Universiteit van Amsterdam

Programmeertalen

Week 7 - Haskell

David Veenstra - Stephan van Schaik

ArnoldH

Introductie

De afgelopen weken hebben jullie kennis mogen maken met zes verschillende programmeertalen, namelijk: Bash, Prolog, Haskell, Python, Go en Erlang. Deze week gaan we geen zevende programmeertaal bekijken, maar gaan we deze juist implementeren: ArnoldH. Deze taal is geïnspireerd door ArnoldC en heeft daarom ook dezelfde motivatie:

"Although the one-liners of Arnold Schwarzenegger are fairly well known the true semantics of the uttering is yet to be understood. This project tries to discover new meanings from the Arnold movies with the means of computer science. "

Deze opdracht bestaat uit twee delen. In het eerste deel moet er een parser geschreven worden, die de code zal ontleden in verschillende basiselementen en tegelijkertijd kijkt of de grammatica, de volgorde van deze elementen, correct is. In het schrijven van deze parser zal waarschijnlijk het meeste werk zitten, dus houd daar rekening mee. En in het tweede deel wordt er code geschreven die deze basiselementen kan evalueren en interpreteren, zodat de code daadwerkelijk uitgevoerd kan worden. Voordat je begint met het schrijven van de code, wordt het sterk aangeraden om eerst naar *Ast.hs* te kijken en de volledige beschrijving van ArnoldH te bekijken, voordat er begonnen wordt met de opdracht.

Oefenopgave

Voor deze opdracht wordt er veelvuldig gebruik gemaakt van I/O en de do-notatie. Om deze onder de knie te krijgen, is het aangeraden om allereerst met de oefenopgave te beginnen, namelijk het implementeren van een guess the number game in Haskell. Hierbij wordt er zoals bij de Mastermind game van de derde week een secret meegegeven aan een functie die vervolgens door de speler geraden moet worden. Bij dit spel wordt de speler telkens gevraagd om het getal te raden. Hierbij kunnen de volgende situaties zich voordoen:

- Als de speler het correcte antwoord heeft geraden, dan ziet hij: "You've won the game!"
- Als de huidige poging dichter bij het correcte antwoord ligt dan de vorige, dan ziet hij: "You're warmer!"
- Als de huidige poging verder weg ligt van het correcte antwoord, dan ziet hij: "You're colder!"

Een voorbeeld van hoe het spel eruit ziet als het gespeeld wordt, ziet er als volgt uit:

Shell snippet

```
Prelude> guessingGame 42
What's your guess? 17
What's your guess? 78
You're colder!
What's your guess? 56
You're warmer!
What's your guess? 24
You're colder!
What's your guess? 32
You're warmer!
What's your guess? 42
You've won the game!
```

Importeer de volgende modules:

Guess.hs

```haskell
import Data.Char
import Data.Maybe
```

> Bekijk voor het maken van deze opdracht de documentatie van de Data.Maybe module (`https://hackage.haskell.org/package/base-4.8.1.0/docs/Data-Maybe.html`).

Het is handig om te beginnen met het implementeren van de `getNum :: String -> IO Int` functie. Zolang er geen geldig getal is opgegeven geeft deze functie de meegegeven string als prompt weer, waarna de gebruiker om input gevraagd wordt. Implementeer hiervoor ook de functie `isNum :: String -> Bool`, die kijkt of een gegeven string een geldig getal is. Gebruik daarvoor de functies: `all :: (a -> Bool) -> [a] -> Bool` en `isDigit :: Char -> Bool`.

Nu dat we de gebruiker kunnen vragen om een getal in te voeren kunnen we het spelletje zelf implementeren. Het spel kan geïmplementeerd worden met een drietal functies:

- `checkAnswer :: Int -> Maybe Int -> Int -> IO ()` om te controleren of het antwoord correct is.
- `checkProximity :: Int -> Maybe Int -> Int -> IO ()` om aan te geven of de gebruiker dichter bij het correcte antwoord of verder van het correcte antwoord vandaan is.
- `guessingGame' :: Int -> Maybe Int -> IO ()` om de gebruiker te vragen om een getal en te controleren of het getal correct is.

De argumenten die meemgegeven worden aan deze functies zijn:

- `secret` (`Int`): het correcte antwoord.
- `previous` (`Maybe Int`): de vorige poging.
- `current` (`Int`): de huidige poging.

Tenslotte moet er een vereenvoudige functie `guessingGame :: Int -> IO ()` geïmplementeerd worden, die een `secret` accepteert.

> Vergeet niet dat je je functies stap voor stap en interactief met *ghci* kunt testen.

ArnoldH

ArnoldH is een simpele programmeertaal met ondersteuning voor integer operaties, logische operator, while-statements, if-statements, functies en een print-statement. Een ArnoldH-programma bestaat op het hoogste niveau alleen maar uit een aantal functie declaraties, waarbij de `main` functie wordt uitgevoerd door de interpreter als deze aanwezig is. Verder kunnen er dus geen globale variabelen gedeclareerd worden.

Elk commando bestaat uit een aantal argumenten. Deze argumenten kunnen of een naam, bestaande uit kleine letters (bv. `result`), of een waarde, bestaande uit cijfers (bv. `42`), aannemen. Ieder commando moet op een eigen regel staan.

De binary operatoren zijn commando's die drie argumenten aannemen: een naam en twee waardes. Het eerste argument is de naam van de variabele waar het resultaat in wordt opgeslagen, de twee andere argumenten vormen de invoer voor de berekeningen. Hieronder een voorbeeld:

The binary operators

```
% main function
LISTEN TO ME VERY CAREFULLY main
  % istwo(3)
  DO IT NOW istwo 3

  YOU SET US UP x 2

  % result = istwo(x)
  DO IT NOW istwo x :: result

  IT'S SHOW TIME result
```

```
12
13 % declaration of a function, istwo, with one argument
14 LISTEN TO ME VERY CAREFULLY istwo input
15   GET DOWN isequal input 2
16   I LIED isequal isequal
17
18   % return True if input == 2
19   BECAUSE I'M GOING TO SAY PLEASE isequal
20     HASTA LA VISTA, BABY 1
21
22   % 0 is returned here as default
```

In ArnoldH zijn de volgende binaire operatoren gedefinieerd:

Operator	Betekenis
GET UP	optellen
GET DOWN	aftrekken
YOU'RE FIRED	vermenigvuldigen
HE HAD TO SPLIT	delen
CONSIDER THAT A DIVORCE	logische OR
KNOCK KNOCK	logische AND
LET OFF SOME STEAM BENNET	kleiner dan

Table 1: De binaire operatoren

Naast de binaire operatoren zijn er ook twee unaire operatoren gedefinieerd in ArnoldH. Deze operator accepteren maar één waarde in tegenstelling tot de binaire operatoren. Hieronder een voorbeeld:

The unary operators

```
1 LISTEN TO ME VERY CAREFULLY main
2   % a = 1
3   YOU SET US UP a 1
4
5   % b = not b
6   % If b is zero then the result is 1, otherwise the result is 0
7   I LIED b b
```

De if-statement, while-statement en functie declaraties zijn zogenaamde block statements, omdat er bij deze statements een blok aan code hoort. In C of Java wordt er gebruik gemaakt van accolades om code te groeperen in blokken. In ArnoldH moet de code, net zoals bij Python, worden ingesprongen om ze te groeperen in blokken. In ArnoldH moet er ingesprongen worden met tabs.

Verder wordt er gebruik gemaakt van lexical scoping, dat houdt in dat een functie of variabele die gedeclareerd is binnen een blok, alleen binnen dat blok en de onderliggende blokken bereikbaar is. De meeste moderne programmeertalen zoals bv. Haskell en Python maken ook gebruik van lexical scoping.

Aangezien ArnoldH alleen maar integers kent, wordt de waarde nul gebruikt voor *False* en is iedere andere waarde gelijk aan *True*.

Hieronder een voorbeeld van if- en while-statements:

if-statements and while-statements

```
1 % main function
2 LISTEN TO ME VERY CAREFULLY main
3   % a = 1
4   YOU SET US UP a 1
5   YOU SET US UP b 0
6
7
8   % if a then a + 1
9   BECAUSE I'M GOING TO SAY PLEASE a
10    GET UP a a 1
11
```

```
    YOU SET US UP counter 10

    % while counter
    I'LL BE BACK counter
      GET DOWN counter counter 1

      GET UP b b 1

    % the variable counter cannot be reached here
```

Functie declaraties beginnen met `LISTEN TO ME VERY CAREFULLY`, gevolgd door een of meerdere namen, waarbij de eerste de functienaam is en de overige de namen van de argumenten. Een waarde kan geretourneerd worden met `HASTA LA VISTA, BABY`. Wanneer er niet expliciet een waarde geretourneerd wordt, dan wordt er impliciet de waarde nul geretourneerd.

Tenslotte kan `DO IT NOW` gebruikt worden om een functie aan te roepen, waarbij er optioneel kan worden aangegeven waar optioneel kan worden aangegeven waar het resultaat moet worden opgeslagen. `IT'S SHOWTIME` kan een waarde uitprinten. En elke regel dat de character '%' bevat, wordt beschouwt als commentaar.

Function declaration, calling functions and print statement

```
% main function
LISTEN TO ME VERY CAREFULLY main
  % istwo(3)
  DO IT NOW istwo 3

  YOU SET US UP x 2

  % result = istwo(x)
  DO IT NOW istwo x :: result

  IT'S SHOW TIME result

% declaration of a function, istwo, with one argument
LISTEN TO ME VERY CAREFULLY istwo input
  GET DOWN isequal input 2
  I LIED isequal isequal

  % return True if input == 2
  BECAUSE I'M GOING TO SAY PLEASE isequal
    HASTA LA VISTA, BABY 1

  % 0 is returned here as default
```

Parser

Tijdens het implementeren van de parser gaan we onderscheid maken tussen drie verschillende soort statements: unaire operatoren, binaire operatoren en statements zoals het print-statement en het return-statement die maar één argument aannemen. Voor deze statements gaan we hulpfuncties schrijven.

Verder gaan we tijdens het parsen onderscheid maken tussen twee vormen van fouten. De eerste vorm is wanneer een parser niet in staat is om iets te parsen, terwijl een andere daartoe mogelijk wel in staat is. De tweede vorm is wanneer het duidelijk is dat de syntax niet correct is, bij dit soort fouten hoort het programma afgesloten te worden met behulp van de functie `error :: String -> a`.

Een voorbeeld van de eerste vorm is wanneer "`IT'S SHOW TIME a`" wordt ontleed met de parser voor return-statements, hierbij willen we niet direct het programma af sluiten, want de parser voor het print-statement kan dit wel ontleden. Een voorbeeld van de tweede vorm is bij het parsen van "`IT'S SHOW TIME A`", omdat variabelen alleen maar uit kleine letters mogen bestaan.

We beginnen met de parser `indentifier :: String -> String` voor het ontleden van een naam. Deze parser moet

direct falen met `error` als er geen geldige naam kan worden ontleed.

Shell snippet

```
*Main> indentifier "temp"
"temp"
*Main> indentifier "TEMP"
"*** Exception: Could not parse indentifier: TEMP
*Main>
```

De `parseValue :: String -> Value` parser is vergelijkbaar, maar moet i.p.v. een naam, een `Value` kunnen parsen.

Shell snippet

```
*Main> parseValue "temp"
Variable "temp"
*Main> parseValue "1234"
IntVal 1234
*Main> parseValue "TEMP"
Variable "*** Exception: Could not parse indentifier: TEMP
```

Elk statement begint met een quote van Meneer de Terminator, gevolgd door een aantal argumenten. De functie `parsePhrase :: String -> String -> Maybe [String]` is een hulpfunctie die de witruimte aan het begin en aan het einde van de regel weghaalt, kijkt of de regel begint met de quote, en de argumenten die gevonden worden na de quote retourneert.

Shell snippet

```
*Main> parsePhrase "DO IT NOW" "   DO IT NOW 1 temp result   hello 4"
Just ["1","temp","result","hello","4"]
*Main> parsePhrase "DO IT NOW" "Nope"
Nothing
```

Zoals eerder vermeld, zijn er een aantal statements die slechts één argument aannemen (bv. het print- en het return-statement). Voor deze statements maken we een hulpfunctie `parseWithSingle :: (String -> Statement) -> String -> String -> Maybe Statement`. Het eerste argument van deze functie is een functie die het argument na het statement transformeert naar een `Statement`, het tweede argument is de quote, en het laatste argument is de regel die ontleed moet worden. Maak hierbij gebruik van de functies die hierboven beschreven zijn. Wanneer deze functie geïmplementeerd is, kan de parser voor het print-statement zo geïmplementeerd worden. De parsers die uit deze functie komen moeten direct falen als er meer dan één argument na de quotes wordt teruggevonden.

Shell snippet

```
*Main> let parsePrint = parseWithSingle (Print . parseValue)  "IT'S SHOW TIME"
*Main> parsePrint "   IT'S  SHOW TIME 23"
Just (Print (IntVal 23))
*Main> parsePrint "   IT'S  SHOW TIME temp"
Just (Print (Variable "temp"))
*Main> parsePrint "   Nope temp"
Nothing
*Main> parsePrint "   IT'S  SHOW TIME TEMP"
Just (Print *** Exception: Could not parse value: TEMP
*Main> parsePrint "   IT'S  SHOW TIME var1 var2"
*** Exception: Could not parse: IT'S SHOW TIME
```

De eerste regel van de if-statement en de while-statements beginnen ook met één argument. Gebruik `parseWithSingle` om een parser te maken voor het return-statement, print-statement en de eerste regels van een if- en while-statement. Het ontleden van het blok dat na een if- en een while-statement volgt wordt later geïmplementeerd.

Shell snippet

```
*Main> parseWhileHeader "I'LL BE BACK a"
Just (While (Variable "a") [])
```

De parsers voor de unaire en binaire operatoren kunnen op een vergelijkbare manier geïmplementeerd worden. Implementeer de hiervoor de volgende functies:

- `parseUnary :: (String -> Value -> Statement) -> String -> String -> Maybe Statement`
- `parseBinary :: (String -> Value -> Value -> Statement) -> String -> String -> Maybe Statement`

Alleen het eerste argument is verschillend, omdat binaire operatoren meer argumenten hebben dan unaire operatoren. De parser die geretourneerd wordt door deze functies moet ook direct falen als er niet het juiste aantal argumenten gevonden wordt na de quote.

Het ontleden van een functieaanroep is iets ingewikkelder, omdat er een variabel aantal argumenten aanwezig kunnen zijn, en er daarnaast mogelijk een return variabele aanwezig kan zijn. Implementeer twee de functies, de eerste ontleed een functieaanroep zonder return variabele, en de tweede een functieaanroep met een return variabele. Combineer deze twee functies dan tot een functie: `parseFunctionCall :: String -> Maybe Statement`.

Shell snippet

```
*Main> parseFunctionCallWithoutReturnVariable "DO IT NOW f x :: result"
Nothing
*Main> parseFunctionCallWithoutReturnVariable "DO IT NOW f x"
Just (FunctionCall "f" [Variable "x"] Nothing)
*Main> parseFunctionCallWithReturnVariable "DO IT NOW f x :: result"
Just (FunctionCall "f" [Variable "x"] (Just "result"))
*Main> parseFunctionCallWithReturnVariable "DO IT NOW f x"
Nothing
*Main> parseFunctionCall "DO IT NOW f x"
Just (FunctionCall "f" [Variable "x"] Nothing)
*Main> parseFunctionCall "DO IT NOW f x :: result"
Just (FunctionCall "f" [Variable "x"] (Just "result"))
```

We kunnen `parseUnary` en `parseBinary` gebruiken om voor elke operator een parser te maken. Het probleem is echter dat elk van deze parsers maar één operator kan parsen. We willen eigenlijk een parser die alle mogelijke operatoren kan parsen. Hetzelfde principe geldt voor het combineren van alle statement parser tot één enkele parser.

Hiervoor implementeren we de parser `parseChoice :: [String -> Maybe a] -> String -> Maybe a`. Deze parser neemt een lijst van `parsers` en probeert een stuk code te ontleden met iedere mogelijke parser totdat er een parser gevonden wordt die het desbetreffende stuk code succesvol kan ontleden. In het geval dat geen van de parsers in staat is om het stuk code te ontleden, retourneert deze functie `Nothing`.

Shell snippet

```
*Main> parseChoice [\_ -> Nothing, \_ -> Just 10, \ _ -> Nothing, \_ -> Just 3] "hello"
Just 10
```

Implementeer de volende parsers met behulp van `parseChoice`

- `parseOperation :: String -> Maybe Statement`
- `parseSingle :: String -> Statement`

Met `parseSingle` hebben we nu een parser die bijna ieder statement kan ontleden, m.u.v. de blokken code die bij sommige statements en functie declaraties horen. Voordat we een functie kunnen ontleden, moeten we een blok met mogelijk geneste blokken kunnen ontleden. Met behulp van de andere parsers kunnen we de functie `parseBlock :: Int -> [String] -> [Statement] -> ([Statement], [String])` implementeren. Deze heeft als argumenten de huidige diepte waarop de code is ingesprongen, gemeten als het aantal tabs, de regels code die nog verwerkt moeten

worden, en de statements die tot nu toe ontleed zijn. Deze functie retourneert een tuple van de ontlede waardes die bij het blok op het gegeven niveau horen, en de regels code die nog ontleed moet worden. Als een regel code een niveau lager ligt, dan is het einde van het blok gevonden.

Shell snippet

```
*Main> parseBlock 1 ["\tIT'S SHOW TIME a", "IT'S SHOW TIME b"] []
([Print (Variable "a")],["IT'S SHOW TIME b"])
*Main> parseBlock 1 ["\tIT'S SHOW TIME a", "\tIT'S SHOW TIME b"] []
([Print (Variable "a"),Print (Variable "b")],[])
*Main> parseBlock 1 ["\tIT'S SHOW TIME a", "\tI'LL BE BACK b", "\t\tIT'S SHOW TIME a"] []
([Print (Variable "a"),While (Variable "b") [Print (Variable "a")]],[])
*Main> parseBlock 0 ["\tIT'S SHOW TIME a", "\tI'LL BE BACK b"] []
*** Exception: Unexpected indent
```

Nu hebben we alle bouwblokken die we nodig hebben om de parser te maken. Implementeer de laatste functies van de template. Test de parser goed voordat het volgende deel wordt geïmplementeerd, hiervoor zijn er een aantal testprogramma's meegeleverd.

Evaluatie

Nu we klaar zijn met het implementeren van de parser, kunnen we beginnen met het implementeren van de evaluatie van het programma. Bij het evalueren van het programma moet er een toestand bijgehouden worden, en dit is precies waar `ProgramState` voor dient. Hierin worden de functies en de variabelen bijgehouden. Verder moeten er hulpfuncties aangemaakt worden die het gemakkelijk maken om lexical scoping te implementeren. Dit wordt gedaan met behulp van een stack van `Map`'s. Wanneer er wordt ingesprongen, dan moet er een nieuwe scope op de stack worden gepusht, zodat de nieuwe variabele in de nieuwe scope terecht komen. Wanneer de evaluatie van een blok code klaar is, dan wordt de meest recente scope van de stack gepopt en zijn de variabelen van dat blok onbereikbaar. Echter blijven de variabelen, die in de hogere blokken gedefinieerd zijn, nog steeds bereikbaar.

> Bekijk voor het maken van deze opdracht de documentatie van de `Data.Map.Strict` module (`https://hackage.haskell.org/package/containers-0.5.6.3/docs/Data-Map-Strict.html`). Deze module zal je ook nodig hebben om korte code te schrijven voor de parser.

Function declaration, calling functions and print statement

```
% main function
LISTEN TO ME VERY CAREFULLY main
  YOU SET US UP x 1

  % scope of main exists out of variable x
  BECAUSE I'M GOING TO SAY PLEASE 1
    % new scope is pushed to stack

    % variable y is added to this scope
    YOU SET US UP y 3

    % the existance of variable x is found one level down the stack
    % and is modified
    YOU SET US UP x 2

  % the scope of the if  statement is popped, hence variable y
  % doesn't exist anymore
```

Implemteer de volgende functies:

- `popScope :: ProgramState -> ProgramState`
- `pushNewScope :: ProgramState -> ProgramState`

- `emptyScope :: ProgramState -> ProgramState`

De laatste functie is nodig om de stack leeg te maken, wat nodig is bij het aanroepen van een functie.

Shell snippet

```
*Main> emptyProgramState
ProgramState [fromList []] (fromList [])
*Main> pushNewScope emptyProgramState
ProgramState [fromList [],fromList []] (fromList [])
*Main> pushNewScope (pushNewScope emptyProgramState)
ProgramState [fromList [],fromList [],fromList []] (fromList [])
*Main> popScope (pushNewScope emptyProgramState)
ProgramState [fromList []] (fromList [])
*Main> popScope (popScope  emptyProgramState)
ProgramState [] (fromList [])
*Main> emptyScope (pushNewScope (pushNewScope emptyProgramState))
ProgramState [fromList []] (fromList [])
```

De `setVar :: ProgramState -> String -> Int -> ProgramState` functie kent een gegeven waarde toe aan een gegeven variabele. Zoals eerder beschreven wordt de hele stack doorzocht naar een scope waarin de variabele gedeclareerd is. Als die gevonden wordt, dan wordt er een nieuwe waarde aan deze variabele toegekend. Anders wordt er een nieuwe variabele toegevoegd aan de scope die zich aan de top van de stack bevindt.

De drie toekenningen die in de code snippets gedaan worden, zien er ongeveer als volgt uit:

Shell snippet

```
*Main> let t1 = setVar emptyProgramState "x" 1
*Main> let t2 = setVar (pushNewScope t1) "y" 3
*Main> let t3 = setVar t2 "x" 2
*Main> t1
ProgramState [fromList [("x",1)]] (fromList [])
*Main> t2
ProgramState [fromList [("y",3)],fromList [("x",1)]] (fromList [])
*Main> t3
ProgramState [fromList [("y",3)],fromList [("x",2)]] (fromList [])
*Main> popScope t3
ProgramState [fromList [("x",2)]] (fromList [])
```

Implementeer nu ook de volgende functies:

- `getVar :: ProgramState -> String -> Maybe Int`
- `getFunction :: ProgramState -> String -> Maybe FunctionPrototype`
- `addFunction :: ProgramState -> FunctionPrototype -> ProgramState`
- `isVarDefined :: ProgramState -> String -> Bool`
- `isFunctionDefined :: ProgramState -> String -> Bool`

In het vervolg mag er worden aangenomen dat als er een variabele of een functie wordt opgevraagd die niet bestaat, dat het programma dan mag worden afgesloten met `error`. Maak een hulpfunctie `getValue :: ProgramState -> Value -> Int` aan, die de integerwaarde uit de `Value` pakt, of de variabele opzoekt in de `ProgramState`.

Shell snippet

```
*Main> getValue emptyProgramState (IntVal 3)
3
*Main> getValue (setVar emptyProgramState "x" 3) (Variable "x")
```

```
3
*Main> getValue (setVar emptyProgramState "x" 3) (Variable "y")
*** Exception: undefined variable: y
```

Nu de hulpfuncties klaar zijn, begint het echte werk. Om te oefenen maken we eerst de `evalBinary :: ProgramState -> Statement -> ProgramState` functie aan, die een binaire operatie kan evalueren. Deze functie evalueert een binaire operator en retourneert de nieuwe `ProgramState`.

Shell snippet

```
*Main> let st = evalBinary emptyProgramState (Add "a" (IntVal 1)  (IntVal 1))
*Main> let st2 = evalBinary st (Mult "a" (Variable "a")  (Variable "a"))
*Main> st
ProgramState [fromList [("a",2)]] (fromList [])
*Main> st2
ProgramState [fromList [("a",4)]] (fromList [])
```

Nu komt de belangrijkste functie voor evaluatie, de `evaluateBlock :: ProgramState -> [Statement] -> IO (ProgramState, Maybe Int)` functie. Deze functie moet, zoals de naam zegt, een code blok kunnen evalueren. Hierbij wordt er gebruik gemaakt van `IO`, omdat er geprint moet kunnen worden. Gebruik hiervoor de kennis die je hebt opgedaan bij het maken van de oefenopgave.

Deze functie retourneert in een `IO` tuple de nieuwe `ProgramState` en een mogelijke returnwaarde. Een `Just` returnwaarde geeft aan dat er geretourneerd is, en deze informatie is belangrijk omdat deze aangeeft dat als er binnen een if-statement of een while-statement geretourneerd wordt, de hele functie klaar is met de evaluatie.

Het is aangeraden om te beginnen met de statements die makkelijk zijn. Op volgorde zijn dit:

1. het print-statement, het return-statement en de operatoren.
2. het if-statement en het while-statement.
3. het statement voor functieaanroep.

Denk bij elk statement eerst goed na wat de semantiek moet zijn en schrijf dan pas de implementatie.

Tenslotte kan de `evaluateProgram :: [FunctionPrototype] -> IO (ProgramState, Maybe Int)` functie geïmplementeerd worden. Hierbij hoeft slechts de juiste beginstaat berekend te worden, en kan `evaluateBlock` gebruikt worden om de `main` functie te evalueren. Daarmee is de interpreter eindelijk klaar en kan er onderzoek worden gedaan naar de diepere betekenis van de quotes van de quotes van Arnold Schwarzenegger door programma's te schrijven in ArnoldH.