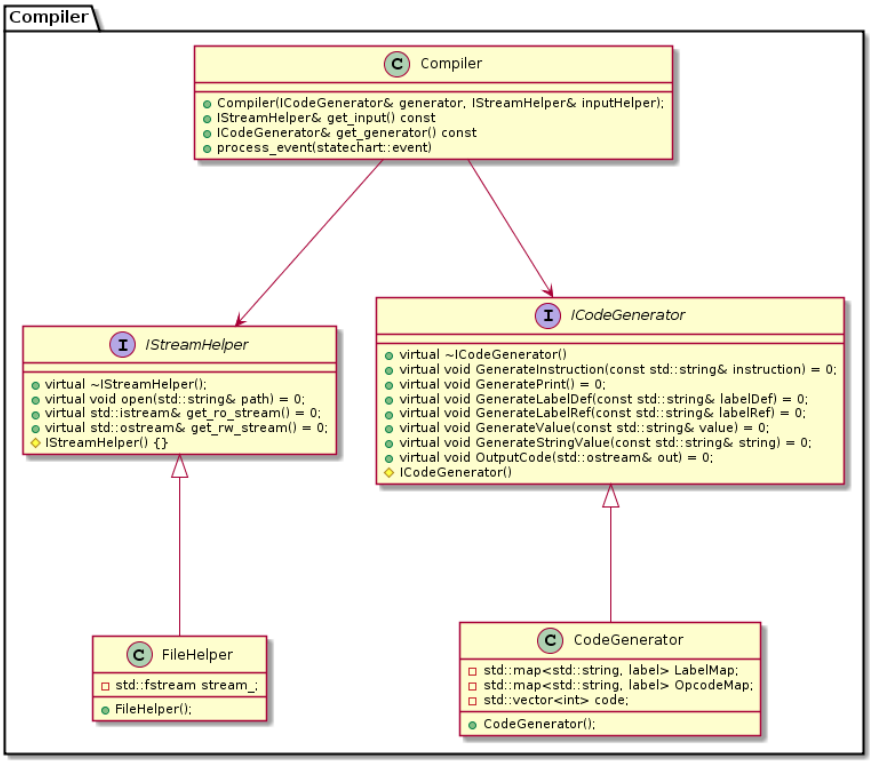
Figur 3.1: Sketch af Interpreter Design

3.2 Compiler

Compileren består af tre overordnede klasser: en til filhåndtering, en til kode generering også selve compilerens state machine.

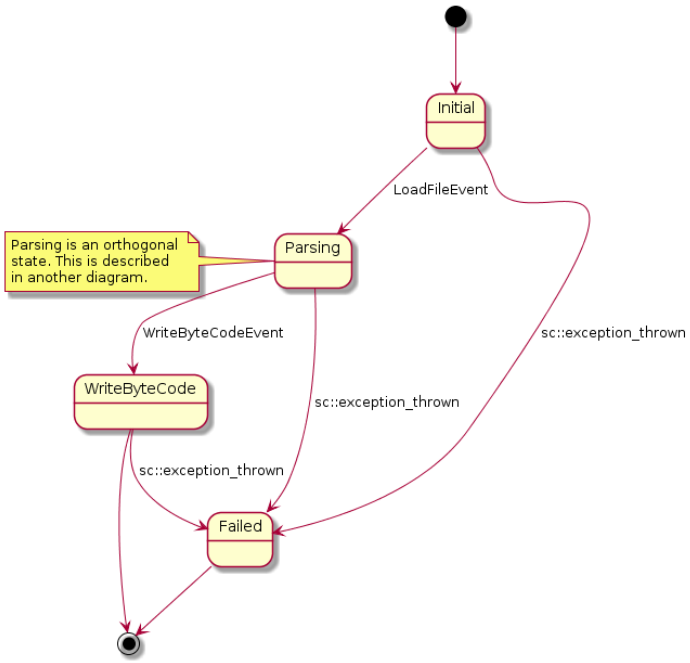
- Compileren state machine sørger for at parse program filen, og kalder kodegeneratoren, hvis den møder instruktioner eller værdier i filen.
- Kodegeneratoren skriver de rette værdier til den binære fil ved hjælp af et opcode map, som mapper binære instruktionsværdier med opcodes.
- Filhåndteringen står for at åbne/lukke fil streams.

Et klassediagram over compileren kan ses i Figur 3.1.



Figur 3.1: Klasse diagram for Compiler

Da compileren implementeres som en state machine følger der et overordnet diagram med, som kan ses i Figur 3.2.

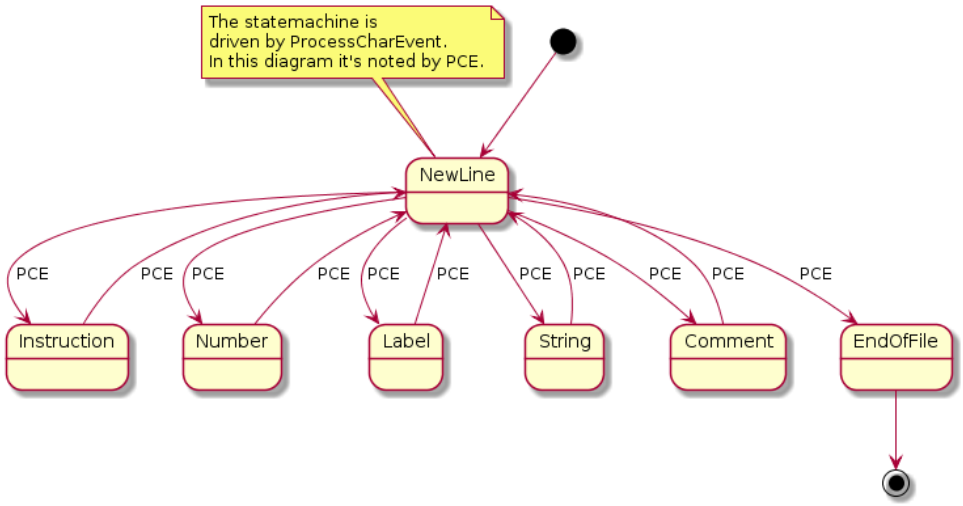


Figur 3.2: State machine diagram for Compiler

I Figur 3.2 ses der fire states. Disse states har følgende ansvar:

- **Initial**: Håndtere opsætning af fil stream.
- **Parsing**: Håndtere fortolkning af fil.
- **WriteByteCode**: Håndtere skrivning af eksekverbar fil.
- **Failed**: Lave fejlhåndtering.

Parsing er en orthogonal state. Heri er en sideløbende state machine, som aggere alt efter hvilken karakter, som bliver læst ind fra filen. Denne kan ses Figur 3.3.



Figur 3.3: State machine for Compiler

4 Implementering

I dette afsnit beskrives implementeringen af interpreteren og compileren.

4.1 Interpreter

Interpreteren lever i namespacet `lvm::interpreter`.

VMSystem

VMSystem struct'en er en template, hvor brugeren selv kan vælge hvilken collection skal holde Program, Stack samt memory.

```
using STACK       = std::array<int32_t,10000>;
using MEMORY      = std::array<int32_t, 10000>;
using PROGRAM     = std::array<int32_t, 10000>;
using OPCODEMAP   = lvm::interpreter::vector<A...>;
using OPCODESET   = lvm::interpreter::oc_11;
using CIN         = std::istream;
using COUT        = std::ostream;
using vmsystem    = lvm::interpreter::vmsystem<lvm::interpreter::vector, OPCODESET, STACK, MEMORY, PROGRAM, CIN, COUT
    >;

int main(int argn, char* argc[]) {
```

```
    auto vm = vmsystem(std::cin, std::cout);
    #INIT VM
    vm.run();
    return vm.exit_code();
}
```

Main loopet hiver værdien fra Program på ProgramPointerens position. Denne værdi bliver så brugt i OpcodeMappet til at finde den rigtige Opcode. Opcoden bliver derefter kaldt med VMSystemet som argument. Dette fortsætter indtil Running er False. Interpreteren afslutter derefter og dens returkode er det der var på toppen af stacken.

- ProgramPointer
- StackPointer (kun hvis stack er array)
- Running (Status om VM kører)

Når interpreteren starter sker følgende.

- Binærefil loaded ind i Program
- ProgramPointer = 0
- StackPointer = 0
- Running = True

StackPointer

Da `std::stack` har både push og pop, er der ingen grund til at have en stackpointer - medmindre stacken er realiseret som et array. Hvis stacken er realiseret som et array, skal systemet selv holde øje med en stackpointer. Dette kunne klares ved altid at have en stackpointer, og så bare ikke bruge den, hvis systemet ikke er lavet med arrays. Dette er jo selvfølgelig spild af plads! Derfor løses det ved at lade VMSystem arve fra VMStack.

```
template < class T >
class has_push_impl
{
    struct Fallback { int push; };
    struct Derived : T, Fallback { };

    template<typename U, U> struct Check;

    typedef char InvalidMember[1];
    typedef char ValidMember[2];

    template<typename U> static InvalidMember & func(Check<int Fallback::*, &U::push> *);
    template<typename U> static ValidMember & func(...);
  public:
    static constexpr bool value = (sizeof(func<Derived>(0)) == 2);
};

template < class T >                                                    \
struct has_push : public std::integral_constant<bool, has_push_impl<T>::value> {};

template <typename STACK, bool> struct VMStack : STACK {
    void init(){};
};

template <typename STACK>
struct VMStack<STACK, false > : STACK {
    typename STACK::value_type ptr = 0;
```

```
    # CODE
};

template<template<typename...> class OPMAP, typename OPCODESET, typename STACK, typename MEMORY, typename PROGRAM,
     typename CIN, typename COUT>
struct vmsystem : VMStack<STACK, has_push<STACK>::value>,
                  VMMemory<MEMORY, has_resize<MEMORY>::value>,
                  VMProgram<PROGRAM, has_resize<PROGRAM>::value> {
    # CODE
}
```

Da templaten tager højde for om STACK'en er et `std::array`, bliver stack_ptr'en kun oprettet i det ene tilfælde. Fordelen ses ved at compilering vil fejle, hvis der forsøges at læse eller skrive til stack_ptr, når en std::stack er brugt.

Opcode Function

Opcodes er dem der flytter rundt på data mellem stack, memory, og program samt opdatere program_ptr og stack_ptr. Da vi ikke ved om der eksisterer en stack_ptr og da nogle operationer kan gøres mere effektiv med de forskellige collections, er vi nødt til at have forskellige implementationer af de enkelte opcodes. Det kan ses nedenfor at swap kan gøres noget hurtigere hvis stacken er et `std::array`, da vi har en index operator. Hvis stacken derimod kun har push og pop, skal der laves en dele mere arbejde. Her bliver der brugt sfinae for at finde ud af hvilken funktion der reelt skal bruges.

```
struct Swap : shared::opcodes::Swap {

    template<typename VM, typename std::enable_if<has_pop<typename VM::stack_type>::value, int>::type = 0>
    static volatile void execute(VM& vm) {
        auto a = vm.stack.top();
        vm.stack.pop();
        auto b = vm.stack.top();
        vm.stack.pop();
        vm.stack.push(a);
        vm.stack.push(b);
        ++vm.program_ptr;
    };

    template<typename VM, typename std::enable_if<!has_pop<typename VM::stack_type>::value, int>::type = 0>
    static volatile void execute(VM& vm) {
        auto a = vm.stack[vm.stack_ptr];
        vm.stack[vm.stack_ptr] = vm.stack[vm.stack_ptr-1];
        vm.stack[vm.stack_ptr-1] = a;
        ++vm.program_ptr;
    };

};
```

4.2 Compiler

Compileren lever i namespace `lvm::compiler`. Compileren er opbygget som en state machine, der gør brug af `boost::statechart`. Derudover er der også anvendt exceptions til fejlhåndtering, alt efter om der sker en fejl med dekodningen eller fil streamet. STL bibliotektet er blev brugt til at gemme byte code og opcodes i hukommelsen mens de blev bearbejdet. Der er også brugt typelists til at constraine hvilke op koder der er tilgængelige.

CodeGenerator

I kode generatoren er der brugt `std::map` og `std::vector` til at gemme opcodes og byte coden i henholsvis map for opcodes og vector for bytekoden. Mappet er brugt til at oversætte den mnemonic'ske

repræsentation til byte code.

```
    code.push_back(OpcodeMap["PUSH"]);
    code.push_back(0);
    std::for_each(stringy.rbegin(), stringy.rend(), [this](std::vector<int>::reference x){
        code.push_back(OpcodeMap["PUSH"]);
        code.push_back(x);
    });
```

Listing 4.1: Brug af std::algorithms

Som det kan ses i overstående kodelisting er der også brugt algorithmer i dette tilfælde `std::for_each` sammen med et lambda udtryk gør det let at flytte bytekoden i ovendt række følge fra en vector til vores code vector.

Compiler STM

Der gøres brug af `boost::statechart` til at facilitere den statemachine, som compileren er designet ud fra. Denne state machine kan ses i Figur 3.2.

```
struct Compiler : sc::state_machine<Compiler, Initial, std::allocator<void>, sc::exception_translator<>> {
  Compiler(ICodeGenerator& generator, IStreamHelper& inputHelper) : generator_(generator), input_(inputHelper) {}

  IStreamHelper& get_input() const { return input_; }

  ICodeGenerator& get_generator() const { return generator_; }

private:
  ICodeGenerator& generator_;
  IStreamHelper& input_;
};

struct Initial : sc::simple_state<Initial, Compiler> {
  typedef mpl::list<sc::custom_reaction<sc::exception_thrown>, sc::custom_reaction<LoadFileEvent>> reactions;

  sc::result react(const LoadFileEvent& ev) {
        ...
  }

  sc::result react(const sc::exception_thrown&) {
    ...
  }
};
```

Listing 4.2: Brug af `boost::statechart` i Compiler STM

I Listing 4.2 kan deklareringen af compilerens state machine ses. De forskellige states bliver deklareret i deres egne structs dvs. at f.eks. har staten *Initial* sin egen struct. *Initial* kan ligeledes påvirkes af events, som i det bedste tilfælde bliver ført hen til *Parsing*. Hvis en fejl måtte opstå og der bliver smidt en exception, vil staten tage stilling til fejlhåndteringen på baggrund af typen af exception. I alle tilfælde vil state machinen ende i *Failed*.

Opcode TypeList

For at holde styr på hvilke opcodes der er supportert af den virtuelle maskine bruges en typelist. Alle opcoderne er definerede som classes der indeholder deres mnemonic og der bytekode værdi. Dette bruges til at lave det map codegeneratoren bruger når der skrives bytekode.

```
typedef MakeTypeList<opcodes::In, opcodes::Push, opcodes::Div, opcodes::Inc, opcodes::Dec, opcodes::Store, opcodes::
    Load, opcodes::JumpNotEqual, opcodes::Halt, opcodes::Out, opcodes::Add, opcodes::Sub, opcodes::Mul, opcodes::
    Duplicate, opcodes::JumpEqual, opcodes::Jump, opcodes::Pop, opcodes::Swap, opcodes::Mod, opcodes::CopyOver>::List
     SupportedOpcodes;
```

```
template <typename TL>
struct MakeOpcodeMap{

    static void run(std::map<std::string, const int> & innerMap){
        innerMap.insert(std::pair<std::string, const int32_t>(std::string(TL::First::name), (int32_t)TL::First::
             template code<int32_t>));
        MakeOpcodeMap<typename TL::Left>::run(innerMap);
    }
};

template <>
struct MakeOpcodeMap<NullType>{

    static void run(std::map<std::string, const int> innerMap)
    {
    }
};
```

Listing 4.3: Brug af TypeList i compiler

Sammen med typelisten bliver der brugt en template funktion der udfra en typeliste kan bygge det mappet. Som det kan ses ovenover.

5 Konklusion

Målet med opgaven har været at udforske hvordan en computer kan virke under kølerhjelmen og for at se om nogle af de advanceret C++ koncepter, som vi har lært igennem kurset kan føre til bedre kode.

Vi har igennem projektet fået et indblik i hvordan en stack baseret maskine virker. Vi har ikke formået at implementere alle de opcodes, som vi har beskrevet, men det vil være ret nemt at tilføje dem, da vi har brugt advanceret C++ features til implementationen.

Vores design af compileren er baseret på en state machine. Derved kunne vi gøre brug af boost's statechart, som giver os en 1–til–1 mapning mellem diagram og implementering, hvilket har gjort ændringer lette at implementere/fjerne.

Vi har kunne kombinere de stærke biblioteker i C++ (boost og STL) ind i vores implementeringer, som er med til at give god kode, der er relativ let at forstå og giver en smæklækker følelse af ekstra sikkerhed.

Især koncepter som lambda udtryk har været gode at bruge med standart algorithmerne, da funktionen står inplace, hvilket er med til at gøre intentioner lette at forstå.