AP Exam Report

By Simon Gustav Cordes

Introduction

This report presents the solutions, assumptions and assessments of the solutions to the questions found in the AP exam assignment. The solution to each question is presented in its own section.

The questions 1, 2 and 3 have been solved. The questions have been solved according to the assignment v1.0.

Question 1 – APMake Parser

This section presents the thoughts and assumptions regarding the solution as well as an assessment of the correctness of the implemented function. All the implemented source code can be found in Appendix A of this report.

Limitations

There are no limitations to the solution in regards to the assignment text.

Solution

The solution is implemented using the handed-out SimpleParse module to parse the string into and abstract syntax tree.

The program is designed so that one function handles one part of the abstract syntax tree. (i.e. pRule builds a Rule, pTargets builds target and so on). This makes it easier to maintain the code and a change in how something should be processed should only make for a need to make a change in as few functions as possible.

Since it makes for easier reading by the user and because it enables the reading of several characters at one, a target, prerequisite or command of the form “hej%” will be translated to [Lt “hej”, St] instead of [Lt “h”, Lt “e”, Lt “j”, St].

Functions

This section describes the functions in the program and what each of them handles in the solution.

parseString

The purpose of the parseString function is to determine whether there the string can be parsed into a makefile.

This is done using the fullParse function from SimpleParse which runs the pRules function until we have reached the end of the string.

The output of fullParse is then examined and if there is no Makefile we can parse the string to the empty list is returned to parseString and parseString outputs Left "Error". In case there is some Makefile we can create from the string we take the first element, *e*, of the list returned by fullParse and return *Right e*.

pRules

This function uses the applicative function *some* which takes the argument pRule to find if we can convert the text string into at least one rule to be returned.

pRule

The purpose of this function is to parse part of a text string into a Rule.

This is done by first using mySpaces to remove all spaces in the beginning of the string. We then use pTargets to parse the next part into a list of FileT's which will be out targets. We can use mySToken to see if the next non-space character is a ':'. We then proceed to use pPrereqs to find a list of FileT's which will be the prerequisites for the file names returned from pTargets.

We then use the applicative function *many* and give this the pCommandT function as an argument to see if there are any commands. We then use mySToken to see if the next non-space character is '\n' and afterwards we remove all the trailing spaces with mySpaces.

Lastly we check if any of the prerequisites or commands contain the literal St. If se we check if all the targets contain an St. If not, we reject the string. In all other cases we return the rule.

All this ensures that rules conform to the rules set in the assignment text.

pTargets

pTargets parses part of a string into a list of FileT's by first removing initial spaces with mySpaces and then using the applicative function *some* with the function *pFile* as argument to parse all the next files. We use *some* since we need at least one FileT in the targets.

pPrereqs

pPrereqs parses part of a string into a list of FileT's by first removing initial spaces with mySpaces and then using the applicative function *many* with the function *pFile* as argument to parse all the prerequisite files, if there are any. We use *many* since there does not have to be any prerequisite files to the target

pCommandT

pCommandT parses part of a string into a CommandT. All command literals must start with a the chars '\n' followed by '\t'. So, we first use mySToken "\n" to parse the next newline symbol and char '\t' to see if the next symbol is a tab symbol. After this we use *some* as gives this the argument *choice[pCommandT', pSt]* since a CommandT is a list of Frag's which can be either a literal or the symbol St. pCommandT' parses all the next allowed symbols until we meet a % in the string or some illegal character.

pCommandT'

Not to be confused with pCommandT, this function parses the next characters with the applicative *some* function which is given the function *pCommandT''* as an argument which returns a string of the next legal characters which are not '%'. The function then returs Lt followed by the string as Lt s.

pCommandT”

No to be confused with pCommandT and pCommandT’, this function parses the next character in the string and checks if the character is not a newline or a ‘%’. If we encounter a ‘\\’ character we check if the next character is either a newline or a ‘\\’ . If the next character is a newline we call pCommandT” recursively. If not, it must be a ‘\\’ and we return this character.

pFile

pFile is the function used to build a Template. We first remove all whitespaces with mySpaces and then use *some* given the function *choice[pLiberal, pSt]* to find a list of at least one frag to return

pSt

Function which checks if the next character in the string is a ‘%’. Uses the char function from the SimpleParse library to check this. Returns St if such a character is found.

pLiteral

pLiteral uses *some pLiteral’* to find the next continuous non-empty string, *s*, of allowed characters and returns this as Lt *s*.

pLiteral’

A function which finds the next allowed characters in matching the regular expression \[a-zA-Z0-9_./-\\]\.

If such a character is found we check if it is a ‘\\’. If not, we return the character, otherwise we check if the character matches the regular expression \[%: \\]\. If so, we return this new character.

myToken

Helper function which removes initial spaces until some condition is met. Inspired by token from SimpleParse.

mySToken

Helper function which removed initial spaces and then attempts to match against a string. Inspired by stoken from SimpleParse.

mySpaces

Helper function which removes all the next occurrences of the character ‘ ‘ (space) from a string. Inspired by the spaces function from SimpleParse

myIsSpace

Helper function which matches a character against the character ‘ ‘ (space). Returns true or false.

anyContainsSt

Helper function which checks if any Template in a list of templates contain the St frag. Returns true if one is found, false otherwise

allContainsSt

Helper function which checks if all Templates in a list of Templates contain the St frag. Returns true if so, false otherwise.

Assessment of the solution.

This section contains the results of automated tests as well as a conclusion to the tests.

Tests

The tests are specified in the ParserTests.hs file in the src directory of the handed in files. The tests can be rerun by loading the ParserTests module and running the functions by calling the function by *test name* (i.e. *test1*). The results are presented in Tabel 1.

Test name	**Purpose**	**Result**
test1	Tests if a string with only a single target is correctly processed	Satisfactory result
test2	Tests if %'s are processed correctly	Satisfactory result
test3	- Tests if several complicated rules are processed correctly - Tests if several prerequisites are processed correctly - Tests that spaces in commands are allowed	Satisfactory result
test4	Tests if two targets are handled correctly	Satisfactory result
test5	Tests if the safeguard against % in prereqs or commands only catches the error	Satisfactory result
test6	Tests if the safeguard against % in prereqs or commands only catches the error	Satisfactory result
test7	Tests if a lacking \n in the end of a rule is caught	Satisfactory result
test8	Tests if a simple string with 1 target, 1 prerequisite and no commands is processed correctly	Satisfactory result
test9	Tests if \\\\ in a FileT is handled correctly	Satisfactory result
test10	Tests if a \\\n in the commands is handled correctly	Satisfactory result
test11	Tests if a \\\\ in the commands is handled correctly	Satisfactory result
test12	Tests if all possible characters are caught correctly in a FileT	Satisfactory result

Tabel 1: APMake Parser Tests

Test conclusions

The tests show that the solution complies with the restriction in the assignment test and that it works correctly.

Conclusion

In conclusion, the program works correctly, different concerns are separated into their own specific functions, and all functionality complies with the assignment text.

Question 2 – APMake Interpreter

This section presents the thoughts and assumptions regarding the solution as well as an assessment of the correctness of the implemented functionality. All the implemented source code can be found in Appendix B of this report.

Limitations

The solution is limited to all build plans, not only minimal build plans.

The solution also builds prerequisites in reverse order. For instance in

Foo.bar : foo.o bar.o,

it will build bar.o first.

The solution, however, complies with all other specifications of the assignment text.

Solution

This part of the assignment is solved by using a monad which maintains a context. In this context we store the Makefile, a list of commands to be executed, a mapping of target files and the rules used to build those, the function to check if a file exists and an integer counter for the depth of the build steps.

The program works by recursively getting the prerequisites for the prerequisites of the wanted target file.

To make sure that we don't run in an infinite loop, we maintain a mapping of the rules used to build a specific file, thus a rule cannot be used for the same target twice.

It was assumed that when we have gone to *x* build steps and there are still files needed to be built, we just return Nothing since we cannot build the wanted file within the *x* steps.

Functions

This section presents the different functions and their purposes.

build

The build function uses the function runMF and runBuild to find a build plan. If runMf returns Just (commands, _) we return Just commands, if returns Nothing, we return Nothing

runBuild

runbuild is the function which handles checking the makefile, getting the prerequisites and commands to build the target file and calling the runBuildRecursion to build the prerequisites for the target.

runBuildRecursion

runbuildRecursion makes sure we don't do too many build steps. We assume that if there are no files left to be built we just return the function. If there are, though we pass the names of the files we need built on to the buildTargets function which handles this. We buildTargets return, we decrease the counter and call runBuildRecursion recursively on the new files needed to be built. Afterwards we add the commands to the list of commands with the addCommands function.

buildTargets

buildTargets handles finding the prerequisites of a list of targets and the commands used to build the target. The function getRule is used to do this and the list of files needed to build the targets and the commands used are returned to the runBuildRecursion function after being returned from getRule.

getRule

getRule is the function which checks if a given target can be built. We check each rule of the makefile to see if a rule can build the target AND if the rule has been used to build the target before. If possible, we return a list of the files needed to build the target and the list of commands needed to build the target.

Assessment of the solution

This section presents the tests of the solution, the results of the test as well as a conclusion on these tests.

Tests

The tests are specified in the InterpTests.hs file in the src directory of the handed in files. The tests can be rerun by loading the InterpTests module and running the functions by calling the function by *test name* (i.e. *test1*). The results are presented in Table 2.

Test name	**Purpose**	**Result**
test1	Test if program makes a build plan without %'s.	Satisfactory result
test2	tests if program catches wrong formatted makefile	Satisfactory result
test3	Tests if the program correctly returns nothing if not enough steps are allowed	Satisfactory result
test4	tests if the program works correctly with %'s	Satisfactory result
test5	tests if the program handles as specified in the assignment test example	Unsatisfactory result
test6	Tests if the program returns Nothing if there is a dependency cycle	Satisfactory result

Tabel 2: APMake Interpreter Tests

Test conclusion

As we can see from the tests, the program successfully builds a build plan in the allowed number of steps, although it builds prerequisite files in reverse for each target and doesn't produce a minimal build plan. The function however, correctly catches dependency cycles

Conclusion

In conclusion, for the specified functionality, the program works correctly.

The functionality too, is in general separated into different functions when needed by use of a monad.

All in all, the solution could have been better; however for the specified functionality it serves a purpose.

Question 3 – Robust Commands

This section presents the thoughts and assumptions regarding the solution as well as an assessment of the correctness of the implemented functionality. All the implemented source code can be found in Appendix C of this report.

Limitations

The only limitation is the return of the exception from the invoker. I am not sure that it conforms to the specified message format.

The solution does not have any inherit limitations in relation to the assignment text and the required specification that I know of apart from the above.

Solution

The solution solution consists of a server implementation which spawns a process maintaining a loop. This loop maintains the Callback Module, a list of the Process ID's for the Running invokers, as well as for the Waiting and Completed invokers, respectively, as well as the limit of concurrent invocations.

The loop manages the creation and abortion of processes which run the invoker-function.

The invoker function, after creation, wait for a signal from the server loop to start or abort the operation.

We assume that the State of the Callback Module doesn't change. We also assume that when an invocation is finished, we do not wish to invoke it again and it is therefore killed.

Functions

This section explains the functions in the gen_command module used to implement the wanted functionality

start/2

The start function attempts to run the initialize function of the module Mod. If it returns a tuple of {ok, State, Limit} we start a new process running the loop function with Mod, three empty lists, the state and the limit as argument. We then return {ok, ServerRef} where ServerRef is the Process ID of the process running loop. If something goes wrong we catch it and send it back as a tuple starting with error.

invoke/3

invoke sends a synchronous to the process maintaining the loop and wait for the response and locks until we receive a confirmation.

avast/2

It is assumed that avast doesn't have to get a response when asking to abort and operation. It therefore uses the async-function to send a message to the process specified the ServerRef.

ahoy/2

ahoy sends a synchronous request since we want an answer from the server loop.

furl/1

furl sends a synchronous request since we want to know that the server has been killed off.

loop/6

The loop maintains the callback module, three lists of running, waiting and completed tasks, respectively, the state of the server and the limit of concurrent executions. We assume that Limit is either a number or the atom infinite. Other values will cause a crash, most probably.

The loop manages the incoming messages from users and distributes them to the correct parties.

If the server received an invoke request, it spawn a new invoker process and sends the CID back to the calling function. It then waits for the invoker to respond. It then checks whether there is space for another running invoker. If so, the invoker is started, if not, it is queued.

If the server received the avast signal, the server sendt an avast signal to the specified invoker and assumes that it is received by the invoker. The invoker is then removed from the running and waiting list (we assume it is in one of them) and is added to the Completed list.

If we received an ahoy request, we check first if the CID is in the Running list, if not we check the waiting list in not, we check the completed list. We return a value based on the list it is found in and return this to the user.

If the loop received a success message we assume it is from an invoker and we then add the invoker to the completed list, send a message to the process specified by From with the reply. We delete the CID from the list of running invoker and see if there are any waiting. If one is waiting we send a start message to it, if not, we just restart the loop.

If we received a failure message, we do the exact same as when we receive a success message.

If we received a furl message, we send a synchronous avast request to all the waiting and running server. We send an asynchronous request to the running ones since we cannot be sure that they haven't finished in the time from sending the request to the time that the invoker received the request. We then call the handle_furl function for the Callback Module.

If we received any other request we just restart the loop unless there is a PID, then we send a reply with the tuple {unknown_request, Other} where other is the message.

Invoker/5

The invoker is a function which maintains a reference to the Parent process, a Callback Module, some Args, the State and the process that wants the result.

The invoker waits until a go or avast message is received. If a go is received the Module:handle_invocation function is called with Args and the State. If a reply is received we send the success message to the Parent. If we receive *blimey* we call the callback function carren/2 for the callback module and send a failure message to the parent.

If something goes wrong we send a message to the Parent with a well-formed error failure message. No matter what we kill the invoker process since it is no longer needed.

If the invoker received an avast signal we send an ok back and call the carren callback function.

Question 4 – Concurrent Plans

Not solved.

Appendix A – APMake Parser

Source code

```
module Parser.Impl where

import Control.Monad ( void )

import Ast
import SimpleParse

type ParseError = String -- Must be instance of (Eq, Show). (String derives Show
and Eq)

parseString :: String -> Either ParseError Makefile
parseString s =
    case fullParse (pRules <* eof) s of
        [] -> Left "Error"
        (e:_) -> Right e


pLiteral' :: Parser Char
pLiteral' = do
    c <- chars $ ['a'..'z'] ++ ['A'..'Z'] ++ ['0'..'9'] ++ ['_', '.', '/', '-',
'\\']
    if c == '\\'
    then do
        c2 <- chars ['%', ':', ' ', '\\']
        return c2
    else
        return c



pLiteral :: Parser Frag
pLiteral = do
    s <- some pLiteral'
    return $ Lt s

pSt :: Parser Frag
pSt = do
    _ <- char '%'
    return St

pFile :: Parser Template
pFile = do
    _ <- mySpaces
    some $ choice [pLiteral, pSt]
```

```haskell
pCommandT'' :: Parser Char
pCommandT'' = do
    c <- satisfy $ \x -> x /= '\n' && x /= '%'
    if c == '\\'
    then do
        c2 <- chars ['\n', '\\']
        if c2 == '\n'
        then pCommandT''
        else return c2
    else
        return c

pCommandT' :: Parser Frag
pCommandT' = do
    s <- some pCommandT''
    return $ Lt s

pCommandT :: Parser CommandT
pCommandT = do
    mySToken "\n"
    _ <- char '\t'
    some $ choice [pCommandT', pSt]



pTargets :: Parser [FileT]
pTargets = do
    _ <- mySpaces
    some pFile

pPrereqs :: Parser [FileT]
pPrereqs = do
    _ <- mySpaces
    many pFile


pRule :: Parser Rule
pRule = do
    _ <- mySpaces
    targets <- pTargets
    _ <- mySToken ":"
    prereqs <- pPrereqs
    commands <- many pCommandT
    _ <- mySToken "\n"
    _ <- mySpaces
    if anyContainsSt prereqs || anyContainsSt commands
    then
        if allContainsSt targets
            then return $ Rule targets prereqs commands
            else reject
    else
        return $ Rule targets prereqs commands

pRules :: Parser Makefile
pRules = some pRule
```

```haskell
-------------
-- Helpers --
-------------

myToken :: Parser a -> Parser a
myToken p = mySpaces >> p

mySToken :: String -> Parser ()
mySToken = void . myToken . string

mySpaces :: Parser String
mySpaces = munch myIsSpace

myIsSpace :: Char -> Bool
myIsSpace c = c == ' '

anyContainsSt :: [[Frag]] -> Bool
anyContainsSt fs = any (\x -> St `elem` x) fs

allContainsSt :: [[Frag]] -> Bool
allContainsSt fs = all (\x -> St `elem` x) fs


parseFile :: FilePath -> IO (Either ParseError Makefile)
parseFile path = fmap parseString $ readFile path
```

Appendix B – APMake Interpreter

Source code

```haskell
module Interp.Impl where

import Control.Monad()

import Ast

replace :: String -> Template -> String
replace _ [] = ""
replace s (x:xs) = case x of
                     St -> s ++ replace s xs
                     Lt l -> l ++ replace s xs


match :: String -> Template -> Maybe (Maybe String)
match "" [] = Just Nothing
match _ [] = Nothing
match s (t:ts) = case t of
                St -> trymatch s (t:ts)
                Lt l -> case prefix s l of
                         (True, ret) -> match ret ts
                         (False, _) -> Nothing

prefix :: String -> String -> (Bool, String)
prefix "" "" = (True, "")
prefix s "" = (True, s)
prefix "" _ = (False, "")
prefix (x:xs) (y:ys) = if x == y
                            then prefix xs ys
                            else (False, "")


trymatch :: String -> Template -> Maybe (Maybe String)
trymatch s t = trymatch' s s [] t



trymatch' :: String -> String -> String -> Template -> Maybe (Maybe String)
trymatch' s _ _ [St] = Just (Just s)
trymatch' _ [] _ _ = Nothing
trymatch' s (x:xs) ys t = if s == replace ys t
                         then Just (Just ys)
                         else trymatch' s xs (ys ++ [x]) t

check :: Makefile -> Bool
check ms = all (\x -> checkrule x) ms

checkrule :: Rule -> Bool
checkrule (Rule targets prereqs commands) =
    if (any (\x -> checktemplate x) prereqs) || any (\x -> checktemplate x)
commands
    then
        all (\x -> checktemplate x) targets
    else
      True
```

```haskell
checktemplate :: Template -> Bool
checktemplate t = any (\x -> St == x) t

type Context = (Makefile, [Command], [(String, [Rule])], (File -> Bool), Int)


newtype MF a = MF {
  runMF :: Context -> Maybe (a, Context)
}

instance Functor MF where
  fmap f m = m >>= \a -> return (f a)

instance Applicative MF where
  pure = return
  df <*> dx = df >>= \f -> dx >>= return . f

instance Monad MF where
  return a = MF $ \ s -> Just (a, s)

  m >>= f = MF $ \ s -> do
    case runMF m s of
        Nothing -> Nothing
        Just (a, s') -> runMF (f a) s'

  fail _ = MF $ \ _ -> Nothing



reject :: MF a
reject = MF $ \ _ -> Nothing

checkmakefile :: MF ()
checkmakefile = do
    (mf, _, _, _, _) <- get
    if check mf then return ()
                else reject

get :: MF Context
get = MF $ \ r -> Just (r, r)

set :: Makefile -> [Command] -> [(String, [Rule])] -> (File -> Bool) -> Int ->
MF ()
set makefile commands targets function int =
    MF $ \ _ -> Just ((), (makefile, commands, targets, function, int))

getCommands :: MF [Command]
getCommands = do
    (_, c, _, _, _) <- get
    return c

getFunction :: MF (File -> Bool)
getFunction = do
    (_, _, _, func, _) <- get
    return func
```

```haskell
getCounter :: MF Int
getCounter = do
    (_, _, _, _, i) <- get
    return i

decreaseCounter :: MF ()
decreaseCounter = do
    (m, c, s, f, i) <- get
    set m c s f $ i-1

getTargets :: MF [(String, [Rule])]
getTargets = do
    (_, _, targets, _, _) <- get
    return targets

addTarget :: String -> Rule -> MF ()
addTarget s r = do
    (m, c, targets, f, i) <- get
    newtargets <- addTarget' s r [] targets
    set m c newtargets f i

addTarget' :: String -> Rule -> [(String, [Rule])] -> [(String, [Rule])] -> MF
[(String, [Rule])]
addTarget' s r targets [] = return $ (s, [r]) : targets
addTarget' s r targets ((target, rulelist):ts) =
    if s == target
    then return $ targets ++ [(target, r:rulelist)] ++ ts
    else
        addTarget' s r ((target, rulelist):targets) ts

-- Returns False if rule not used for particular target
-- Return
ruleUsed :: String -> Rule -> MF Bool
ruleUsed s r = do
    targets <- getTargets
    ruleUsed' s r targets

ruleUsed' :: String -> Rule -> [(String, [Rule])] -> MF Bool
ruleUsed' _ _ [] = return False
ruleUsed' s r ((target, rules):targets) =
    if s == target
    then return $ r `elem` rules
    else ruleUsed' s r targets


addCommand :: Command -> MF ()
addCommand command = do
    (mf, commands, s, f, i) <- get
    set mf (command : commands) s f i

addCommands :: [Command] -> MF ()
addCommands commands1 = do
    (mf, commands2, s, f, i) <- get
    set mf (commands2 ++ commands1) s f i

build :: Makefile -> File -> (File -> Bool) -> Int -> Maybe [Command]
build mf s f i = case runMF (runBuild s) (mf, [], [], f, i) of
    Just (commands, _) -> Just commands
    Nothing -> Nothing
```

```
runBuild :: String -> MF [Command]
runBuild s = do
    checkmakefile
    (prereqs, commands) <- getRule s
    newprereqs <- trimPrereqs prereqs
    decreaseCounter
    runBuildRecursion newprereqs
    addCommands commands
    getCommands

trimPrereqs :: [String] -> MF [String]
trimPrereqs prereqs = do
    func <- getFunction
    return $ filter (\x -> not $ func x) prereqs



runBuildRecursion :: [String] -> MF ()
runBuildRecursion [] = return ()
runBuildRecursion prereqs = do
    i <- getCounter
    if i < 1
    then reject
    else do
       (newprereqs, commands) <- buildTargets prereqs
       decreaseCounter
       runBuildRecursion newprereqs
       addCommands commands



buildTargets :: [String] -> MF ([String], [String])
buildTargets [] = return ([], [])
buildTargets (t:ts) = do
     (prereqs1, commands1) <- buildTargets ts
     (prereqs2, commands2) <- getRule t
     return $ (prereqs1 ++ prereqs2, commands1 ++ commands2)
```

```
getRule :: String -> MF ([String], [String])
getRule s = do
    (mf, _, _, _, _) <- get
    getRule' s mf
        where getRule' _ [] = reject
              getRule' s1 (rule:rs) = do
              used <- ruleUsed s1 rule
              if used
              then
                getRule' s1 rs
              else
                getRule'' s1 rule rule
                where getRule'' _ (Rule [] _ _) _ = getRule' s rs
                      getRule'' s2 (Rule (t:ts) prereqs commands) r =
                        case match s2 t of
                        Nothing -> getRule'' s2 (Rule ts prereqs commands) r
                        Just Nothing -> let
                                        tempPrereqs = map (\x -> replace "" x)
prereqs
                                        newcommands = map (\x -> replace "" x)
commands
                                        in do
                                        addTarget s2 r
                                        newprereqs <- trimPrereqs tempPrereqs
                                        return (newprereqs, newcommands)
                        Just (Just s3) -> let
                                        tempPrereqs = map (\x -> replace s3 x)
prereqs
                                        newcommands = map (\x -> replace s3 x)
commands
                                        in do
                                        addTarget s2 r
                                        newprereqs <- trimPrereqs tempPrereqs
                                        return (newprereqs, newcommands)
```

Appendix C – gen_command

Source code

```
-module(gen_command).

-export([start/2, invoke/3, avast/2, ahoy/2, furl/1]).

start(Mod, Args) ->
    try Mod:initialise(Args) of
        {ok, State, Limit} ->
            ServerRef = spawn(fun() -> loop(Mod, [], [], [], State, Limit) end),
            {ok, ServerRef};
        Other ->
            {error, Other}
    catch
        Other ->
            {error, Other}
    end.


invoke(ServerRef, Args, From) ->
    request_reply(ServerRef, {invoke, Args, From}).

avast(ServerRef, CID) ->
    async(ServerRef, {avast, CID}).

ahoy(ServerRef, CID) ->
    request_reply(ServerRef, {ahoy, CID}).

furl(ServerRef) ->
    request_reply(ServerRef, furl).
```

```
loop(Module, Running, Waiting, Completed, State, Limit) ->
    receive
        {Pid, {invoke, Args, From}} ->
            Me = self(),
            CID = spawn(fun() -> invoker(Me, Module, Args, State, From) end),
            reply(Pid, {Me, CID}),
            receive
                % Vi venter på, at invokeren er klar før vi går videre
                {CID, alive} ->
                    % Hvis der stadig er plads til en ekstra aktiv så kører vi
den næste.
                    % Hvis ikke så sætter vi den i vente-kø.
                    case length(Running) < Limit of
                        true ->
                            NewRunning = Running ++ [CID],
                            startInvoker(CID),
                            loop(Module, NewRunning, Waiting, Completed, State,
Limit);
                        false ->
                            NewWaiting = Waiting ++ [CID],
                            loop(Module, Running, NewWaiting, Completed, State,
Limit)
                    end
            end;

        {_, {avast, CID}} ->
            async(CID, {self(), avast}),
            NewRunning = lists:delete(CID, Running),
            NewWaiting = lists:delete(CID, Waiting),
            NewCompleted = Completed ++ [CID],
            loop(Module, NewRunning, NewWaiting, NewCompleted, State, Limit);

        {Pid, {ahoy, CID}} ->
            case lists:any(fun(E) -> E == CID end, Running) of
                true ->
                    reply(Pid, running);
                false ->
                    case lists:any(fun(E) -> E == CID end, Waiting) of
                        true ->
                            reply(Pid, queued);
                        false ->
                            case lists:any(fun(E) -> E == CID end, Completed) of
                                true ->
                                    reply(Pid, completed);
                                false ->
                                    reply(Pid, {error, nomatch})
                            end
                    end
            end,
            loop(Module, Running, Waiting, Completed, State, Limit);

        {CID, {success, From, Reply}} ->
            reply(From, {success, Reply}),
            NewCompleted = Completed ++ [CID],
            NewRunning = lists:delete(CID, Running),
            % Vi ser om der er nogen som venter
            case length(Waiting) > 0 of
                true ->
                    Next = lists:nth(1, Waiting),
```

```
                    NewWaiting = lists:delete(Next, Waiting),
                    startInvoker(Next),
                    NewNewRunning = [Next] ++ NewRunning,
                    loop(Module, NewNewRunning, NewWaiting, NewCompleted, State,
Limit);
                false ->
                    loop(Module, NewRunning, Waiting, NewCompleted, State,
Limit)
            end;

        {CID, {failure, From, Reply}} ->
            reply(From, {failure, Reply}),
            NewCompleted = Completed ++ [CID],
            NewRunning = lists:delete(CID, Running),
            case length(Waiting) > 0 of
                true ->
                    Next = lists:nth(1, Waiting),
                    NewWaiting = lists:delete(Next, Waiting),
                    startInvoker(Next),
                    NewNewRunning = [Next] ++ NewRunning,
                    loop(Module, NewNewRunning, NewWaiting, NewCompleted, State,
Limit);
                false ->
                    loop(Module, NewRunning, Waiting, NewCompleted, State,
Limit)
            end;
        {Pid, furl} ->
            lists:foreach(fun(CID) ->
                             request_reply(CID, avast)
                          end,
                          Waiting),
            lists:foreach(fun(CID) ->
                               async(CID, {self(), avast})
                          end,
                          Running),
            Module:handle_furl(State),
            reply(Pid, ok);
        {Pid, Other} ->
            reply(Pid, {unknown_request, Other}),
            loop(Module, Running, Waiting, Completed, State, Limit);
        _ ->
            loop(Module, Running, Waiting, Completed, State, Limit)
    end.
```

```erlang
invoker(Parent, Module, Args, State, From) ->
    Parent ! {self(), alive},
    receive
        {Parent, go} ->
            try Module:handle_invocation(Args, State) of
                {reply, Reply} ->
                    reply(Parent, {success, From, Reply});
                blimey ->
                    Module:carren(State, Args),
                    reply(Parent, {failure, From, aborted})
            catch
                Error ->
                    reply(Parent, {failure, From, Error})
            end;
        {Parent, avast} ->
            reply(Parent, ok),
            Module:carren(State, Args)
    end.



%%% Synchronous communication
reply(Pid, Reply) ->
    Pid ! {self(), Reply}.

async(Pid, Msg) ->
    Pid ! Msg.

request_reply(ServerRef, Msg) ->
    ServerRef ! {self(), Msg},
    receive {ServerRef, Reply} ->
        Reply
    end.

startInvoker(CID) ->
    Me = self(),
    async(CID, {Me, go}).
```