



Programmatie logica

Procedures en functies

Webleren

School je gratis bij via het internet. Waar en wanneer je wilt.

www.vdab.be/webleren

© COPYRIGHT 2015 VDAB

Niets uit deze syllabus mag worden verveelvoudigd, bewerkt, opgeslagen in een database en/of openbaar gemaakt door middel van druk, fotokopie, microfilm, geluidsband, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van VDAB.

Hoewel deze syllabus met zeer veel zorg is samengesteld, aanvaardt VDAB geen enkele aansprakelijkheid voor schade ontstaan door eventuele fouten en/of onvolkomenheden in deze syllabus en of bijhorende bestanden.

Inhoud

Hoofdstuk 1.	Inleiding	5
1.1.	Algemene informatie.....	5
1.1.1.	Waarom?	5
1.1.2.	Werkwijze	5
Hoofdstuk 2.	Procedures en functies	7
2.1.	Inleiding	7
2.1.1.	Voorbeelden	7
2.1.2.	Hoe in Structorizer.....	10
2.2.	Procedures.....	10
2.2.1.	Intro	10
2.2.2.	Een voorbeeld.....	11
2.2.3.	Gebruik van parameters.....	11
2.2.4.	Meerdere parameters	12
2.2.5.	Syntax	13
2.2.6.	Soorten parameters	13
2.2.7.	Opgave: Examens3	15
2.3.	Functies	16
2.3.1.	Syntax	16
2.3.2.	Uittesten in Lazarus	17
2.3.3.	Opgave1: Examens4	17
2.3.4.	Opgave2: Faculteit2.....	17
2.4.	Recurisie	18
2.4.1.	Voorbeeld 1: Faculteit	18
2.4.2.	Wanneer?	21
2.4.3.	Opgave: Som van de cijfers	22
2.5.	Scope van variabelen.....	22
2.5.1.	Ter herhaling: een overzicht.....	22
2.5.2.	De scope van variabelen.....	23
2.5.3.	Waar declareren.....	24
2.5.4.	Voorbeeld 1: Met poppen spelen.....	24
2.5.5.	Voorbeeld 2: Examens4.....	25
2.5.6.	Voorbeeld 3: Einddatum berekenen	27

2.5.7.	Opgave - Bonus malus	29
2.6.	Opgaven.....	30
2.6.1.	Opgave 1: Rekeningnummer	30
2.6.2.	Opgave 2: Welke wagen?	30
2.6.3.	Opgave 3: Breuken vereenvoudigen	30
2.6.4.	Opgave 4: Perfecte getallen	30
2.6.5.	Opgave 5: Schrikkeljaar	31
2.6.6.	Opgave 6: Torens van Hanoi.....	31
2.6.7.	Opdracht voor de coach: Mastermind	31
Hoofdstuk 3.	Tot slot.....	33
3.1.	Einde cursus.....	33
3.1.1.	Eindoefening.....	33
3.1.2.	Wat nu?	33

Hoofdstuk 1. Inleiding

1.1. Algemene informatie

1.1.1. Waarom?

Waarom leren programmeren? De belangrijkste redenen op een rijtje:

Programmeren leert je een probleem-oplossende manier van denken aan. Je leert **analytisch denken**.

Als je kan programmeren kun je een hoop dingen **automatiseren** en ben je waarschijnlijk ook handig met andere ict-gerelateerde zaken.

Je leert werken met **gegevens**. Informatica is een studie over gegevens en informatie. Bij programmeren leer je hoe je met gegevens om moet gaan en wat je ermee kunt doen.

Websites maken is waardevol tegenwoordig. Bijna elk bedrijf of project heeft een **bijhorende site** die moet worden onderhouden (Javascript, PHP,...).

...

In deze cursus **programmatieloga** leer je gestructureerd programmeren, dwz dat je leert zelfstandig problemen op te lossen met de computer. Je gebruikt Nassi-Shneiderman diagrammen. Deze leggen de basis voor het echte programmeerwerk in één of andere taal.

De cursus programmaticaloga bestaat uit verschillende delen.

Het eerste deel, **Basisstructuren**, heb je nu achter de rug. Blijf wat je daar geleerd hebt ook in dit deel toepassen. We beschouwen dit als reeds verworven kennis.

In dit deel, **Procedures en functies**, leer je werken met procedures en functies.

Als je het onderdeel **Arrays en Strings** nog niet hebt doorgenomen, kan je dat na dit deel nog doen. Heb je dan nog steeds zin in meer, kan je nog verder verdiepen in de delen **Sorteren** en/of **Bestanden**.

De nadruk in alle delen ligt op het probleemoplossend denken. Na het tekenen van de diagrammen gaan we deze schema's ook omzetten naar echte programmacode om ze uit te testen.

1.1.2. Werkwijze

In deze cursus gebruiken we een tekentool om Nassi-Shneiderman diagrammen, PSD's of Programma Structuur Diagrammen, te tekenen (**Structorizer**). Je kan alle diagrammen ook maken met pen en papier. We gebruiken ook het programma **Lazarus** om onze code uit te testen. Het installeren en gebruiken van deze programma's werd reeds uitgelegd in het onderdeel Basisstructuren.

In deze cursus zijn heel wat oefeningen voorzien. Je kan ze onderverdelen in twee categorieën:

Gewone oefeningen:

Dit is het elementaire oefenmateriaal dat noodzakelijk is om de leerstof onder de knie te krijgen. Bij deze oefeningen kan je ook altijd een modeloplossing (van het PSD) terug vinden.

Opdrachten voor de coach:

Per hoofdstuk is er een opdracht voor de coach voorzien. Deze kan als 'test' voor dat hoofdstuk dienen. Dit is een samenvattende oefening van de voorgaande leerstof. Voor deze opdrachten zijn er geen modeloplossingen voorzien.

Als je deze cursus volgt binnen een **competentiecentrum**, spreek je met je **instructeur** af hoe de opdrachten geëvalueerd worden.

Anders stuur je je oplossingen van de *opdrachten voor de coach* door aan je **coach** ter verbetering. Als je een probleem of vraag hebt over één van de gewone oefeningen, kan je ook bij je coach terecht. Stuur steeds zowel het .nsd-bestand als het .pas-bestand dat je gemaakt hebt door. Dit maakt het voor je coach makkelijker om je vraag snel te beantwoorden.

Let op!

Het is niet de bedoeling om met Structorizer te leren werken, wel om een probleem te leren analyseren en in een gestructureerd diagram weer te geven. Structorizer en Lazarus zijn enkel hulpmiddelen om het tekenen en uittesten van die diagrammen te vereenvoudigen.

Hoofdstuk 2. Procedures en functies

2.1. Inleiding

Als je in je programma met alle mogelijkheden rekening wil houden en alle mogelijke problemen wil opvangen, wordt je programma al snel erg complex.

Er kunnen zich een aantal problemen voordoen:

Je programma wordt al snel onoverzichtelijk en moeilijk te lezen.

Ook kom je wel eens voor een moeilijke keuze te staan: wat krijgt voorrang, de efficiëntie of de structuur van je programma.

Soms merk je dat een bepaald stukje code op verschillende momenten wordt herhaald in je programma óf zelfs binnen veel verschillende programma's.

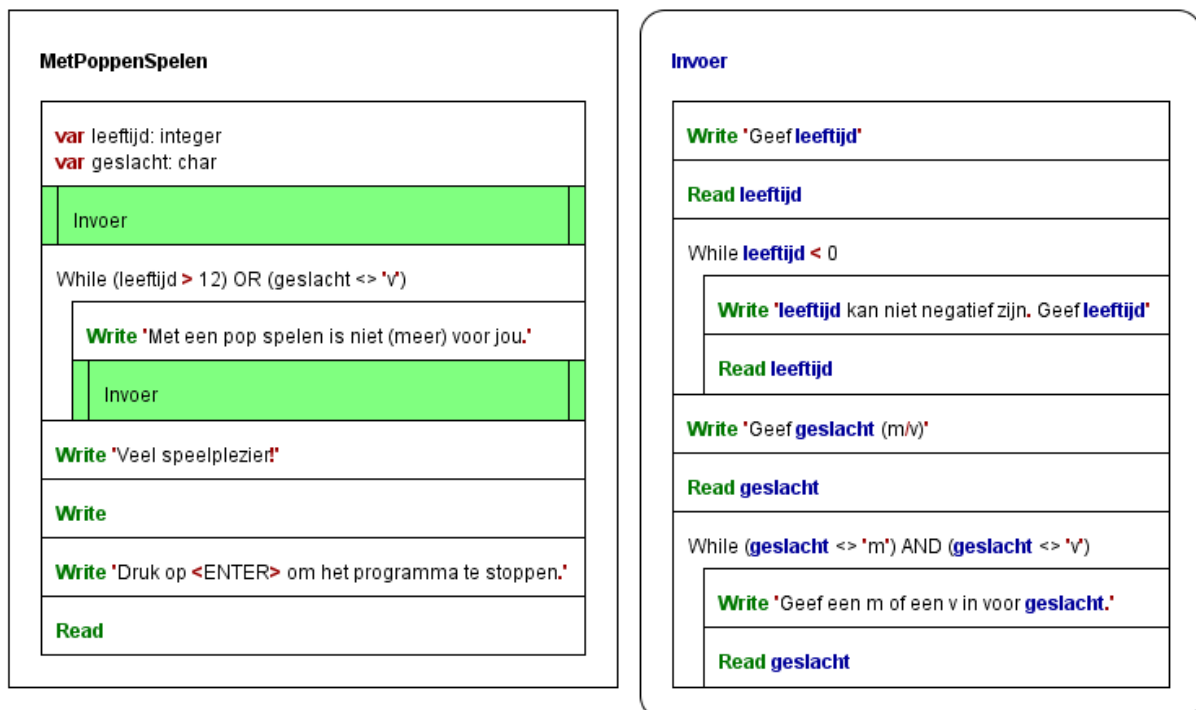
Denk bv. aan het controleren of een e-mailadres geldig is.

Je kan daarom je probleem gaan opsplitsen in deelproblemen, of deel-PSD's: we noemen dat **procedures**. Je eerste schema blijft je globale PSD, maar daarnaast komen een aantal aparte PSD's (die elk een naam krijgen) die je binnen het globale PSD gaat aanroepen.

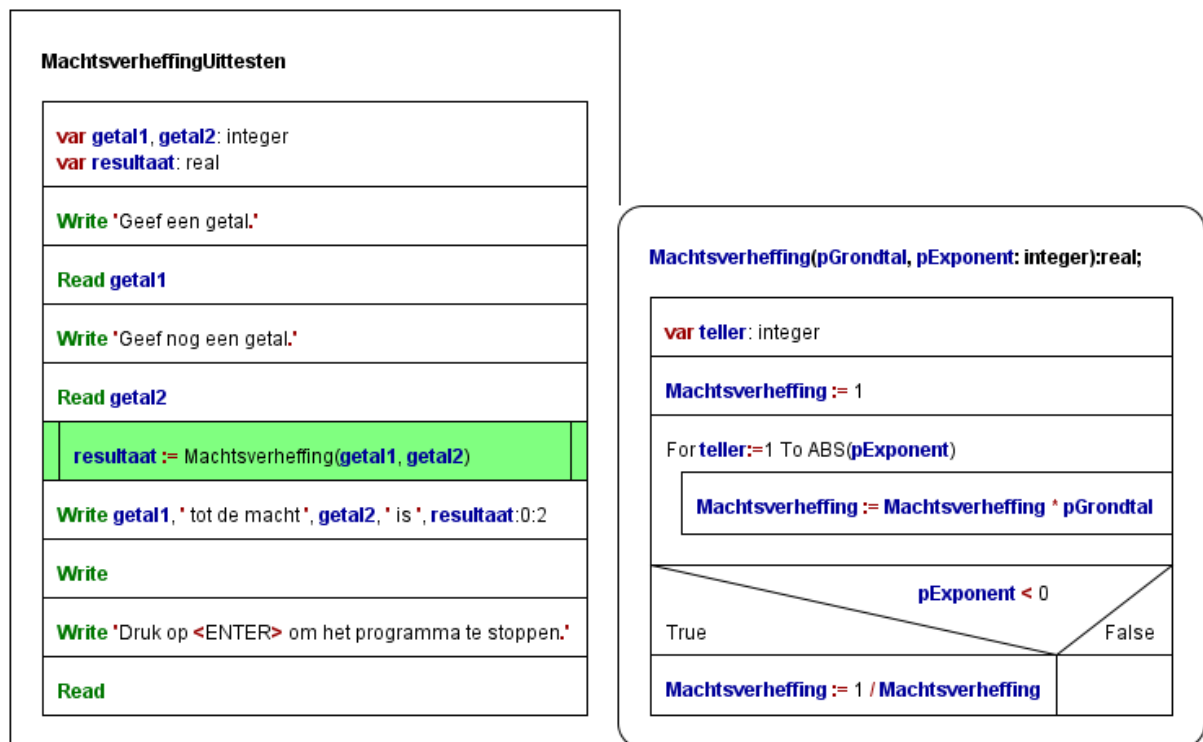
2.1.1. Voorbeelden

Hieronder worden drie voorbeelden getoond om aan te geven wanneer het interessant is om gebruik te maken van procedures en functies.

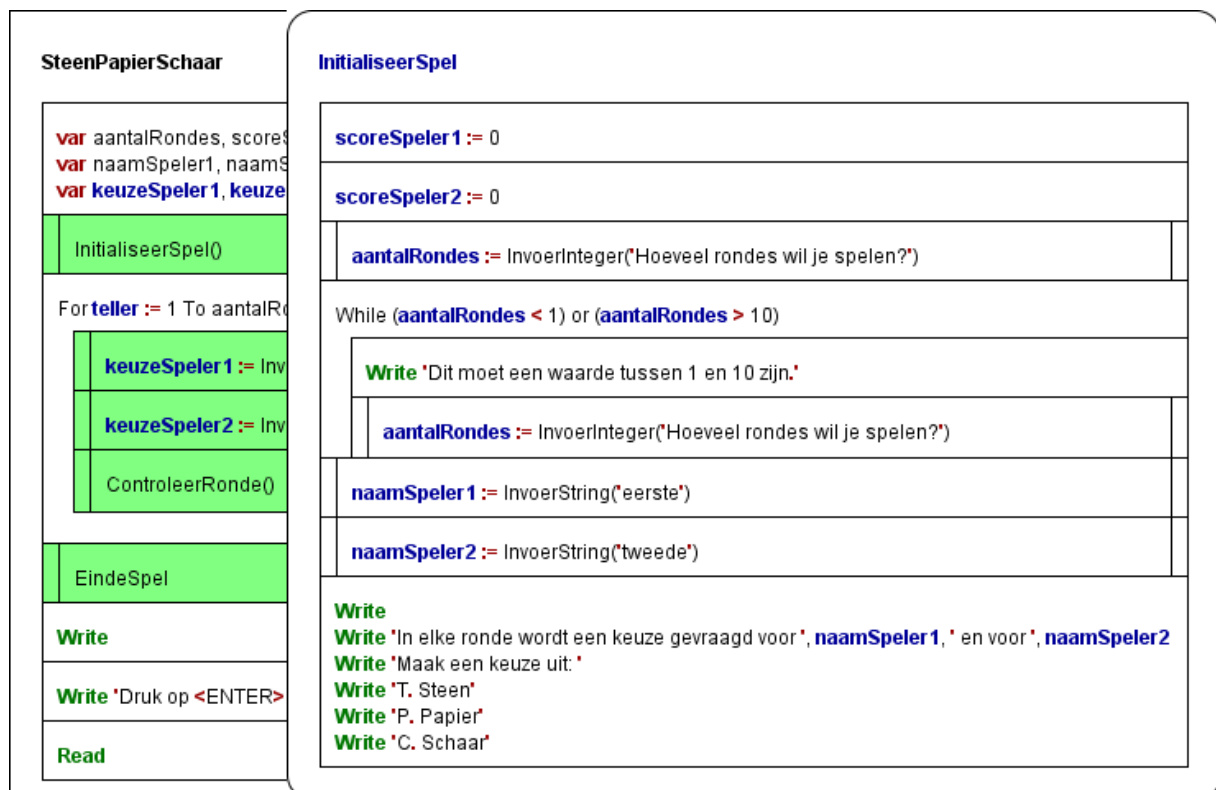
Om te zorgen dat je code niet op verschillende plaatsen moet herhalen



Om dezelfde logica vanuit verschillende programma's te kunnen oproepen



Om je programma of code overzichtelijker te maken



SteenPapierSchaar

```

var aantalRondes, scoreSpeler1, scoreSpeler2, teller: integer
var naamSpeler1, naamSpeler2: string
var keuzeSpeler1, keuzeSpeler2: char

```

```

InitialiseerSpel()

```

```

For teller := 1 To aantalRondes

```

```

    keuzeSpeler1 := InvoerKeuze(naamSpeler1)

```

```

    keuzeSpeler2 := InvoerKeuze(naamSpeler2)

```

```

    ControleerRonde()

```

```

EindeSpel

```

```

Write

```

```

Write "Druk op <ENTER> om het programma te beëindigen"

```

```

Read

```

InvoerKeuze(pNaamSpeler:string):char

```

var keuze:char

```

```

Write "Keuze" + pNaamSpeler + "?"

```

```

Read keuze

```

```

While (keuze <> "T") and (keuze <> "P") and (keuze <> "C")

```

```

    Write "Ongeldige waarde. Kies uit T, P of C"

```

```

    Write "Keuze" + pNaamSpeler + "?"

```

```

    Read keuze

```

```

InvoerKeuze := keuze

```

SteenPapierSchaar

```

var aantalRondes, scoreSpeler1, scoreSpeler2, teller: integer
var naamSpeler1, naamSpeler2: string
var keuzeSpeler1, keuzeSpeler2: char

```

```

InitialiseerSpel()

```

```

For teller := 1 To aantalRondes

```

```

    keuzeSpeler1 := InvoerKeuze(naamSpeler1)

```

```

    keuzeSpeler2 := InvoerKeuze(naamSpeler2)

```

```

    ControleerRonde()

```

```

EindeSpel

```

```

Write

```

```

Write "Druk op <ENTER> om het programma te beëindigen"

```

```

Read

```

ControleerRonde

```

True

```

```

Write "Gelijkspel: jullie kozen allebei hetzelfde."

```

```

"P"

```

```

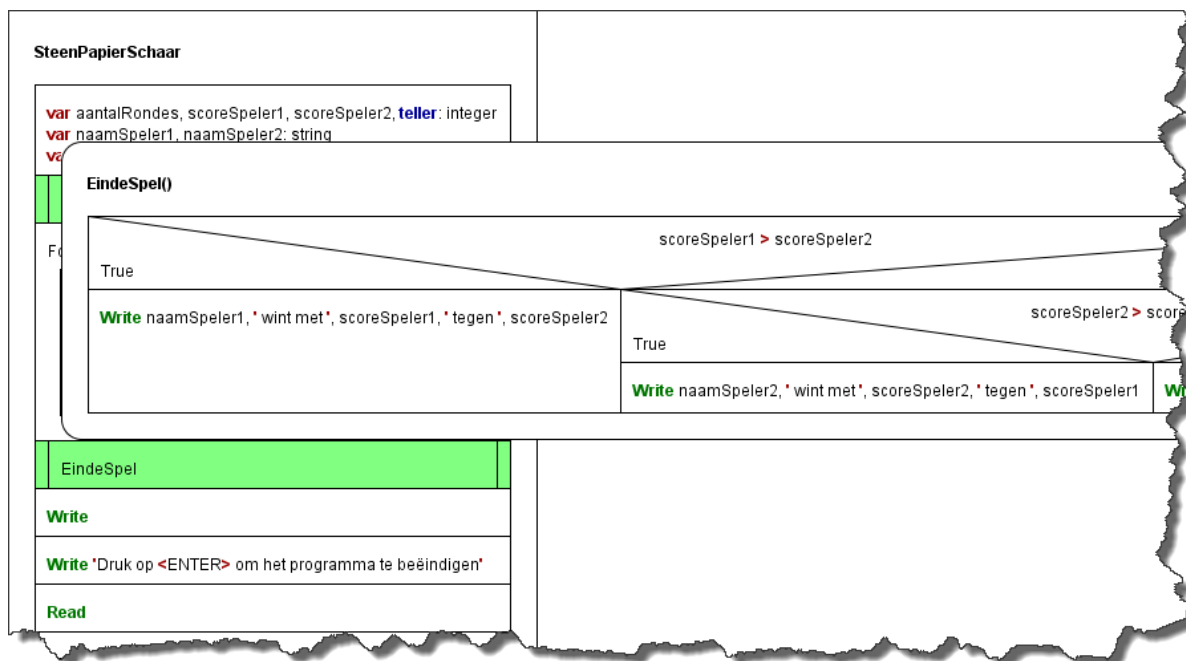
scoreSpeler2 := scoreSpeler2 + 1

```

```

Write naamSpeler2, " wint. Het papier bedekt de steen."

```



In deze voorbeelden kan je al een eerste verschil zien tussen een procedure en een functie.

Een functie geeft een waarde terug en deze kan dan opgeslagen worden in een variabele. Zo zie je in het tweede voorbeeld dat het resultaat van de functie **Machtsverheffing** opgeslagen wordt in de variabele **resultaat**.

2.1.2. Hoe in Structorizer



In Structorizer kan je een subroutine herkennen aan de afgeronde hoeken. Met de knop **Sub diagram** kan je voor die afgeronde hoeken zorgen.

Het wijzigen in een procedure is vooral belangrijk voor de omzetting naar Pascal. Met ronde hoeken krijg je in Pascal *function* ipv *Program*.

Voor elke subroutine moet je een apart psd-bestand maken.



Met de knop **Insert a new call after the selected element** kan je een call-instructie toevoegen.

Voor de omzetting naar Pascal maakt dit geen verschil. In je PSD zie je wel duidelijk het verschil tussen een gewone instructie en het oproepen van een procedure of functie.

2.2. Procedures

2.2.1. Intro

In dit onderdeel bekijken we hoe je **procedures** kan maken en gebruiken.

Hiervoor is het van belang dat je begrijpt wat **parameters** zijn en welke **verschillende soorten** parameters er zijn.

2.2.2. Een voorbeeld

Stel dat je in een programma regelmatig scheidingslijntjes wil tekenen, om twee stukken uitvoer van mekaar te scheiden. Je kan hiervoor gewoon een reeks gelijkheidstekens (=) of mintekens (-) of iets dergelijks afdrukken. Hiervoor zou je een procedure **TekenLijn** kunnen maken:



In je hoofd-PSD kan je deze procedure aanroepen door gewoon de naam van de procedure als opdracht te geven:



2.2.3. Gebruik van parameters

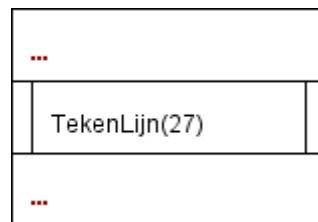
Het nadeel hieraan is dat deze lijn altijd even lang is. Nu zou het nuttig kunnen zijn ervoor te zorgen dat die lengte variabel wordt. De handigste manier van werken is dat je in de aanroep van de procedure een parameter meegeeft: extra informatie die de procedure gaat gebruiken bij de uitvoering.

Je zou je procedure dan kunnen schrijven als:



In het voorschrift van je procedure wordt tussen ronde haakjes een parameter (**pLengte**) aangegeven. Deze wordt in de body van de procedure gebruikt.

Het aanroepen van deze procedure gebeurt weer door de naam van de procedure op te geven, maar nu moet je tussen haakjes de parameter(s) meegeven:



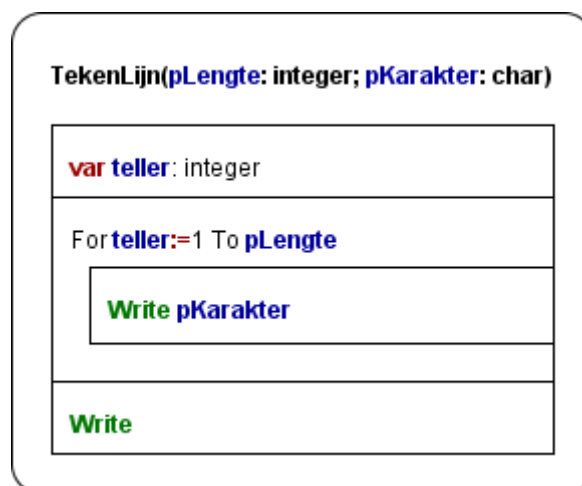
Er zal een lijn getekend worden van 27 gelijkheidstekens (=).

2.2.4. Meerdere parameters

Je kan ook procedures maken die meerdere parameters gebruiken.

Het **gegevenstype** van je parameters, het **aantal** parameters dat je gebruikt en ook de **volgorde** van je parameters in de aanroep moet altijd overeenkomen met de definitie van je procedure!

Je zou voorgaande procedure bijvoorbeeld kunnen uitbreiden met een tweede parameter, nl. het karakter waarmee de lijn getekend gaat worden:



In je hoofd-PSD kan je deze procedure aanroepen als volgt:



De eerste keer wordt er een lijn getekend van 27 sterren (*).

De tweede keer zal er een lijn getekend worden van 14 mintekens (-).

Zo maak je zowel de lengte als het symbool voor het tekenen van de lijn variabel.

2.2.5. Syntax

De definitie van je procedure volgt de volgende syntax:

```
<NaamProcedure>(<pNaamPar1>[,<pNaamPar2>:<gegevenstype>[:<pNaamPar3>:<gegevenstype>])
```

Een procedure met twee parameters met een verschillend gegevenstype (let op de *puntkomma*):

TekenLijn(**pLengte**: integer; **pKarakter**: char)

Als je verschillende parameters na elkaar hebt met hetzelfde gegevenstype (let op de *komma*):

SchrijfTitels(**pTitel**, **pSubtitel**: string)

2.2.6. Soorten parameters

Er bestaan twee verschillende soorten parameters die je kan doorgeven aan een procedure:

Invoer-parameters (**call by value**)

In/uitvoer-parameters (**call by reference**)

Bij een **invoer**-parameter wordt een kopie van de inhoud van de variabele in het hoofdprogramma doorgegeven aan de sub-procedure. Wanneer je terug in het hoofdprogramma komt, is **de waarde van de variabele daar niet gewijzigd**.

Bij een **in/uitvoer**-parameter wordt de variabele zelf doorgegeven aan de sub-procedure (in de praktijk: het geheugenadres waar de variabele zich bevindt). Wanneer je terug in het hoofdprogramma komt, is **de waarde van de variabele daar wel gewijzigd**.

Voorbeeld

Om het verschil tussen beide soorten parameters duidelijk te maken: een kleine vergelijking. Dezelfde procedure maken we twee keer: één keer met een call by reference en één keer met een call by value.

Bekijk volgend programma:

UittestenCallBy

```

var getal: integer

Write "Geef getal"

Read getal

VerdubbelValue(getal)

Write "Waarde na value: ", getal

VerdubbelReference(getal)

Write "Waarde na reference: ", getal

Write

Write "Druk op <ENTER> om het programma te verlaten"

Read

```

VerdubbelValue(pGetal: integer)

```

pGetal := pGetal * 2

Write "pGetal: ", pGetal

```

Je ziet maar één verschil tussen de twee procedures, nl. het woordje 'var' voor de parameter.

Dit woordje bepaalt of de parameter wel of niet een referentie is tot de variabele in je hoofdprogramma.

VerdubbelReference(var pGetal: integer)

```


pGetal := pGetal * 2

Write "pGetal: ", pGetal

```

Mogelijke uitvoer

Hieronder een voorbeeld van mogelijke uitvoer voor dit programma:



```

Geef getal
5
pGetal: 10
Waarde na value: 5
pGetal: 10
Waarde na reference: 10

Druk op <ENTER> om het programma te verlaten

```

Bij een **call by value** wordt de inhoud van dat geheugenadres (een kopie dus van de oorspronkelijke variabele) doorgegeven. Wanneer **pGetal** wordt gewijzigd, heeft dat geen enkele invloed op **getal**. Vandaar dat de vijfde regel van het hoofd-PSD als uitvoer "5" oplevert.

Bij een **call by reference** wordt een verwijzing naar het geheugenadres doorgegeven. Wijzigen we in de subprocedure de variabele **pGetal** (verdubbeling), dan betekent dat dat in de hoofdprocedure ook de variabele **getal** (die immers naar hetzelfde geheugenadres verwijst) gewijzigd is. Vandaar dat de zevende regel van het hoofd-PSD de uitvoer “10” oplevert.

Uittesten in Lazarus

Om je programma uit te testen in Lazarus, moet je een paar extra stappen doen:

Exporteer elk PSD naar een Pascal/Delphi bestand.

Open je hoofdprogramma als een nieuw Lazarus project.

Verplaats je variabele-declaratie.

Voor elke subprocedure:

- Open de procedure als een normale bron.
Dit betekent dat je Nee moet antwoorden als Lazarus vraagt een nieuw project te starten.
- Wijzig het woordje *function* in *procedure*.
- Verplaats indien nodig je variabele-declaratie.
- Vervang het puntje op het einde door een punt-komma.
- Kopieer alle code en plak deze meteen na de variabelen-declaratie van je hoofdprogramma.

Probeer dit uit voor het programma **UittestenCallBy**.

In de webcursus vind je een animatie waarin de verschillende stappen getoond worden.

2.2.7. Opgave: Examens3

We hernemen nogmaals de oefening Examens.

Lees de examenuitslagen in voor drie vakken (wiskunde, boekhouden en informatica). Elk van de vakken staat op 10 punten.

Voeg de validatie toe bij het inlezen van de punten, zodat je punten kan blijven ingeven totdat je een juiste waarde hebt voor de punten van wiskunde, boekhouden en informatica.

De student is geslaagd indien hij/zij voor wiskunde minstens 6/10 haalt, en voor boekhouden en informatica samen minstens 12/20.

Toon op het scherm of de student geslaagd is, en indien de student niet geslaagd is, toon je ook de reden daarvoor.

Schrijf en gebruik een procedure met parameters voor het inlezen en valideren van de punten voor de drie vakken (wiskunde, boekhouden en informatica).

Denk eraan dat je in je hoofdprogramma nog verder wilt kunnen werken met de ingelezen punten.

In de webcursus vind je de oplossing en een animatie waarin de verschillende stappen getoond worden.

2.3. Functies

Je kan ook procedures maken die een resultaatwaarde teruggeven. Dan spreken we van **functies** ipv procedures.

In de meeste programmeertalen zitten al een aantal functies ingebouwd.

Enkele voorbeelden:

De functie `LEN()` geeft als resultaat de lengte van de opgegeven string
bijv.

```
lengte := LEN("vdab okee!")
```

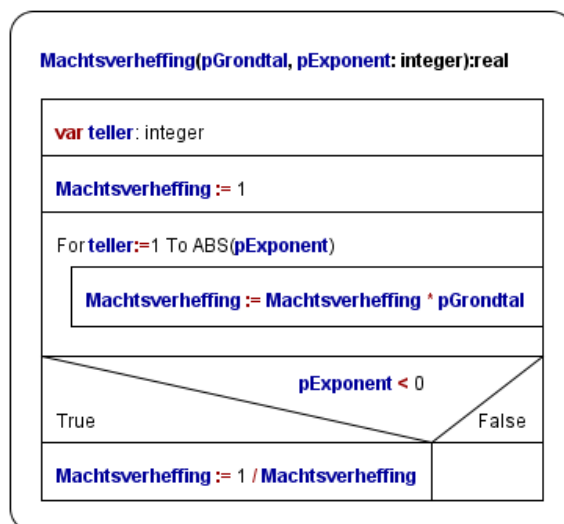
waarna `lengte` = 10

De functie `STR()` geeft als resultaat een string die overeenkomt met het opgegeven getal
bijv.

```
tekst := STR(1025)
```

waarna `tekst` = "1025"

Wanneer je zelf functies maakt, dan moet je ergens in de body van die functie opgeven welke waarde als resultaat moet worden gegeven. Dat doe je door de waarde toe te kennen aan een variabele met dezelfde naam als de functie.



2.3.1.Syntax

De definitie van je functie volgt de volgende syntax:

<NaamFunctie>(<pNaamPar1>[,<pNaamPar2>:<gegevenstype>[;<pNaamPar3>:<gegevenstype>]):gegevenstype

Je geeft de functie zelf dus ook een gegevenstype.

Machtsverheffing(pGrondtal, pExponent: integer):integer

2.3.2. Uittesten in Lazarus

Ga als volgt te werk:

Exporteer elk PSD naar een Pascal/Delphi bestand

Open in Lazarus het hoofdprogramma als een nieuw project.

Verplaats je variabelen-declaratie.

Voor elke functie:

- Open de functie als een normale bron.
(Nee voor nieuw project)
- Verplaats indien nodig je variabelen-declaratie.
- Vervang het puntje op het einde door een punt-komma.
- Kopieer alle code en plak het meteen na de variabelen-declaratie van je hoofdprogramma.

Probeer dit uit voor het programma **MachtsverheffingUittesten**.

In de webcursus worden de verschillende stappen getoond in een animatie.

2.3.3. Opgave1: Examens4

We hernemen nogmaals de oefening Examens.

Lees de examenuitslagen in voor drie vakken (wiskunde, boekhouden en informatica). Elk van de vakken staat op 10 punten.

Voeg de validatie toe bij het inlezen van de punten, zodat je punten kan blijven ingeven totdat je een juiste waarde hebt voor de punten van wiskunde, boekhouden en informatica.

De student is geslaagd indien hij/zij voor wiskunde minstens 6/10 haalt, en voor boekhouden en informatica samen minstens 12/20.

Toon op het scherm of de student geslaagd is, en indien de student niet geslaagd is, toon je ook de reden daarvoor.

Schrijf en gebruik een **functie** voor het inlezen en valideren van de punten voor de drie vakken (wiskunde, boekhouden en informatica).

In de webcursus vind je de oplossing en een animatie waarin de verschillende stappen getoond worden.

2.3.4. Opgave2: Faculteit2

We hernemen nogmaals de oefening Faculteit.

De faculteit van een geheel getal is het product van alle gehele getallen van 1 t.e.m. het getal zelf.

Bijv: $5! = 1 * 2 * 3 * 4 * 5 = 120$.

Opgelet: de faculteit van 0 (0!) is 1, en de faculteit van een negatief getal bestaat niet.

Maak een programma dat een getal inleest, controleert of het een positief geheel getal is (0 hoort bij de positieve getallen), en indien nodig de invoer blijft herhalen en tenslotte de faculteit van dat getal berekent en afdruckt.

Schrijf en gebruik een **functie** voor het berekenen van de faculteit.

De oplossing vind je in de webcursus.

2.4. Recursie

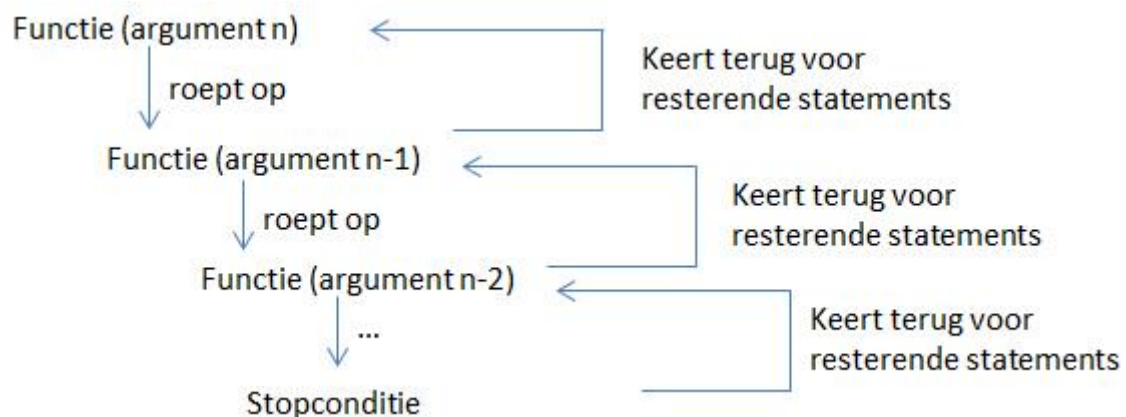
Recursieve functies of procedures zijn functies/procedures die zichzelf terug aanroepen.

Zoals deze omschrijving al doet vermoeden kan dit een oneindige lus veroorzaken: je moet dus wel voorzien dat vanaf een bepaald moment de functie NIET langer zichzelf aanroept. Eén keer moet het eindigen. Belangrijk is dus de voorwaarde waarbij de functie zichzelf oproept!

Bij een recursieve functie moet steeds een **stopconditie** ingebouwd worden!

Na de stopconditie wordt de recursie nog verder afgewerkt: de statements die volgen na de aanroep van "zichzelf" worden in omgekeerde volgorde nog x keer (het aantal keer dat de functie zichzelf heeft aangeroepen) uitgevoerd (met de waarden die de variabelen op het moment van de aanroep hadden).

Schematisch:



Voorbeelden zullen dit verder toelichten.

2.4.1. Voorbeeld 1: Faculteit

Een eenvoudig voorbeeld is een algoritme om de faculteit van een getal te berekenen.

Bijv. $5! = 1 * 2 * 3 * 4 * 5 = 120$

Opgelet: $0! = 1$

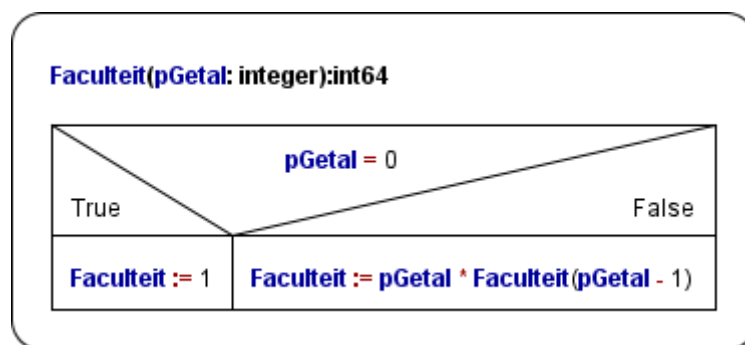
We maakten hiervoor reeds een algoritme in één van de voorgaande oefeningen, waarbij we een iteratie gebruikten, bijv.:



Deze functie berekent de faculteit van het opgegeven getal en geeft dat als resultaat terug. In je hoofd-PSD kan je dan bijvoorbeeld volgende instructie opgeven:

```
Write getal, '!= ', Faculteit(getal)
```

Dezelfde functie kan ook als een recursief algoritme worden geprogrammeerd: de faculteit van een getal x is immers altijd gelijk aan dat getal x vermenigvuldigd met de faculteit van $(x-1)$. (Uitzondering: het getal 0 heeft als faculteit 1.)



De recursieve functie gaat dus controleren of de invoerwaarde 0 is; zo ja, dan is de faculteit gelijk aan 1; zo neen, dan is de faculteit gelijk aan $\text{getal} * \text{faculteit}(\text{getal}-1)$.

In je hoofd-PSD kan je net dezelfde instructie opgeven als bij het niet-recursieve algoritme, en dit zal ook precies hetzelfde resultaat opleveren:

```
Write getal, '!= ', Faculteit(getal)
```

Wiskundige opbouw

Stel dat vanuit het hoofdprogramma de functie wordt aangeroepen met cijfer 0:

Faculteit(0)

Dan krijgt in de functie **Faculteit** de variabele **pGetal** de waarde 0.

0 is gelijk aan 0, dus geeft deze functie de waarde 1 terug. De functie wordt dus éénmalig uitgevoerd en er is geen sprake van recursie.

Stel nu dat vanuit het hoofdprogramma de functie wordt aangeroepen met cijfer 5:

Faculteit(5)

Dan krijgt in de functie **Faculteit** de variabele **pGetal** de waarde 5.

5 is niet gelijk aan 0, dus geeft deze functie de waarde terug van $5 * \text{Faculteit}(4)$. Doch in deze berekening zit opnieuw de aanroep van de functie **Faculteit** met waarde 4. De vermenigvuldiging van $5 * \text{Faculteit}(4)$ kan nog niet uitgevoerd worden vermits het resultaat van **Faculteit(4)** nog niet gekend is, het is immers opnieuw een aanroep van zichzelf.

Dus variabele **pGetal** krijgt waarde 4. 4 is niet gelijk 0 en ook deze functie geeft de waarde terug van $4 * \text{Faculteit}(3)$. Ook dit kan nog niet uitgerekend worden, want het is opnieuw een aanroep van zichzelf met getal 3.

Dus variabele **pGetal** krijgt waarde 3. 3 is niet gelijk 0 en ook deze functie geeft de waarde terug van $3 * \text{Faculteit}(2)$. Ook dit kan nog niet uitgerekend worden, want het is opnieuw een aanroep van zichzelf met getal 2.

Zo gaat dit verder ...totdat de functie **Faculteit** wordt aangeroepen met getal 0. Dit is de STOP van het aanroepen van zichzelf. Resultaat van deze aanroep met getal 0 is 1 ($0! = 1$).

Eigenlijk is dus volgende wiskundige berekening opgebouwd:

$$\begin{aligned} &5 * \text{Faculteit}(4) \\ &= 5 * (4 * \text{Faculteit}(3)) \\ &= 5 * (4 * (3 * \text{Faculteit}(2))) \\ &= 5 * (4 * (3 * (2 * \text{Faculteit}(1)))) \\ &= 5 * (4 * (3 * (2 * (1 * \text{Faculteit}(0))))) \end{aligned}$$

Stappen bij het doorlopen van het PSD

Terug naar ons PSD:

De recursie stopt wanneer de functie **Faculteit** wordt aangeroepen met getal 0.

Deze functie geeft dan waarde 1 terug:

Faculteit := 1

In omgekeerde volgorde wordt nu, per keer dat de functie **Faculteit** is opgeroepen, de rest van deze functie afgewerkt.

De returnwaarde 1 van de oproep van **Faculteit** met getal 0 wordt gebruikt om de vorige oproep van de recursie af te werken nl. waar getal de waarde 1 had: deze functie geeft terug

Faculteit := $1 * \text{Faculteit}(0)$, dus $1 * 1 = 1$.

Opnieuw wordt dit resultaat gebruikt om de recursieve oproep van daarvóór af te werken, dus waar **getal** de waarde 2 had: deze functie geeft terug

Faculteit := 2 * **Faculteit**(1) of 2 * 1 = 2.

Opnieuw wordt dit resultaat gebruikt om de recursieve oproep van daarvóór af te werken, dus waar **pGetal** de waarde 3 had: deze functie geeft terug

Faculteit := 3 * **Faculteit**(2) of 3 * 2 = 6.

Zo gaat dit verder, dus

Faculteit := 4 * **Faculteit**(3) of 4 * 6 = 24

En tot slot de laatste recursie-oproep

Faculteit := 5 * **Faculteit**(4) of 5 * 24 = 120.

In omgekeerde volgorde wordt dus steeds de rest van de functie **Faculteit** afgewerkt.

Vervolg wiskundige opbouw

Nog even terug naar de laatste stap van de wiskundige opbouw:

$5 * (4 * (3 * (2 * (1 * \text{Faculteit}(0)))))$

Wanneer je dit uitwerkt volgens de wiskundige regels (eerst de haken), dan geeft dit:

$5 * (4 * (3 * (2 * (1 * \text{Faculteit}(0)))))$, **Faculteit** van 0 = 1, dus
 $= 5 * (4 * (3 * (2 * (1 * 1))))$
 $= 5 * (4 * (3 * (2 * 1)))$
 $= 5 * (4 * (3 * 2))$
 $= 5 * (4 * 6)$
 $= 5 * 24$
 $= 120$

2.4.2. Wanneer?

Wanneer maak je nu goed gebruik van dit mechanisme? Om deze vraag te beantwoorden plaatsen we de eigenschappen van recursie tegenover die van iteratie.

De **voordelen** van recursie t.o.v. iteratie zijn:

- vaak korte code
- soms onvermijdbaar
- indien het algoritme gekend is, relatief gemakkelijk te coderen.

De **nadelen** van recursie t.o.v. iteratie zijn:

- moeilijk om een recursieve eigenschap te ontdekken en om te zetten naar een algoritme

aanslag op het geheugen (voor de afwerking van de recursie in omgekeerde volgorde, waardoor vele tussenwaardes van variabelen dienen bewaard te worden) en daardoor: trage(re) uitvoering.

Het antwoord op bovenstaande vraag kan beter gezien worden als een vuistregel:

Recursie gebruik je wanneer een iteratieve oplossing te ingewikkeld of onmogelijk is. In alle andere gevallen maak je best gebruik van iteratieve oplossingen!

In de module 'Programmatielogica - Sorteren' zullen we een toepassing zien van recursie bij het quicksort-algoritme.

Het is zeker niet altijd eenvoudig om de recursieve eigenschap van een probleem te ontdekken en in een algoritme om te zetten!

Belangrijk is om het **begrip** recursie te kennen!

2.4.3. Opgave: Som van de cijfers

Schrijf een recursieve functie voor het maken van de som van de cijfers van een getal.

Bijv.:

-1025: Som van de cijfers = $1 + 0 + 2 + 5 = 8$
 925: Som van de cijfers = $9 + 2 + 5 = 16$
 18 : Som van de cijfers = $1 + 8 = 9$
 7 : Som van de cijfers = 7

Uiteraard roep je dit ook op via een hoofdprogramma.

De oplossing van deze opgave vind je inde webcursus.

2.5. Scope van variabelen

In dit onderdeel gaan we verder in op het begrip **scope** (geldigheidsgebied) van een variabele.

Als we spreken van scope, hebben we het over de levensduur en de toegankelijkheid van een variabele.

2.5.1. Ter herhaling: een overzicht

Nog even ter herhaling en om een overzicht te hebben:

Wanneer gebruik je een functie en wanneer een procedure?

- een **procedure** gebruik je om
 - een groot programma op te delen in handelbare stukjes
Door logische namen te geven aan je procedures, maakt dit je programma een stuk overzichtelijker.

- een reeks opdrachten die meermaals voorkomt slechts één keer te moeten schrijven
Hierdoor is je code ook makkelijker te onderhouden. Je moet aanpassingen nog maar op één plaats doen.
- die reeks opdrachten variabel te maken
Kleine verschillen in je code kan je wegwerken door gebruik te maken van parameters.
- een **functie** gebruik je voor dezelfde redenen als een procedure. Een functie zal bij de uitvoering exact één resultaatwaarde teruggeven waarmee later verder gewerkt kan worden.
- **recursie** gebruik je als
 - het probleem niet opgelost kan worden met een iteratie of
 - de oplossing met een iteratie complex en moeilijk leesbaar wordt

2.5.2. De scope van variabelen

Bij het programmeren maak je gebruik van verschillende soorten variabelen. Hun zichtbaarheid en levensduur is hiervan afhankelijk.

- **globale variabelen**
zichtbaarheid: doorheen het hele programma en binnen alle onderliggende subroutines
levensduur: zolang het programma draait
- **lokale variabelen**
Zichtbaarheid: enkel binnen de subroutine
Levensduur: ontstaan bij de aanroep van de routine en verdwijnen bij het verlaten van de routine
- **Parameters by value.**
Deze parameters zijn eveneens lokale variabelen. De inhoud van deze variabelen is een kopie van de inhoud van de variabelen die je doorgeeft. Dus aan de originele variabelen wordt niet geraakt.
Zichtbaarheid: enkel binnen de subroutine
Levensduur: ontstaan bij de aanroep van de routine en verdwijnen bij het verlaten van de routine
- **Parameters by reference.**
Zichtbaarheid: enkel binnen de subroutine
Levensduur: Deze parameters zijn eigenlijk verwijzingen naar andere variabelen. Ze worden ook wel referencevariabelen genoemd. Je werkt met de originele variabelen die je doorgeeft, dus wijzigingen worden in de originele variabelen doorgevoerd.

- **Functies**

De functie op zich is ook een lokale variabele

Zichtbaarheid: enkel binnen de subroutine

Levensduur: ontstaat bij het aanroepen van de functie en verdwijnt na het doorgeven van de waarde.

2.5.3. Waar declareren

Je bepaalt over welk type variabele het gaat, door het op de juiste plaats te declareren.

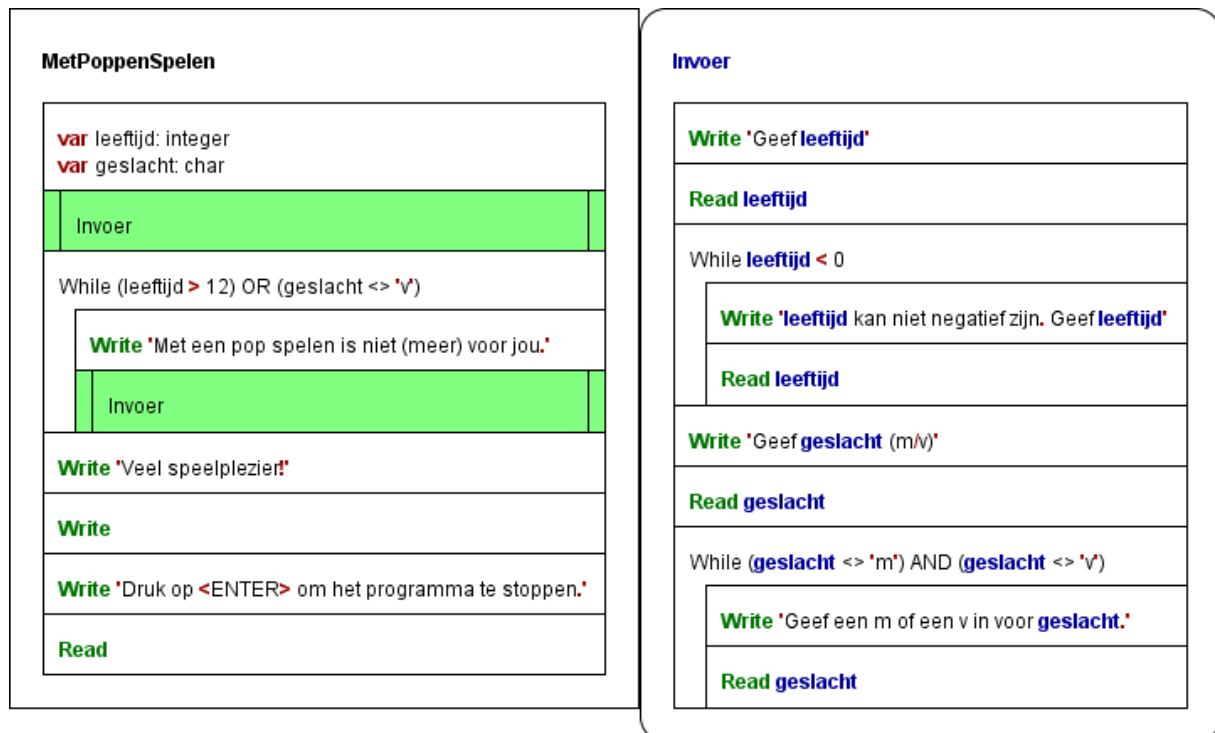
- **Globale variabelen** zet je bovenaan onder program.
- **Lokale variabelen** plaats je onder procedure of functie van die subroutine.
- **Parameters by value** plaats je in de definitie van je subroutine, tussen de haakjes na de naam.
- **Parameters by reference** plaats je eveneens in de definitie van je subroutine, voorafgegaan door het woordje var.
- **Functies:** de definitie van je functie is op zich een declaratie, doordat je er een gegevenstype aan toekent.

Het is belangrijk dat je altijd je scope/bereik zo klein mogelijk houdt. Dit om twee redenen:

- Je loopt niet het risico dat een variabele (al dan niet opzettelijk) voor de verkeerde doeleinden gebruikt wordt.
Het helpt je om je programma op een duidelijke en logische manier in elkaar te steken.
- Door de levensduur van je variabelen te beperken, gebruik je tijdens de uitvoer van je programma zo weinig mogelijk geheugenruimte.

2.5.4. Voorbeeld 1: Met poppen spelen

We hernemen een voorbeeld uit de inleiding: MetPoppensSpelen.



Dit is een perfect voorbeeld om het gebruik van **globale** variabelen te illustreren.

Je declareert de variabelen **leeftijd** en **geslacht** in het hoofdprogramma:

```

var leeftijd: integer
var geslacht: char

```

In de subroutine kan je deze variabelen rechtstreeks aanspreken:

```

Read leeftijd
Read geslacht

```

In het hoofdprogramma kan je op de ingelezen waardes testen:

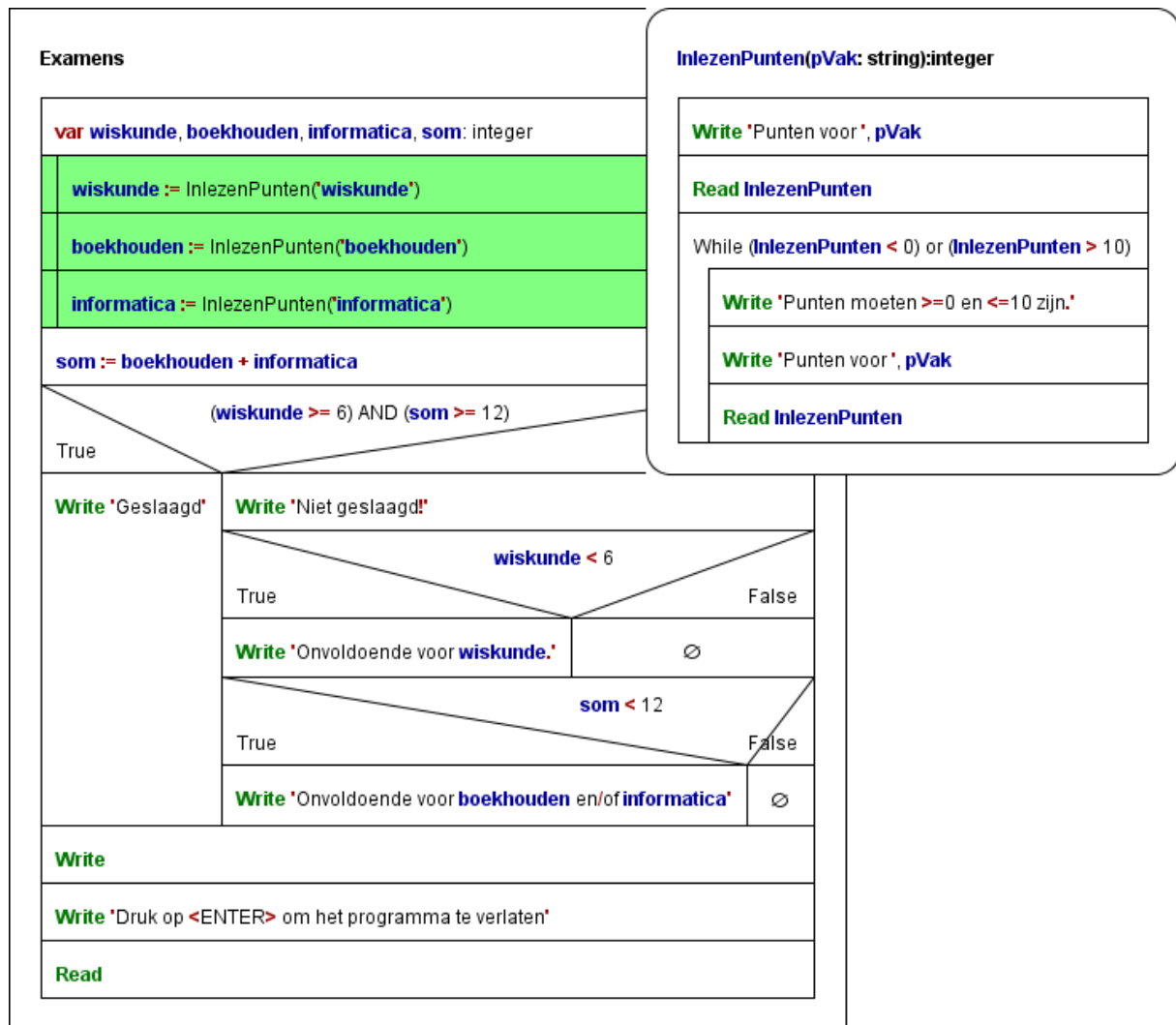
```

While (leeftijd > 12) OR (geslacht <> 'v')

```

2.5.5. Voorbeeld 2: Examens4

We hernemen Examens4, waarbij we een functie gebruiken om de cijfers op te vragen.



Hier zie je verschillende soorten variabelen.

Eerst en vooral hebben we de parameter by value **pVak**.

pVak: string

Aangezien parameters lokale variabele zijn is de zichtbaarheid en de levensduur beperkt tot de routine. Deze wordt aangemaakt bij het aanroepen van de functie en verdwijnt zodra de functie eindigt.

Het gaat hier om een functie dus heb je een tweede lokale variabele, nl. **InlezenPunten**.

InlezenPunten(...): integer

Ook deze variabele wordt aangemaakt bij het aanroepen van de functie en ze verdwijnt weer zodra de waarde teruggegeven wordt.

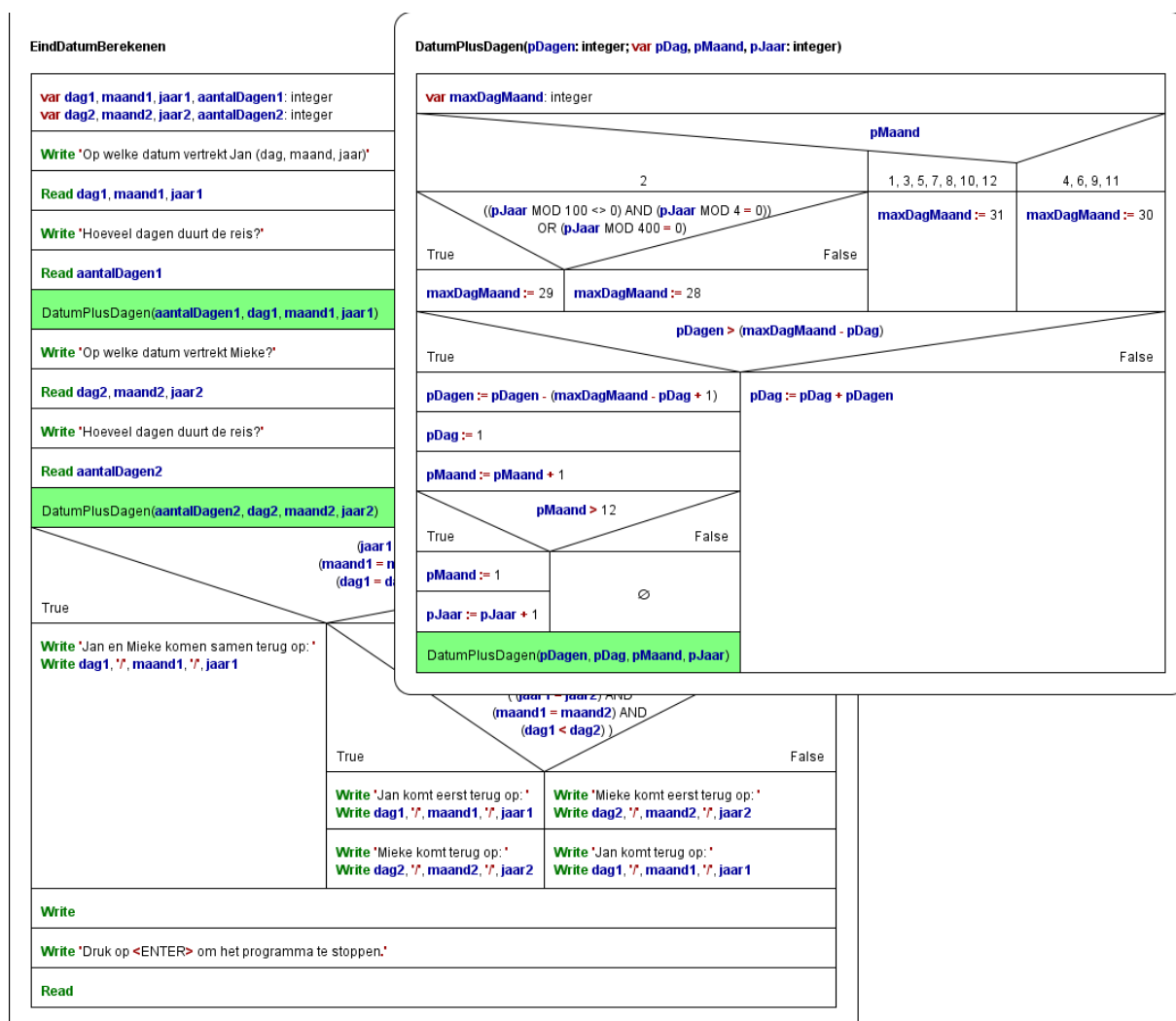
De drie globale variabelen **wiskunde**, **boekhouden** en **informatica**, worden in het hoofdprogramma opgevuld met de waarde die resulteert uit de functie:

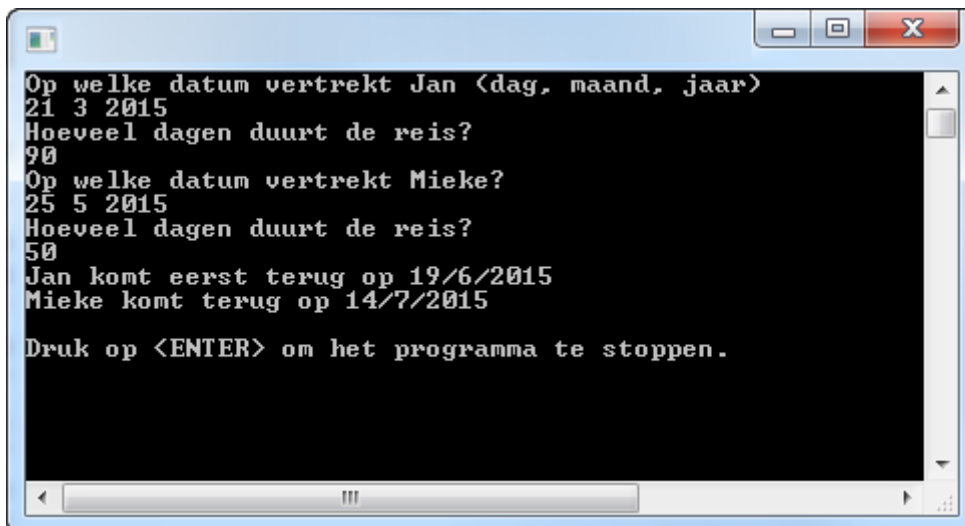
wiskunde := InlezenPunten('wiskunde')
boekhouden := InlezenPunten('boekhouden')
informatica := InlezenPunten('informatica')

2.5.6. Voorbeeld 3: Einddatum berekenen

Jan en Mieke willen beiden afzonderlijk een reis boeken. Afhankelijk van hun vertrekdatum en hoe lang ze weg blijven, willen ze graag weten wat ieder zijn terugkomdag is en wie het eerste terug zal zijn.

Op eerste zicht lijkt dit redelijk eenvoudig, maar om te rekenen met datums, moet je toch met een aantal zaken rekening houden.





```

Op welke datum vertrekt Jan <dag, maand, jaar>
21 3 2015
Hoeveel dagen duurt de reis?
90
Op welke datum vertrekt Mieke?
25 5 2015
Hoeveel dagen duurt de reis?
50
Jan komt eerst terug op 19/6/2015
Mieke komt terug op 14/7/2015

Druk op <ENTER> om het programma te stoppen.

```

EindDatumBerekenen:

In het hoofdprogramma zorgen we eerst voor de nodige invoer: de startdata (dag, maand, jaar) en de duur voor beide reizen. Daarna roepen we een subroutine op om de einddata te berekenen, waarna we deze vergelijken en weergegeven.

De variabelen:

Je ziet hier een hele reeks globale variabelen:

```

var dag1, maand1, jaar1, aantalDagen1: integer
var dag2, maand2, jaar2, aantalDagen2: integer

```

Je kan deze variabelen aanspreken doorheen het hele programma, dus ook vanuit de subroutines binnen dit programma.

Toch worden er bij de oproep van **DatumPlusDagen** een deel van deze variabelen meegegeven:

```

DatumPlusDagen(aantalDagen1, dag1, maand1, jaar1)
DatumPlusDagen(aantalDagen2, dag2, maand2, jaar2)

```

Op eerste zicht lijkt dit misschien onlogisch, daar je globale variabelen kunt aanspreken vanuit je subroutines.

Als je goed kijkt zie je echter dat je hier twee keer dezelfde subroutine (en dus dezelfde code) uitvoert met andere variabelen.

Dit zorgt er dus voor dat dezelfde code op verschillende variabelen uitgevoerd kan worden.

DatumPlusDagen:

Eerst bepalen we het aantal dagen in de gegeven maand. Daarna controleren we of de duur van de reis voorbij het einde van de maand gaat. Als dit niet zo is, kunnen we gewoon het aantal dagen optellen bij de startdatum en zijn we klaar.

In het andere geval, trekken we het aantal dagen die er nog in de maand zitten af van de duur van de reis en roepen we de procedure opnieuw op vanaf de eerste dag van de volgende maand. Deze routine is dus recursief.

De variabelen:

Je ziet hier één lokale variabele:

```
var maxDagMaand: integer
```

Je kan deze enkel aanspreken vanuit deze routine en de levensduur is beperkt tot deze routine. Het hoofdprogramma kent **maxDagMaand** niet. Dit is ook niet nodig. Je hebt deze variabele ook enkel nodig om te bepalen hoeveel dagen er nog in de huidige maand zitten.

Verder wordt er enkel gewerkt met de parameters van deze procedure:

```
DatumPlusDagen(pDagen: integer; var pDag, pMaand, pJaar: integer)
```

Beide soorten parameters komen aan bod. Bij **pDagen** wordt een waarde doorgegeven (**call by value**) en bij de andere parameters, **pDag**, **pMaand** en **pJaar**, wordt een referentie naar de variabelen doorgegeven (**call by reference**).

Concreet houdt dit in dat de globale variabelen **aantalDagen1** en **aantalDagen2** die als eerste parameter worden meegegeven, niet wijzigen tijdens de uitvoering van deze subroutine.

pDagen is een lokale variabele, enkel aanspreekbaar binnen de routine. Deze variabele wordt aangemaakt bij het aanroepen van de routine en verdwijnt weer als de routine eindigt.

De andere variabelen, **dag1**, **dag2**, **maand1**, **maand2**, **jaar1** en **jaar2** wijzigen wel. Hierin komt telkens de einddatum van de reis te staan.

pDag, **pMaand** en **pJaar** zijn lokale referencevariabelen en verwijzen naar de originele variabelen **nl.dag1**, **nl.dag2**, **nl.maand1**, **nl.maand2**, **nl.jaar1** en **nl.jaar2**. Alle wijzigingen gebeuren dus rechtstreeks in de originele variabelen. De benamingen **pDag**, **pMaand** en **pJaar** zijn alleen gekend in de routine als verwijzing naar de andere.

2.5.7. Opgave - Bonus malus

Schrijf een programma dat de nieuwe bonus-malus trap berekent van de autoverzekering, gegeven de bonus-malus trap van vorig jaar en het aantal ongevallen waarvoor het afgelopen jaar een vergoeding werd betaald of zal betaald worden.

Werkwijze:

- de trap is minimum 1 en maximum 18,
- ingeval geen ongeval, is er een daling van 1 trap,
- het eerste ongeval veroorzaakt een stijging van 2 trappen,
- de volgende ongevallen geven een stijging van 3 trappen per ongeval,
- is trap 18 bereikt, dan laten we de klant weten dat hij een andere verzekeringsmaatschappij moet zoeken.

Denk in je opgave goed na over welke procedures of functies je best zou kunnen gebruiken en waar je welke variabelen gaat declareren.

2.6. Opgaven

In dit onderdeel maak je 8 oefeningen. De oplossingen van deze oefeningen, uitgezonderd de laatste, vind je in de webcursus.

Vergeet niet de oplossing van de laatste oefening aan je **coach** te mailen ter verbetering!

2.6.1. Opgave 1: Rekeningnummer

In een bank moet regelmatig gecontroleerd worden of een ingevoerd bankrekening al dan niet juist is.

De controle gebeurt als volgt:

neem de eerste 10 cijfers en deel deze door 97

als de rest van deze deling 0 is, moeten de laatste twee cijfer '97' zijn

anders moet de rest van deze deling gelijk zijn aan de twee laatste cijfers

Schrijf een standaard routine die hiervoor gebruikt kan worden.

Maak een programma om deze routine te testen.

2.6.2. Opgave 2: Welke wagen?

Bij de aankoop van een nieuwe wagen spelen vrij veel factoren een rol. Essentieel is de keuze tussen een diesel- of benzinewagen.

Onderstaande formule berekent het aantal kilometers dat per jaar gereden moet worden, opdat de aankoop van een dieselwagen rendabel zou zijn.

$$\frac{((\text{prijs DW} - \text{prijs BW}) / (\text{jaren in gebruik}) + \text{taks DW} - \text{taks BW}) * 100}{(\text{prijs B} * \text{verbruik BW}) - (\text{prijs D} * \text{verbruik DW})}$$

DW = dieselwagen D = diesel (1 liter)

BW = benzinewagen B = benzine (1 liter)

Schrijf een PSD dat toelaat alle gegevens uit de formule in te voeren, het aantal kilometers per jaar berekent en volgende informatie op het scherm afdrukt:

De dieselversie is goedkoper vanaf km per jaar.

2.6.3. Opgave 3: Breuken vereenvoudigen

Schrijf een programma dat een teller en een noemer inleest van een breuk en deze breuk dan vereenvoudigt.

2.6.4. Opgave 4: Perfecte getallen

Een perfect getal of volmaakt getal is een strikt positief natuurlijk getal dat gelijk is aan de som van zijn echte delers (dus buiten zichzelf, 1 wordt als echte deler meegerekend).

Een aantal voorbeelden:

6 is een perfect getal, want de som van zijn echte delers, $1 + 2 + 3 = 6$

28 is een perfect getal, want de som van zijn echte delers, $1 + 2 + 4 + 7 + 14 = 28$

Schrijf een programma dat twee getallen vraagt en tussen die getallen het kleinste perfect getal zoekt.

2.6.5. Opgave 5: Schrikkeljaar

Schrijf een programma dat een jaartal inleest en uitschrijft of het een schrikkeljaar is.

Zorg dat je deze controle ook op andere plaatsen opnieuw kan gebruiken.

Een jaar is een schrikkeljaar wanneer het deelbaar is door 4, maar niet door 100. Het is toch een schrikkeljaar wanneer het deelbaar is door 400.

2.6.6. Opgave 6: Torens van Hanoi

Volgens een oud Oosters verhaal hangt de tijdsduur van het bestaan van het heelal af van de tijd die een groep gestaag doorwerkende monniken nodig heeft om 64 gouden schijven, alle van verschillende doorsnede, te verplaatsen van de paal waarop zij zijn opgestapeld (van onder naar boven van groot naar klein gerangschikt) naar een zekere doelpaal (in dezelfde rangschikking). Bij het verplaatsen moeten bepaalde regels in acht worden genomen:

1. Niet meer dan één schijf mag tegelijkertijd worden verplaatst.
2. De moniken mogen van een tussenpaal gebruik maken voor de tijdelijke opslag van schijven.
3. Nooit mag een grotere schijf op een kleinere rusten.

Vraag:

Schrijf een programma dat eerst de nodige gegevens vraagt:

het aantal schijven dat verplaatst moet worden
de bronstok (waar de schijven in het begin liggen)
de doelstok (waar de schijven terecht moeten komen)

Aan de hand van deze gegevens schrijf het programma de verschillende stappen die moeten genomen worden.

2.6.7. Opdracht voor de coach: Mastermind

Mastermind is een klassieker onder de spellen in logisch denken. De ene speler probeert de geheime code van de andere speler te ontrafelen.

Je kiest zelf vier cijfers (die in het echte spel voor kleuren staan) 1, 2, 3 en 4. Genereer hiervoor via de Random- functie een combinatie van 4 cijfers. Een cijfer mag meerdere keren gebruikt worden.

De gebruiker (speler) geeft vier cijfers in.

Hierop geeft het programma een resultaat weer nl. in de vorm van letters:

- een rode pin (R): een cijfer staat op de juiste positie

- een witte pin (W): het cijfer komt wel in de code voor, maar staat niet op de juiste positie

Een voorbeeld:

Veronderstel dat de geheime code 3 2 1 2 is.

De gok is 1 2 3 4 dan geeft dit als resultaat: RWW

R: 2 staat op zijn plaats

W: 1 zit erin maar staat niet juist

W: 3 zit erin maar staat niet juist

voor de 4 wordt niets getoond.

De speler mag opnieuw proberen tot hij de juiste combinatie gevonden heeft.

Als de code gekraakt is, laat je de speler weten hoeveel pogingen hij nodig had om de code te kraken.

Stuur je oplossing (zowel .nsd- als .pas-bestand) aan je **coach**. Gebruik als onderwerp "**Mastermind**".

Hoofdstuk 3. Tot slot

In dit laatste hoofdstuk overlopen we nog de volgende onderdelen:

de eindoefening
hoe je de afdrukbare versie van deze cursus kan downloaden
wat je best doorneemt na deze module

3.1. Einde cursus

3.1.1. Eindoefening

Bij deze cursus hoort ook een **eindoefening**. In die eindoefening komen de belangrijkste zaken van deze cursus terug aan bod.

Stuur een bericht aan je **coach** met als onderwerp "**Opdracht eindoefening Procedures en functies**".

3.1.2. Wat nu?

Je hebt nu het Programmatie logica - Procedures en functies afgerond.

Heb je het deel **Arrays en strings** nog niet doorgenomen, dan kan je dat nu doen.

Als je na deze drie delen nog zin hebt in meer, kan je nog verder verdiepen in de delen

Sorteren: sorteren van gegevens

Bestanden: omgaan met bestanden

Na het doornemen van **Basisstructuren, Procedures en functies** en **Arrays en strings** kan je voor de opleiding *Programmatielogica* een getuigschrift met resultaatsvermelding krijgen. Hiervoor leg je een eindtest af in één van de VDAB-opleidingscentra. De nodige informatie over de verschillende centra vind je terug in de module **Intro Informatica**.

Ligt je focus meer op **webapplicaties** en wil je graag al beginnen aan een concrete taal, dan zijn onze cursussen **Javascript** of **Inleiding tot PHP** misschien wel interessant voor jou. Je neemt dan best eerst nog het onderdeel **Arrays en strings** door.