



Programmatie logica

Basisstructuren

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	7
1.1.	Algemene informatie.....	7
1.1.1.	Waarom?	7
1.1.2.	Werkwijze	7
Hoofdstuk 2.	Inleiding	9
2.1.	Probleem oplossen	9
2.1.1.	Probleemstelling.....	9
2.1.2.	Een ander voorbeeld	10
2.1.3.	Definitie Programma	10
2.1.4.	Invoer-proces-uitvoerschema's.....	10
2.1.5.	Algoritme	11
2.1.6.	Gestructureerd programmeren.....	12
2.1.7.	Programmeertalen	13
2.2.	Opgaven.....	14
2.2.1.	Opgave 1.....	14
2.2.2.	Opgave 2: Opdracht voor je coach	14
Hoofdstuk 3.	Variabelen en operatoren	15
3.1.	Variabelen	15
3.1.1.	Definitie	15
3.1.2.	Een voorbeeld.....	15
3.1.3.	Naamkeuze	16
3.1.4.	Toekenningsopdracht.....	16
3.1.5.	Lees- en schrijfoopdrachten	17
3.1.6.	Initialiseren	18
3.2.	Opgaven1 – Opdracht voor je coach	18
3.3.	Gegevenstypes	18
3.3.1.	Enkelvoudige gegevenstypes	18
3.3.2.	Boolean.....	19
3.3.3.	Char	19
3.3.4.	String	20
3.3.5.	Integer	20
3.3.6.	Real	22

3.3.7.	Array	22
3.3.8.	Keuze van het gegevenstype	23
3.3.9.	Variabelen declareren	23
3.3.10.	Variabelentabellen	24
3.3.11.	Toestandstabellen	24
3.4.	Operatoren	25
3.4.1.	Definitie	25
3.4.2.	Unair en binair	25
3.4.3.	Logische, wiskundige en vergelijkingsoperatoren	26
3.4.4.	De verschillende operatoren	26
3.4.5.	Prioriteitsregels	27
3.4.6.	Operatoren en gegevenstype	28
3.4.7.	Logische uitdrukking	29
3.5.	Opgaven2	31
Hoofdstuk 4.	Tekenen van diagrammen	32
4.1.	Structorizer	32
4.1.1.	Installatie	32
4.1.2.	Configuratie	33
4.1.3.	Gebruik	35
4.2.	Opgaven1	36
4.2.1.	Opgave 1	36
4.2.2.	Opgave 2	37
4.3.	Lazarus	37
4.3.1.	Installatie	37
4.3.2.	Gebruik	38
4.3.3.	Variabelen declareren	40
4.4.	Opgaven2	42
4.4.1.	Opgave 1	42
Hoofdstuk 5.	Basisstructuren	43
5.1.	Sequentie	43
5.1.1.	Voorbeeld 1	43
5.1.2.	Voorbeeld 2	45
5.2.	Opgaven1	45
5.2.1.	Opgave 1: Kwadraat	46

5.2.2.	Opgave 2: Som en gemiddelde.....	46
5.2.3.	Opgave 3: Loon.....	46
5.2.4.	Opgave 4: Duurtijd	46
5.2.5.	Opdracht voor je coach: Draagkracht	46
5.3.	Selectie	46
5.3.1.	Samengestelde voorwaarde.....	47
5.4.	Opgaven2	49
5.4.1.	Opgave 1: Temperatuur	50
5.4.2.	Opgave 2: Factuur	50
5.4.3.	Opgave 3: Rechthoek	50
5.4.4.	Opgave 4: Examens	50
5.4.5.	Opgave 5: Deling	50
5.4.6.	Opgave 6: Eenvoudige sortering	50
5.4.7.	Opdracht voor je coach: Kindergeld.....	50
5.5.	Iteratie	51
5.5.1.	For ... to	52
5.5.2.	While	53
5.5.3.	Until	54
5.5.4.	Vergelijking.....	57
5.6.	Opgaven3	57
5.6.1.	Opgave 1: Waar loopt het fout?.....	58
5.6.2.	Opgave 2: Iteratie toevoegen.....	58
5.6.3.	Opgave 3: Som van 10	60
5.6.4.	Opgave 4: Temperatuur	60
5.6.5.	Opgave 5: Examens2	60
5.6.6.	Opgave 6: Getal raden.....	60
5.6.7.	Opgave 7: Min en max.....	60
5.6.8.	Opgave 8: Faculteit.....	60
5.6.9.	Opgave 9: Fibonacci.....	61
5.6.10.	Opgave 10: Tafel van vermenigvuldiging	61
5.6.11.	Opgave 11: Sparen	61
5.6.12.	Opdracht voor je coach: Lidgeld.....	61
5.7.	Meervoudige selectie	61
5.7.1.	Variant op selectie.....	61

5.7.2.	Meerdere waarden.....	63
5.8.	Opgaven4	65
5.8.1.	Opgave 1: Omrekenen.....	65
5.8.2.	Opgave 2: Graad.....	65
5.8.3.	Opdracht voor je coach: Rekenmachine	66
5.9.	Gestructureerd programmeren.....	66
5.9.1.	Efficiënt werken.....	66
5.9.2.	Efficiënt programmeren	67
5.9.3.	Onderhoud	67
5.10.	Opgaven5	68
5.10.1.	Opgave 1: Fibonacci structureren	68
5.10.2.	Opgave 2: Steen - Papier - Schaar	70
Hoofdstuk 6.	Einde cursus.....	72
6.1.	Eindoefening.....	72
6.2.	Wat nu?	72

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 **programmatielogica** 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 programmaticalogica bestaat uit verschillende delen.

In het eerste deel, **Basisstructuren**, leer je

- omgaan met variabelen en operatoren
- de basisstructuren van het programmeren gebruiken

Na dit doorgenomen te hebben kan je de delen **Procedures en functies** en/of **Arrays en Strings** doornemen.

Als je na deze drie delen nog zin hebt in meer, kan je 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. Later meer over beide programma's.

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. Inleiding

2.1. Probleem oplossen

In dit onderdeel geven we een inleiding en definiëren we enkele belangrijke begrippen.

2.1.1. Probleemstelling

Amedé krijgt een erfenis van € 100.000,00 en hij heeft het geld niet direct nodig. Daarom gaat hij het geld beleggen. We kunnen dit probleem op 2 manieren aanpakken:

Specifieke aanpak

Zijn bankdirecteur vertelt hem dat hij een formule kan kiezen waarbij hij elk jaar 8% intrest krijgt, als hij zijn geld voor 10 jaar vastzet. Amedé wil graag weten hoeveel geld hij na die tien jaar zal overhouden.

Er bestaan natuurlijk (ingewikkelde) formules om dit soort berekeningen uit te voeren, en de bank zal waarschijnlijk ook kant-en-klare intresttabellen ter beschikking hebben. Maar als Amedé zelf aan het rekenen wil gaan, dan zou hij dat als volgt kunnen doen:

- Na 1 jaar is mijn kapitaal gelijk aan mijn startkapitaal + 8%,
of dus € 108 000,00
- Na 2 jaar is mijn kapitaal het eindkapitaal van het eerste jaar + 8%:
dus € 108 000,00 + 8% geeft € 116 640,00
- Voor elk volgend jaar is mijn eindkapitaal gelijk aan het beginkapitaal van dat jaar, verhoogd met 8%.

Algemene aanpak

Amedé wil niet over één nacht ijs gaan. Hij wil een aantal beleggingsformules met elkaar vergelijken. Hierbij stelt hij vast dat hij eigenlijk elke keer dezelfde procedure moet afhandelen, maar met andere cijfertjes: de looptijd kan variëren, het intrestpercentage kan variëren, in feite kan ook zijn startkapitaal variëren, en afhankelijk daarvan komt hij altijd tot een ander eindresultaat.

Maar algemeen blijft de berekening steeds: voor elk jaar van de looptijd neem je het kapitaal aan het einde van de vorige periode en je verhoogt dat met de rentevoet.

Algemeen gesteld wordt het probleem dan:

“Wat is de eindwaarde van mijn belegging (noem dit **eindkapitaal**) als ik een bepaald beginkapitaal (noem dit **kapitaal**) beleg aan een bepaalde jaarlijkse rentevoet (noem dit **rente**) en voor een bepaald aantal jaren (noem dit **looptijd**).”

De woorden die in **vet blauw** zijn gedrukt, noemt men **variabelen**: het zijn objecten die een bepaalde waarde kunnen hebben. Die waarde kan variëren (vandaar de naam “variabelen”).

Het specifieke probleem van hierboven is dan eigenlijk een concreet voorbeeld van het algemene probleem, waarbij de variabelen volgende waarden krijgen: **kapitaal**=€ 100 000,00, **rente**=8% en **looptijd**=10 jaar. Het **eindkapitaal** wordt dan het resultaat van de hele berekening (wat afgerond € 215 892,50 is).

2.1.2. Een ander voorbeeld

Je wil het gemiddelde berekenen van de getallen 16, 8, 5, 12 en 23. Dit is een **specifiek probleem**.

De oplossing ligt voor de hand: de getallen optellen (64) en die som delen door 5. Maar deze berekening ($64/5=12,8$) is natuurlijk alleen juist in dit ene geval: hebben we andere getallen, of hebben we zelfs meer of minder getallen, dan zal de berekening telkens anders zijn.

We kunnen het probleem omvormen tot een **algemeen probleem**:

“Wat is het gemiddelde (noem dit **gemiddelde**) van een vijftal getallen (noem deze **getal1**, **getal2**, **getal3**, **getal4**, **getal5**)?”

Nog algemener wordt dit:

“Wat is het gemiddelde (noem dit **gemiddelde**) van een willekeurig aantal getallen (noem deze **getal1** t/m **getalN**)?”

2.1.3. Definitie Programma

We kunnen nu de stappen die nodig zijn om ons probleem op te lossen zo gaan formuleren dat ze begrijpelijk en uitvoerbaar zijn voor een computer. Dan spreken we van een programma.

DEFINITIE

Een **programma** (of computerprogramma) is een lijst van instructies die door de computer kunnen worden uitgevoerd, en die tot doel hebben een probleem op te lossen.

Indien een programma gemaakt is om een **algemeen probleem** op te lossen, kan het gebruikt worden om een hele reeks **specifieke problemen** op te lossen, door steeds andere waarden in te vullen voor de **variabelen**.

2.1.4. Invoer-proces-uitvoerschema's

In elk van de bovengenoemde voorbeelden kan je een aantal gemeenschappelijke elementen onderscheiden:

- we hebben telkens een aantal begingegevens (**invoervariabelen**)
- die een bepaalde verwerkingsprocedure ondergaan (**verwerkingsproces**)
- wat dan resulteert in andere gegevens (**uitvoervariabelen**)

Voor het beleggingsvraagstuk van Amedé kan je dit schematisch weergeven als volgt:

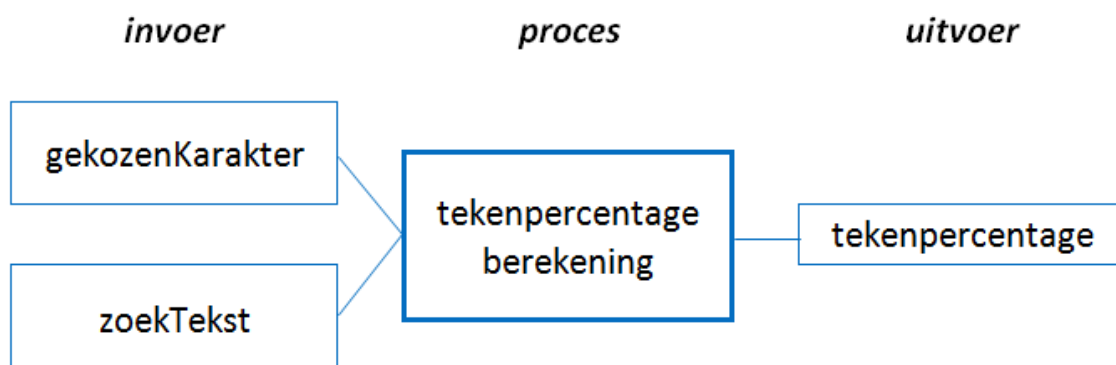


Tot hiertoe stonden de variabelen steeds voor getallen, maar dat kunnen evengoed letertekens (*karakters*), reeksen letertekens (*strings* of *tekenreeksen*), logische waarden (*waar* of *onwaar*) en dergelijke zijn.

Een voorbeeld: we wensen een programma dat in een tekst gaat tellen hoe vaak een bepaald teken voorkomt. De algemene probleemstelling is dan:

Bepaal hoe vaak, in percenten uitgedrukt (noem dit **tekenpercentage**) een bepaald karakter (noem dit **gekozenKarakter**) in een ingegeven tekst (noem deze **zoekTekst**) voorkomt.

Het schema zal er voor dit voorbeeld als volgt uitzien:



2.1.5. Algoritme

Het verwerkingsproces is in deze schema's een beetje een "zwarte doos": je weet wat je er in stopt, en je kent de betekenis van wat eruit komt, maar wat er tussenin exact gebeurt, wordt in het midden gelaten.

In de praktijk zullen we nochtans deze berekening door een computer willen laten uitvoeren, maar dan moeten we precies vertellen hoe die berekening moet gebeuren. Zo'n exacte beschrijving van een gegevensverwerkend proces noemt men een algoritme. In feite is het algoritme nog iets meer dan dat: het beschrijft óók de invoer- en uitvoervariabelen.

DEFINITIE

Een **algoritme** is een exacte beschrijving van een gegevensverwerkend proces én van de invoer- en uitvoervariabelen van dat proces.

De term "algoritme" is afkomstig van de naam van een befaamd Arabisch auteur van leerboeken, Abu Ja'far Mohammed ibn Musa al-Khowarizmi: het laatste deel van zijn naam werd verbasterd tot "algoritme". De man was onder andere auteur van het boek "Kitab al jabr w'ahl", en uit "al jabr" werd dan weer het woord "algebra" afgeleid!

Een goede metafoor voor een algoritme is een recept uit een kookboek:

- er wordt in aangegeven wat de invoer is (de ingrediënten en eventueel het nodige materiaal),
- wat de uitvoer zal zijn (het te bereiden recept, bijv. "Stoofvlees op grootmoeders wijze") en

- hoe de invoer tot de uitvoer wordt omgevormd (de eigenlijke receptinstructies: “smelt de boter, ...”)

Vereisten

Om voor een computer bruikbaar te zijn, moet een algoritme voldoen aan een aantal vereisten:

1. De **invoer** en **uitvoer** moeten duidelijk gespecificeerd zijn.
vb. in een recept: wat is een “snuifje” zout precies?
2. Het moet **robuust** zijn.
vb. niet toegestane invoerwaarden moeten opgevangen worden, bijv. een deling door 0
3. Het moet **uitvoerbaar** zijn.
Het moet bestaan uit opdrachten die de computer – of algemener de verwerker van de opdrachten – kent: een beginnende kok kent wellicht de term “blancheren” niet...
4. Het moet **eenduidig** zijn.
De opdrachten moeten nauwkeurig én ondubbelzinnig zijn.
5. Het moet **eindig** zijn.
Het aantal opdrachten moet eindig zijn, maar het geheel moet ook in een eindige tijd kunnen worden uitgevoerd: telkens wanneer in een algoritme een aantal opdrachten herhaald worden, moet ook duidelijk opgegeven worden wanneer de herhaling moet stoppen.
6. Het moet **algemeen** zijn.
Zo breed mogelijk toepasbaar: een recept moet bijvoorbeeld kunnen worden uitgevoerd voor elk willekeurig aantal genodigden.
7. Het moet **correct** zijn.
8. Het moet **efficiënt** zijn.

De laatste voorwaarde wordt in deze cursus niet verder uitgediept: hoe evident deze eis op het eerste zicht ook mag lijken, het aantonen van het efficiënt-zijn van een algoritme is alles behalve eenvoudig.

2.1.6. Gestructureerd programmeren

Een belangrijke methode om algoritmes te creëren die aan de vermelde eisen voldoen, is het gestructureerd programmeren: door een vast organisatiepatroon te volgen, vastgelegd in logische stappen, wordt een “proper” algoritme gemaakt.

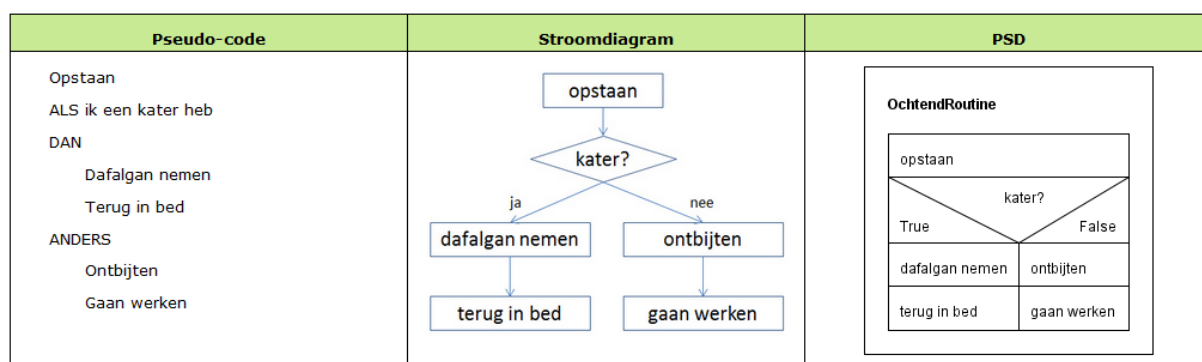
Gestructureerd programmeren heeft heel wat voordelen:

- Het ontwerpen van een oplossing wordt vereenvoudigd, doordat je gebruik kan maken van “standaard”-oplossingswijzen.
- Je programma wordt beter leesbaar doordat het logisch opgebouwd is.
- Er treden doorgaans minder fouten op.
- Hierdoor wordt de testperiode ingekort.
- Onderhoud wordt door de “zuivere” opbouw vereenvoudigd.

Er zijn verschillende voorstellingswijzen voor deze gestructureerde algoritmes:

- *Pseudocode* (in min of meer gewone spreektaal wordt de logische structuur weergegeven)
- *Stroomdiagram*
- *Nassi-Shneiderman-diagrammen* (of PSD's: **P**rogramma **S**tructuur **D**iagrammen)

Ter illustratie een heel eenvoudig voorbeeld in elk van de drie notaties. Het gaat om een hopelijk niet te herkenbaar ochtendritueel...



2.1.7. Programmeertalen

Belangrijk is dat elk van deze voorstellingswijzen los staat van de programmeertalen die je uiteindelijk zal gaan gebruiken: dat kan dus VB.net, C#.net, Java, C, C++, Pascal, PHP, Cobol of zowat elke andere taal zijn.

Hoe zit dat nu eigenlijk met al die programmeertalen?

Computers moeten heel precieze en ondubbelzinnige opdrachten krijgen om iets nuttig te kunnen uitvoeren. Computers begrijpen, helaas, geen Nederlands of Engels (of welke menselijke taal dan ook). Daarom moeten de opdrachten aangeboden worden in een vorm die een computer kan begrijpen en die tegelijkertijd begrijpbaar is voor een mens: de programmeertalen.

De computer zelf werkt – zoals je wel weet – met énen en nullen, de zogenaamde bits. Dit betekent dat alle opdrachten die de computer kan uitvoeren in bits moeten worden uitgedrukt. Men noemt dit **binaire code** of **machinetaal**.

Al snel na de ontwikkeling van de eerste computers ging men meer “menselijke” talen ontwikkelen. Voor elke taal werd een compiler gemaakt: dat is een programma dat de voor mensen leesbare code (de broncode of source code) omzet naar machinetaal, leesbaar door de computer (object code).

De eerste echte programmeertalen, de zogenaamde assembleertalen, waren in onze ogen nog zeer primitief en moeilijk leesbaar. Men noemt ze ook **“tweede generatietalen” (2GL)**, waarbij de eerste generatie dan de pure machinetaal was.

Gaandeweg werden steeds menselijkere talen ontwikkeld (waarvoor de compilers natuurlijk ook steeds complexer werden, omdat de source code steeds verder van de object code kwam te staan).

Talen van de **derde generatie (3GL)** stonden weer een stuk verder, ondersteunden gestructureerde algoritmes en konden bijvoorbeeld met variabelen gaan werken (zie verder), waardoor ze nog dichter bij de mens (en dus verder van de machine) staan. Vele “moderne” talen vallen in deze categorie: verouderde talen als Fortran, ALGOL en COBOL, maar ook recentere talen als BASIC, C, C++ en Java. Je kan ervan uitgaan dat de meeste momenteel in gebruik zijnde programmeertalen tot de 3GL behoren.

Vierde-generatietalen (4GL) staan nog dichter bij de mens en werden meestal ontwikkeld in database-context, bijv. SQL. Een precieze definitie geven van wat 4GL zijn is moeilijk, en het wordt nog verwarrender doordat er een 4GL is met de naam... 4GL!

Tenslotte onderscheidt men ook nog **vijfde-generatietalen (5GL)**. Hierbij specificeert de programmeur geen algoritme maar het probleem zelf, met een aantal bijbehorende beperkingen. Vijfde generatietalen worden vooral gebruikt op het gebied van kunstmatige intelligentie. Het bekendste voorbeeld is Prolog.

Welke programmeertaal kan je, na deze cursus, nu best leren?

Het antwoord ligt voor de hand: er is geen “beste” programmeertaal. Het hangt er allemaal vanaf wat je wil gaan doen.

Hou altijd in je achterhoofd dat een programmeertaal alleen maar een gereedschap is om ideeën om te zetten in automatische bewerkingen. De kwaliteit van een programma wordt voor het grootste deel bepaald door de programmeur, de gebruikte gereedschappen zijn van minder belang.

2.2. Opgaven

2.2.1. Opgave 1

Maak de oefening in de webcursus

2.2.2. Opgave 2: Opdracht voor je coach

De volgende taken worden in een bedrijf op dit moment manueel gedaan:

- maandloon berekenen
- de jaarlijkse loonstaat voor elke werknemer berekenen en opstellen
- een lijst opstellen van werknemers die in aanmerking komen voor een sinterklaasgeschenk
- klachten van klanten opvolgen
- dagelijks de gepresteerde uren van elke werknemer registreren

Kies uit deze lijst 2 taken die je als eerste zou automatiseren. Stuur je antwoord op deze vraag aan je coach. Vermeld ook waarom je deze twee taken gekozen hebt. Gebruik als onderwerp van je bericht **"Probleem oplossen"**.

Hoofdstuk 3. Variabelen en operatoren

3.1. Variabelen

In wat voorafging werkten we al af en toe met variabelen (bijv de invoer- en uitvoervariabelen). We gingen ervan uit dat die bepaalde waarden kunnen aannemen, maar hoe dat in z'n werk gaat, hebben we nog niet nader bekeken. In dit hoofdstuk gaan we daar wat dieper op in.

Welke soorten gegevens (getallen, karakters, ...) er in een variabele kunnen worden gestopt, is een ander onderwerp dat in dit hoofdstuk aan bod komt.

Ook zullen we een aantal richtlijnen geven i.v.m. de naamgeving van variabelen.

3.1.1. Definitie

De gegevens die we in de loop van een programma gebruiken, zowel de invoergegevens als de uitvoergegevens, worden opgeslagen in het interne geheugen van de computer. Voor elke variabele wordt geheugen **gereserveerd**: het interne geheugen is opgedeeld in een heleboel aparte vakjes, de geheugenplaatsen, van een welbepaalde lengte, en voor elke variabele worden een aantal plaatsen gereserveerd – zoveel als er nodig zijn om de informatie die in de variabele gestopt zal worden, op te slaan. Doordat deze ruimte wordt gereserveerd, wordt dit stuk van het geheugen geblokkeerd voor alle andere toepassingen, zodat “per ongeluk wissen” uitgesloten is.

Elke variabele heeft 4 eigenschappen:

- een **naam**
- een **geheugenplaats** waar de naam naar verwijst
- een **waarde** die op die geheugenplaats wordt gestockeerd
- een **gegevenstype** dat aangeeft welk soort gegevens in de variabele kan worden opgeslagen

DEFINITIE

Een **variabele** is een gereserveerde geheugenplaats en de naam die daaraan wordt gegeven in het programma. Een variabele wordt gekenmerkt door de naam, de geheugenplaats, de waarde en het gegevenstype.

3.1.2. Een voorbeeld

Je kan je het geheugen van de computer voorstellen als een muur met bagagekluizen in een station. Elke kluis komt overeen met een geheugenplaats, en het nummer van de kluis komt overeen met het adres van de geheugenplaats. Iemand die een kluis huurt, wordt even eigenaar van die kluis, en wordt dan de “variabele” die naar de kluis verwijst. Het soort bagagestukken dat in de kluis kan (afhankelijk van het formaat van de kluis) kan je dan beschouwen als het gegevenstype, en de bagagestukken als de waarde.

Adres:	Nummer kluis
Naam variabele:	Huurder kluis
Waarde:	Welke bagagestukken
Gegevenstype:	Soort bagagestukken

In deze metafoor is de bewaker van de kluizenzaal het besturingssysteem, dat er onder meer voor zorgt dat geen twee huurders dezelfde kluis krijgen.

3.1.3. Naamkeuze

Algemeen kan je stellen dat de naam van een variabele steeds één woord is, bestaande uit een combinatie van letters, cijfers en een aantal speciale tekens (bijv. underscore: `_`). Spaties zijn in de meeste programmeertalen niet toegestaan. Deze naam moet aan twee voorwaarden voldoen:

- de gekozen naam moet **betekenisvol** zijn: de naam zegt iets over de waarde die er in bijgehouden wordt
- elke variabele moet een **unieke** naam krijgen

Sommige programmeertalen maken een onderscheid tussen hoofd- en kleine letters (dan is de variabele **naam** niet gelijk aan de variabele **Naam**), andere niet.

Binnen deze cursus maken we gebruik van **lowerCamelCase**.

CamelCase is de gewoonte om de namen van je variabelen samen te stellen uit verschillende woorden. Waar de woorden normaal met een spatie worden gescheiden, wordt de spatie weggelaten en de daarop volgende letter wordt vervangen met de corresponderende hoofdletter. **lowerCamelCase** betekent dat je het eerste woord met een kleine letter laat beginnen.

Voorbeeld: **gekozenKarakter**

3.1.4. Toekenningsopdracht

Met een **toekenningsopdracht** of assignment wordt een waarde toegekend aan een variabele, en dus opgeslagen in de geheugenplaats waarnaar de variabele verwijst. In een PSD wordt dit genoteerd als:

```
<variabele> := <waarde>
```

In pseudo-code zou je kunnen schrijven: “variabele **wordt** waarde”. Een concreet voorbeeld van een assignment is:

```
getal := 27
```

Hierbij wordt in de geheugenplaats waarnaar de variabele **getal** verwijst, de waarde 27 opgeslagen.

In plaats van een waarde kan je ook algemener het resultaat van een berekening of **expressie** in de variabele stoppen. Een eenvoudig voorbeeld:


```
getal := 9 * 3
```

Hierbij wordt het resultaat van de berekening “9 maal 3” in de variabele **getal** gestopt.

Of iets moeilijker:

```
getal := getal * getal
```

In dit geval wordt de inhoud van de geheugenplaats waarnaar **getal** verwijst gelezen, het getal wordt met zichzelf vermenigvuldigd (het kwadraat dus) en het resultaat wordt opnieuw in dezelfde geheugenplaats gestopt.

Algemeen is een toekenningso opdracht dus van de vorm:

```
<variabele> := <expressie>
```

3.1.5. Lees- en schrijfo opdrachten

Om in een PSD gegevens in te lezen wordt de **Read**-instructie gebruikt:

```
Read getal
```

Dit betekent dat het programma hier wacht op invoer (door de gebruiker, dus doorgaans vanaf het toetsenbord) van een waarde voor de variabele **getal**.

Gegevens uitvoeren (doorgaans naar het scherm) gebeurt met de **Write**-instructie:

```
Write getal
```

Hierdoor wordt de waarde van de variabele op het scherm getoond.

Standaard wordt steeds gelezen vanaf het toetsenbord en geschreven naar het scherm, maar de meeste programmeertalen laten vanzelfsprekend ook toe andere “media” te gebruiken, bijv. bestanden (invoer en/of uitvoer), printers (uitvoer) en muizen (invoer).

Meerdere variabelen inlezen of wegschrijven kan door meerdere Read- of Write-opdrachten na elkaar te geven:

```
Read getal1
```

```
Read getal2
```

```
Read getal3
```

Maar dit kan ook korter, door de variabelen achter elkaar op te sommen, gescheiden door komma's:

```
Read getal1, getal2, getal3
```

3.1.6. Initialiseren

Voor je een variabele kan gebruiken in een expressie of een schrijfo opdracht, moet deze variabele eerder in het programma al een waarde gekregen hebben.

Vaak spreken we van initialiseren: je geeft je variabele een initiële waarde.

<code>getal := 0</code>
<code>getal := getal * getal</code>

Je kan een *lees*-opdracht of de *toewijzing* van een expressie evengoed beschouwen als een initialisatie.

<code>Read getal</code>	<code>Read getalA</code>
<code>getal := getal * getal</code>	<code>Read getalB</code>
	<code>som := getalA + getalB</code>

Als je straks gaat werken in Structorizer, zal je merken dat je variabele pas een blauwe kleur krijgt als deze geïnitieerd is.

3.2. Opgaven1 – Opdracht voor je coach

Je wil, uitgaande van het maandloon van een werknemer, een algoritme uitwerken om het loon te berekenen voor een bepaald aantal gewerkte dagen.

Bijv: een werknemer wordt slechts 20 dagen van een bepaalde maand uitbetaald.

In ons bedrijf berekenen we het dagloon door het maandloon te delen door 25.

Vraag:

Welke variabelen zou je definiëren?

Kan je omschrijven hoe je (in grote lijnen) dit probleem verder zou oplossen ?

Zet je antwoord in een bericht aan je coach. Gebruik als onderwerp "**Variabelen**".

3.3. Gegevenstypes

In dit onderdeel bekijken we de verschillende gegevenstypes. We geven ook enkele tips om het juiste gegevenstype te kiezen.

3.3.1. Enkelvoudige gegevenstypes

We kunnen een viertal verschillende gegevenstypes onderscheiden die we onder de noemer *enkelvoudig* samenbrengen:

- Het type **integer**: gehele getallen (dus zonder decimalen)
- Het type **real**: reële getallen, oftewel kommagetallen

- Het type **char**: één enkel karakter (letter, cijfer of ander teken); in wat volgt zullen we dergelijke waarden tussen enkele aanhalingstekens zetten:
teken := 'a'.
- Het type **boolean**: kan enkel de waarden "True" of "False" aannemen.

Bij programmeertalen bestaan er nog andere gegevenstypes. Denk bijv. aan een **date** (datum/tijd) gegevenstype, zoals we dat ook in Microsoft Access kennen. Zo is er ook het tekst-type of **string**, dat een reeks van karakters kan bevatten (in plaats van slechts één enkel karakter).

Tevens bestaan er veel complexere gegevenstypen; de meeste daarvan zijn samengesteld uit meerdere enkelvoudige types: de samengestelde gegevenstypes. In wat volgt zullen we onder andere het **array type** (tabeltype) als extra gegevenstype introduceren.

DEFINITIE

Een **gegevenstype** is een waardenverzameling met een aantal daarop gedefinieerde bewerkingen.

De waardenverzameling van een gegevenstype omvat alle mogelijke verschillende waarden die in een variabele van dat type kunnen worden opgeslagen. De grootte van die verzameling is onder meer afhankelijk van de hoeveelheid geheugenruimte die aan dat gegevenstype gekoppeld is.

3.3.2. Boolean

Oorspronkelijk nam een **boolean** als geheugenplaats slechts 1 bit in beslag. Zoals je weet is dat de meest elementaire geheugenruimte: een schakelaartje dat aan of uit kan staan. De waardenverzameling van dit type bevat dus slechts twee waarden: aan en uit.

In moderne computerstructuren is een byte de kleinste adresseerbare eenheid van geheugen. Om dan toch met bits te kunnen werken, zouden er zogenaamde "bit shift operaties" moeten gebeuren, wat voor de nodige vertraging zorgt. Daarom zullen de meeste programmeertalen een byte ipv een bit reserveren voor een boolean. De waardenverzameling blijft wel beperkt tot de twee waarden.

De twee verschillende waarden worden vaak weergegeven met 0 en 1, waarbij de 0 staat voor het ontbreken van een signaal (de schakelaar staat uit; dit komt overeen met onwaar, **FALSE**) en de 1 (of soms ook -1) voor de aanwezigheid van een signaal (de schakelaar staat aan; dit komt overeen met waar, **TRUE**).

Je kan deze waardenverzameling dan weergeven als {True, False}.

3.3.3. Char

De waardenverzameling van het **char** type noemt men ook wel een alfabet, maar dat moet je ruimer interpreteren dan de 26 letters van ons Latijnse alfabet. Een veel gebruikte waardenverzameling is ASCII (American Standard Code for Information Interchange). Oorspronkelijk werden hiervoor 7 bits voorzien; in de praktijk gebruikte men meestal een byte, waarbij de achtste bit dan als controle werd gebruikt. Op deze manier konden 128 verschillende waarden (7 bits, dus 2^7) worden weergegeven. Je kan de in de webcursus downloaden.

Wanneer je een char-variabele gebruikt, wordt in de bijhorende geheugenruimte dus eigenlijk een getal in het bereik 0-127 opgeslagen. Wanneer deze waarden op het scherm worden getoond, wordt dit weergegeven als het teken dat je in de tabel in de bijlage vindt. Zo komt de hoofdletter 'A' overeen met getalwaarde 65; de kleine letter 'a' met getalwaarde 97. Een variabele met char als gegevenstype, kan ook een cijfer bevatten, bijv. het cijfer 7 dat overeenkomt met de getalwaarde 55. Beperking hierbij is dat er met dergelijke variabelen, ook al bevatten ze een cijfer, niet kan gerekend worden (bijv. maken van een som). Er kan alleen gerekend worden met echte numerieke gegevenstypes, zoals integer en real (zie verder).

Later werd een uitgebreide ASCII-tabel gecreëerd door ook de achtste bit echt in gebruik te nemen: men noemt dit Extended ASCII of ASCII-II. Deze waardenverzameling bevat dus 2^8 of 256 verschillende waarden. Ook deze 128 extra tekens kan je in de webcursus vinden.

Recenter nog werden nieuwe codetabellen ingevoerd. De meest verspreide daarvan is Unicode, die 16 bits per variabele reserveert ($2^{16}=65.536$ verschillende waarden). Enkel meer recente programmeertalen (bijv. Java) kunnen hiermee werken. In deze cursus zullen we echter steeds werken met de "gewone" ASCII-waardenverzameling.

3.3.4.String

Een **string** is niets meer of minder dan een reeks karakters die deel uitmaken van een bepaalde waardenverzameling (bvb. ASCII). In deze waardenverzameling vind je zowel letters, cijfers als speciale tekens terug.

Voorbeelden van geldige strings zijn dan ook: 'jan', 'jan30', 'jan-willem', maar ook reeksen die uitsluitend uit cijfers bestaan – bvb. '123', '5482'.

Het verschil tussen het getal 123 en de string '123' is dat je met het eerste kan rekenen en met de tweede niet.

Voor cijferreeksen waar je niet mee hoeft te rekenen – bvb. postcode – kan je kiezen: ofwel sla je deze op als 3600 of '3600'.

Let op!

Bij het **sorteren van nummers** wordt rekening gehouden met de numerieke waarde, zo is 100 groter dan 2.

Bij het **sorteren van strings** wordt er karakter per karakter vergeleken en rekening gehouden met de overeenstemmende getalwaarde of ASCII-waarde van het karakter in de waardenverzameling. Zo heeft het karakter '1' van '100' als getalwaarde 49, en het karakter '2' de getalwaarde 50. Het resultaat is dat '100' kleiner is dan '2'.

3.3.5.Integer

Het **integer** type heeft in principe als waardenverzameling de "gehele getallen" uit de wiskunde. Er is één belangrijk verschil: de wiskundige verzameling is oneindig, terwijl de waardenverzameling voor een computer per definitie eindig is (er kan slechts een begrensd aantal verschillende combinaties worden gemaakt met de bits in de gereserveerde geheugenplaatsen).

Hoeveel verschillende waarden in de waardenverzameling zitten, is afhankelijk van de computer waarop je werkt (en meer specifiek het besturingssysteem) en de programmeertaal die je gebruikt: er wordt immers niet altijd even veel geheugenruimte voorzien.

Voor een variabele van het type integer worden vaak 2 bytes gereserveerd, dus 16 bits. Eén bit wordt meestal gebruikt voor het teken (+ of -) zodat de waardenverzameling de getallen $\{-32.768, -32.767, \dots, 32.766, 32.767\}$ bevat (65.536 waarden $= 2^{16}$).
($2^{15} = 32.768$)

Indien (zoals in heel wat recentere programmeertalen) meteen 4 bytes worden gereserveerd, dan wordt de waardenverzameling natuurlijk een stuk uitgebreider: $\{-2.147.483.648, -2.147.483.647, \dots, 2.147.483.646, 2.147.483.647\}$ ($4.294.967.296$ waarden $= 2^{32}$).
($2^{31} = 2.147.483.648$)

Het feit dat deze waardenverzamelingen altijd begrensd zijn, heeft ook vervelende consequenties. Het kan bijvoorbeeld zijn dat de som van twee variabelen te groot is om nog in een integer-variabele te passen. De resultaten zijn dan onvoorspelbaar: in sommige omstandigheden wordt een fout gegenereerd, in andere gevallen zal de computer aan het andere eind van de waardenverzameling opnieuw beginnen tellen. In elk geval is het resultaat dan fout, als er al een resultaat gegeven wordt. Men noemt deze situatie **overflow**; het is aan de programmeur om dit soort fouten te voorkomen (lees: onmogelijk te maken).

Een gelijkaardig probleem is trouwens de deling door 0: ook dit zal een fout genereren (i.p.v. de waarde *onbepaald* die je vanuit de wiskunde verwacht).

Voorbeeld overflow

Een programma vraagt om een getal en rekent dan voor jou de primoriaal uit.

De primoriaal van een getal is het product van alle priemgetallen kleiner dan of gelijk aan dat getal.

Zo is de primoriaal van 7, gelijk aan 210 :

$$7\# = 2 * 3 * 5 * 7 = 210$$

Als je voor de primoriaal in je programma het gegevenstype **integer** kiest en dit zorgt voor een geheugenallocatie van **2 bytes**, kan je te maken krijgen met **overflow**.

$$17\# = 510.510$$

Dit is veel groter is als de maximum waarde van je gegevenstype (32.767).

De computer zal tekens hij bij de bovengrens van het gegevenstype komt (32.767) verder tellen vanaf het ondergrens (-32.768).

Voor deze ene berekening zal dit 7 keer gebeuren. Het uiteindelijke resultaat dat je krijgt is 18.990.

3.3.6. Real

De waardenverzameling van het type **real** komt op het eerste zicht overeen met de reële getallen uit de wiskunde (de “kommagetallen” dus), met als beperking natuurlijk dat ook deze waardenverzameling (onderaan en bovenaan) begrensd is. Daarnaast is er ook een beperking op de nauwkeurigheid: er kan maar een begrensd aantal beduidende cijfers voor en/of na de komma getoond worden.

Meestal wordt met een systeem van **floating point** (=drijvende komma) gewerkt: er worden een aantal beduidende cijfers onthouden, en daarnaast wordt onthouden waar zich ergens de komma bevindt. Dat laatste gebeurt door te onthouden met welke macht van 10 het beduidende deel moet worden vermenigvuldigd (de **exponent**). Men spreekt ook van **wetenschappelijke notatie**.

Ter illustratie:

Getal	Wetensch.notatie	Beduidende cijfers	Exponent
148,256	1.48256 E+2	1.48256	$10^2=100$
0,148256	1.48256 E-1	1.48256	$10^{-1}=0,1$

Er worden voor een variabele van het reële type dus een aantal bits gereserveerd voor de exponent en een aantal bits voor de beduidende cijfers (en doorgaans ook een bit voor het teken: 0 voor positieve getallen en 1 voor negatieve getallen).

3.3.7. Array

Je kan ook gebruik maken van **arrays**. Dit is eigenlijk geen echt gegevenstype. Het is een lijst van gelijksoortige gegevens die door een naam en een **index** kunnen worden gekenmerkt. Elk gegeven dat in de array aanwezig is noemen we een element.

Een array is dus een variabele die verwijst naar een reeks van geheugenplaatsen **van hetzelfde type**, die elk een eigen inhoud kunnen krijgen.

In *Programmatielogica - Procedures, functies en arrays* komen we hier uitgebreid op terug.

Voorbeeld:

Zo zou je dus bijvoorbeeld een array kunnen hebben met 7 elementen, waarbij je in elk element een dag van de week zet.

Je array bestaat dus uit 7 elementen van het type string. Er wordt voor elk element genoeg geheugen gereserveerd om een string in op te slaan.

1	2	3	4	5	6	7
'maandag'	'dinsdag'	'woensdag'	'donderdag'	'vrijdag'	'zaterdag'	'zondag'

3.3.8. Keuze van het gegevenstype

Het is van belang om voor je variabelen de **juiste gegevenstypes** te kiezen. Te grote types (bijv. 8 bytes i.p.v. 4 bytes) kan verspilling van geheugenruimte betekenen, maar te kleine types kunnen dan weer problemen veroorzaken met de genoemde overflow-situatie.

Je bent misschien al vertrouwd met een databaseprogramma zoals Microsoft Access. Wanneer je in een database een tabel aanmaakt om gegevens in te stockeren, dan moet je je bij elk veld ook goed afvragen welk gegevenstype en welke veldlengte je best kiest, m.a.w. hoeveel ruimte je voor dat veld reserveert in elk record.

Voor **aantalKinderen** kan je bijvoorbeeld met het kleinste type gehele getallen (een byte, dus gehele getallen van 0 t.e.m. 255) volstaan: een negatief aantal kinderen, of een niet-geheel aantal kinderen, of een aantal kinderen groter dan 255 zijn allemaal onwaarschijnlijke toestanden. Zou je voor dit veld kiezen voor integers (de standaard in Access is lange integer, die 4 bytes in beslag neemt), dan heb je in elk record een heleboel ruimte (namelijk 3 bytes) die je zeker nooit gaat gebruiken. Voor een tabel met 10.000 records betekent dat 30.000 bytes, of bijna 30 kB verspilde ruimte, enkel voor dat éne veld in die éne tabel.

Voor andere gegevens zal je andere veldlengtes moeten kiezen: **huisnummer** kan natuurlijk groter dan 255 zijn, dus een byte volstaat niet. En voor **loon** zal je in dit euro-tijdperk opnieuw kommagetallen moeten voorzien, zodat een real type i.p.v. een integer type moet worden gekozen.

3.3.9. Variabelen declareren

Een gegevenstype toekennen aan een variabele doe je door middel van een **variabele declaratie**. Zet de declaraties van al je variabelen steeds in het begin van je programma.

Niet elke taal verplicht je om je variabelen te declareren. Het is een goede gewoonte om dit toch altijd te doen. Je vermijdt hiermee ook veel problemen.

Een voorbeeld:

BELEGGEN

```
var kapitaal, rente, eindkapitaal: real
var looptijd: integer
```

```
Write 'Kapitaal?'
```

```
Read kapitaal
```

```
Write 'Rente?'
```

```
Read rente
```

```
Write 'Looptijd?'
```

```
Read looptijd
```

```
eindkapitaal := kapitaalberekening(kapitaal, rente, looptijd)
```

```
Write 'Uw eindkapitaal: ', eindkapitaal
```

3.3.10. Variabelentabellen

Bij de ontwikkeling van een PSD kan het nuttig zijn op een stuk papier een tabel bij te houden met alle variabelen die je aanmaakt, een zogenaamde **variabelentabel**. Je geeft daarin aan wat de naam en het type van de variabele zijn, en eventueel enkele andere bijzonderheden.

variabele	type	beschrijving
kapitaal	integer	basiskapitaal
rente	real	rentevoet in %

3.3.11. Toestandstabellen

Daarnaast kan het ook nuttig zijn bij het doorlopen van een PSD elke wijziging in de waarde van de variabelen bij te houden. Op die manier kan je vrij eenvoudig en snel achterhalen waarom je programma op bepaalde punten raar gaat doen en foutmeldingen of foute resultaten toont.

Een voorbeeld: Stel dat we onderstaand PSD hebben:
(Gemakshalve zijn hier de instructies genummerd.)

Dit zou een **toestandstabel** kunnen zijn als de letters A en N ingelezen worden:

var kar1, kar2, kar3: char	
Read kar1	1
Read kar2	2
kar3 := kar1	3
kar1 := kar2	4
kar2 := kar3	5
Write kar1, kar2	6

lezen	na opdracht	kar1	kar2	kar3	schrijven
	begin	?	?	?	
A	1	A	?	?	
N	2	A	N	?	
	3	A	N	A	
	4	N	N	A	
	5	N	A	A	
	6	N	A	A	NA

3.4. Operatoren

In dit hoofdstuk bekijken we de verschillende bewerkingen die we kunnen uitvoeren op variabelen.

3.4.1. Definitie

Operatoren zijn bewerkingen die op één of meer **operanden** kunnen worden toegepast. Operanden kunnen variabelen of constanten of expressies zijn.

Operatoren kunnen symbolen zijn, of gereserveerde woorden. Dit is sterk afhankelijk van programmeertaal tot programmeertaal. Vele operatoren, zoals + en -, zijn in zowat alle programmeertalen hetzelfde, al kan het gebruik en de betekenis (lichtjes) verschillen.

We kunnen de operatoren op verschillende manieren onderverdelen in categorieën: op basis van het **aantal** operanden of op basis van de **soort** operanden waarop ze kunnen worden toegepast.

3.4.2. Unair en binair

Een eerste onderverdeling van de operatoren kan gemaakt worden op basis van het aantal operanden.

Er zijn enkele operatoren die **unair** zijn, d.w.z. dat ze slechts één operand hebben. De operator staat dan doorgaans voor de operand. Voorbeelden hiervan zijn de worteltrekking en het negatief-teken.

- $\sqrt{4}$
- -3.99

De meeste operatoren zijn echter **binair**: ze hebben twee operanden. Meestal staat de operator dan tussen de beide operanden in:

- **getal** + 23
- $27 > 0$

3.4.3. Logische, wiskundige en vergelijkingsoperatoren

We kunnen operatoren ook indelen op basis van het soort bewerkingen dat ze uitvoeren. Meestal heeft dat ook gevolgen voor het gegevenstype van de operanden.

Eenzijds zijn er de **wiskundige operatoren**: de optelling, aftrekking, vermenigvuldiging, worteltrekking en dergelijke meer. Deze kunnen enkel toegepast worden op numerieke variabelen (het gehele of reële gegevenstype).

Daarnaast zijn er de **logische operatoren**: de nevenschikking (and), het alternatief (or) en de ontkenning (not). Zij kunnen enkel op logische waarden worden toegepast.

Tenslotte zijn er de **vergelijkingsoperatoren** (gelijk aan, kleiner dan, ...) die op alle soorten gegevens kunnen worden toegepast.

3.4.4. De verschillende operatoren

We kunnen de operatoren indelen volgens het aantal operanden:

Unaire operatoren

Operator	Type	Betekenis	Voorbeeld
+	numeriek	positief- teken	+27
-	numeriek	negatief- teken	-13
√	numeriek	worteltrekking	√4
...	numeriek	absolute waarde	-23 (=23)
NOT	logisch	ontkenning	NOT (a > 10)

Binaire operatoren

Operator	Type	Betekenis	Voorbeeld
\wedge	numeriek	machtsverheffing	16^2 (=256)
*	numeriek	vermenigvuldiging	$16 * 2$
/	numeriek	deling	$16 / 2$
DIV	numeriek	gehele deling (deling zonder rest)	$16 \text{ DIV } 3$ (=5)
MOD	numeriek	modulus (rest van de gehele deling)	$16 \text{ MOD } 3$ (=1)
+	numeriek	optelling	$16 + 2$
-	numeriek	aftrekking	$16 - 2$
=	vergelijking	gelijkheid	$13 = 15$ (=onwaar)
<>	vergelijking	ongelijkheid	$13 <> 15$ (=waar)
<	vergelijking	kleiner dan	$13 < 15$ (=waar)
>	vergelijking	groter dan	$13 > 15$ (=onwaar)
<=	vergelijking	kleiner dan of gelijk aan	$13 <= 15$ (=waar)
>=	vergelijking	groter dan of gelijk aan	$13 >= 15$ (=onwaar)
AND	logisch	nevenschikking	$(a > 10) \text{ AND } (a < 25)$
OR	logisch	alternatief	$(a < 10) \text{ OR } (a > 25)$

3.4.5. Prioriteitsregels

Elke operator heeft een prioriteit. Operatoren met een **hogere prioriteit** (kleiner getal in het schema hieronder) worden **eerst uitgevoerd**:

$$15 + 3 * 7$$

wordt dus 36 (eerst wordt de vermenigvuldiging uitgevoerd, dan de optelling).

Indien meerdere operatoren **gelijke prioriteit** hebben, worden ze **van links naar rechts** uitgevoerd.

Als je wil afwijken van de prioriteitsregels, gebruik je haakjes:

$$(15 + 3) * 7$$

wordt dus $18 * 7$ of 126.

Je kan zoveel **haakjes** rondom mekaar zetten als je wil; zorg er wel voor dat je elk haakje dat je opent ook weer afsluit!

$$16 * (((7+3)^2) - (15 * 23 / 7))$$

Overzicht:

Operator	Prioriteit
+ - √ ... NOT (unaire operatoren)	1
^	2
/ * DIV MOD	3
+ - (binaire operatoren)	4
= <> < > <= >= (vergelijkingsoperatoren)	5
AND	6
OR	7

3.4.6. Operatoren en gegevenstype

Bij de gewone deling (/) kan je als resultaat zowel een geheel getal als een decimaal getal krijgen.

$$8 / 2 = 4$$

$$7 / 2 = 3.5$$

Je moet dus bij het kiezen van de **gegevenstypes** voor je variabelen ook rekening houden met de **bewerkingen** die je in je programma gaat gebruiken.

Stel je hebt in je programma volgende instructie:

quotiënt := **deeltal** / **deler**

En je moet gaan bepalen welk gegevenstype **deeltal**, **deler** en **quotiënt** hebben.

Afhankelijk van het programmeertaal waarin je werkt, kan het zijn dat je anders moet gaan denken.

- De operanden **deeltal** en **deler**:
var deeltal, deler: integer
 - Sommige talen zullen een fout genereren als je hier de operator '/' op wil uitvoeren. Minstens één van de operanden moet van het type real zijn.
 - Andere talen zullen geen foutmelding geven, maar in plaats van de gewone deling (/) wordt de gehele deling uitgevoerd (DIV).
 - Weer andere talen hechten geen belang aan het gegevenstype van **deeltal** en **deler**.

- Het resultaat:
var quotiënt: integer
 - Ook hier zullen er talen zijn die geen foutmelding geven, maar het resultaat zal natuurlijk zonder decimalen opgeslagen worden.
 - Meestal ga je hier wel een foutmelding krijgen, omdat het resultaat van een deling enkel in een variabele van het gegevenstype real kan gestopt worden.

Als je zeker bent dat je een geheel getal wil als resultaat, gebruik je de operator **DIV**

getalC := 7 DIV 2

Dus **getalC** := 3

3.4.7. Logische uitdrukking

Logica is een studiegebied op zich: heel interessant, maar veel te uitgebreid om hier in detail op in te gaan. Een aantal zaken zijn voor beginnende programmeurs toch wel van belang, en daarom zullen we daarover toch één en ander bekijken.

Een **logische uitdrukking** is een expressie (dus een constante, een variabele, of een combinatie daarvan met operatoren) die als resultaat steeds TRUE of FALSE heeft. Je zou dit ook een binaire expressie kunnen noemen: ze heeft slechts twee mogelijke oplossingen.

Enkele voorbeelden uit het dagelijkse leven:

Het regent.
 Als ik zeven pinten drink, dan ben ik zat.

In de context van programmatielogica zullen logische uitdrukkingen heel vaak een vergelijkingsoperator bevatten, bijv.:

getal1 < **getal2**
naam = 'stop'

Een logische uitdrukking kan ook gewoon een logische variabele zijn (een variabele van het logische gegevenstype). Als je volgende toekenning doet:

varLog := TRUE

dan is **varLog** een logische uitdrukking.

In de meeste programmeertalen hebben deze constanten de waarden TRUE en FALSE.

Ontkenning - NOT

Aangezien een logische uitdrukking slechts twee waarden kan hebben, is het makkelijk in te zien wat het tegengestelde ervan is. Als de uitdrukking TRUE is, is de ontkenning ervan FALSE (en omgekeerd).

Schematisch:

Uitdrukking	Ontkenning
TRUE	FALSE
FALSE	TRUE

Zo'n schematisch overzicht noemt men ook wel een **waarheidstabel**.

Enkele voorbeelden:

Uitdrukking	Ontkenning
Het regent.	Het regent niet.
getal > 10	getal <= 10
varA = varB	varA <> varB

De ontkenning wordt aangeduid met het gereserveerde woord **NOT**.

De ontkenning van **getal** > 10 is dus NOT(**getal** > 10), wat je ook kan schrijven als **getal** <= 10.

Nevenschikking - AND

Met een nevenschikking wordt aangegeven dat twee (of meer) logische uitspraken **tegelijk waar** moeten zijn.

Enkele voorbeelden:

Het regent EN ik heb mijn paraplu bij (ik heb dus geluk)
getal > 10 EN **getal** < 20 (het getal ligt dus tussen 10 en 20, grenswaarden 10 en 20 niet inbegrepen)

Een nevenschikking is enkel waar als BEIDE leden waar zijn! Of anders gezegd: wanneer minstens één van beide leden onwaar is, is de nevenschikking onwaar.

Schematisch:

Uitdrukking 1	Uitdrukking 2	Nevenschikking
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Je kan de betekenis hiervan eens proberen na te gaan met bijv.

uitdrukking 1: Het regent
 uitdrukking 2: De zon schijnt
 (als dit waar is, zie je een regenboog...)

of met:

uitdrukking 1: **getal** < 10
 uitdrukking 2: **getal** > 20
 (dit kan wiskundig gezien nooit waar zijn).

Alternatief - OR

Met een alternatief wordt aangegeven dat minstens één van twee (of meer) uitdrukkingen waar is.

Als ik zeg “het regent OF de zon schijnt”, dan is het mogelijk dat

- het enkel regent en er geen zon te zien is;
- of het kan zijn dat de zon schijnt en het helemaal niet regent;
- maar het kan ook zijn dat het regent en dat de zon schijnt.

Enkel als beide uitdrukkingen FALSE zijn wordt het alternatief ook FALSE.

Schematisch:

Uitdrukking 1	Uitdrukking 2	Alternatief
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

Je kan de betekenis hiervan eens proberen na te gaan met bijv.

uitdrukking 1: “Het regent”
 uitdrukking 2: “De zon schijnt”
 (dit is enkel onwaar als het niet regent EN de zon niet schijnt)

of met:

uitdrukking 1: **getal** < 10
 uitdrukking 2: **getal** > 20
 (dit is voor alle getallen waar, behalve voor de getallen tussen 10 en 20, grenswaarden 10 en 20 inbegrepen).

3.5. Opgaven2

Maak de oefeningen in de webcursus

Hoofdstuk 4. Teken van diagrammen

4.1. Structorizer

Je kan alle diagrammen natuurlijk met pen en papier aanmaken, maar misschien vind je het toch handiger een aparte tool te gebruiken.

Enerzijds zijn er gewone “tekenprogramma’s” die je toelaten de diagrammen netjes te maken, te bewerken, om te zetten naar een afbeelding enz. Een voorbeeld hiervan is Visio van Microsoft.

Anderzijds zijn er meer gespecialiseerde tools die bijvoorbeeld ook in staat zijn syntax te controleren, code te genereren in een of meerdere programmeertalen, enz. Een voorbeeld uit deze categorie is **Structorizer**.

Met het programma Structorizer gaan we

1. programmastructuurdiagrammen (PSD's) maken
2. Pascalcode genereren die we dan kunnen testen met behulp van het programma Lazarus

In dit onderdeel gaan we Structorizer downloaden en bekijken we ook hoe je het programma moet gebruiken.

4.1.1. Installatie

Je kan Structorizer gratis downloaden:

- Surf naar <http://structorizer.fisch.lu/>
- Klik op de knop *Downloads*
- Download de laatste Java-versie die correspondeert met jouw besturingssysteem en bewaar de zipfile op je computer.

Er is een versie voor Windows & Linux en een versie voor Mac.

Om Structorizer te kunnen gebruiken, moet je de zipfile ergens uitpakken. Afhankelijk van je besturingssysteem, vertrek je vanuit een ander bestand:

Windows

Om het programma te starten, volstaat het om te dubbelklikken op **structorizer.exe**. Om plaats te besparen op je computer, kan je het bestand *Structorizer.sh* verwijderen. Dit heb je niet nodig.

Linux

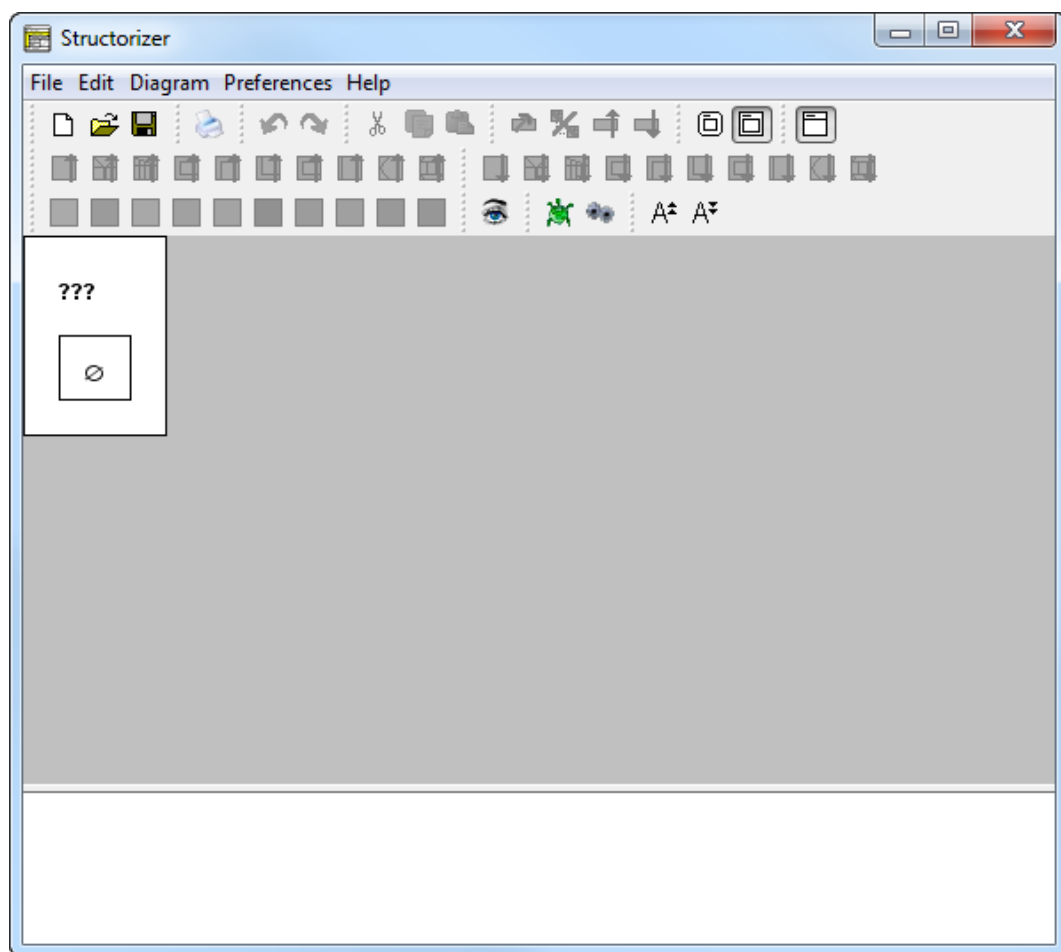
Om het programma te starten, volstaat het om te dubbelklikken op **structorizer.sh**. Om plaats te besparen op je computer, kan je het bestand *Structorizer.exe* verwijderen. Dit heb je niet nodig.

Mac OSX

Verplaats de Structorizer applicatie naar je applicatiefolder of start het programma vanuit de map waar je het programma uitgepakt hebt. Als je plaats wil besparen op je computer, kan je de bestanden *structorizer.exe* en *structorizer.sh* verwijderen.

Je kan ook de Java Web Start versie downloaden. Om dan Structorizer te starten, volstaat het om te dubbelklikken op het bestand **Arranger.jnlp**.

Als je het programma opstart, krijg je het volgende scherm:

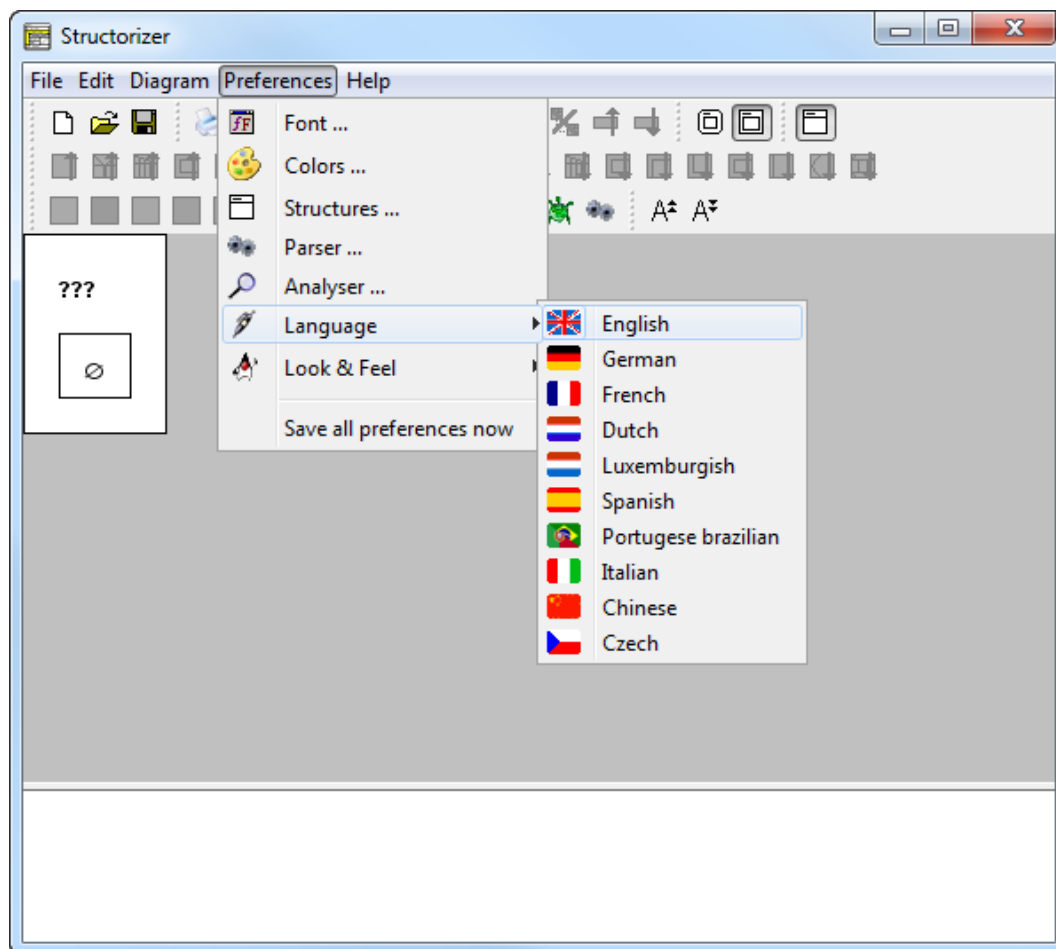


4.1.2. Configuratie

Alvorens je met Structorizer aan de slag kan gaan, moet je een aantal dingen juist instellen:

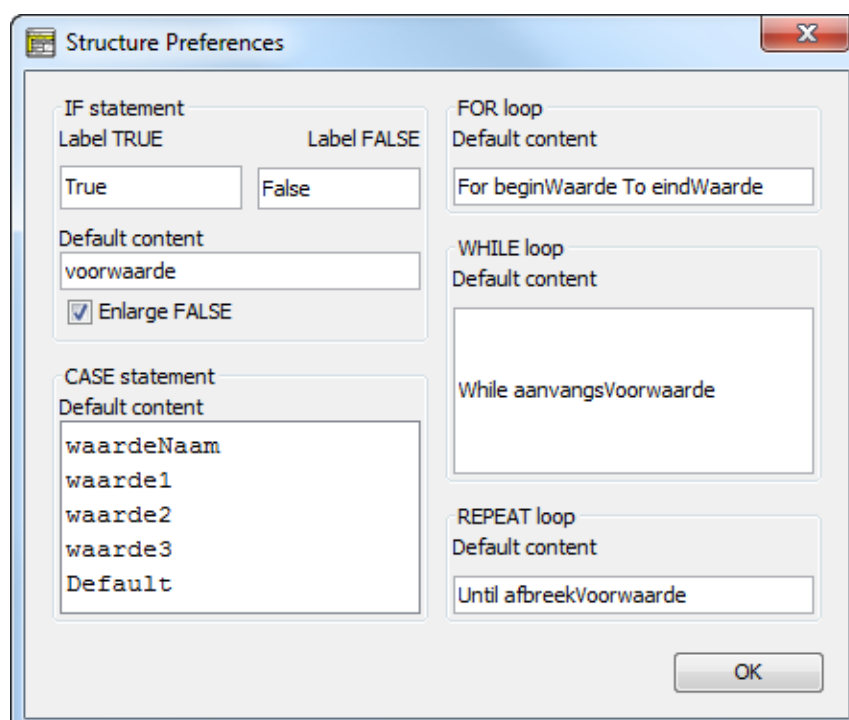
Taal instellen

Mocht het nog niet zo zijn, stel de taal dan in op Engels:



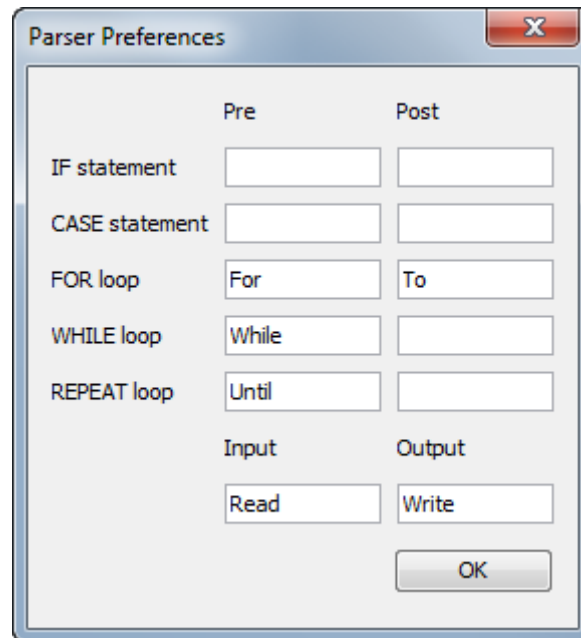
Structuren definiëren

Via het menu **Preferences** en dan de optie **Structures** bepaal je wat er straks als standaardtekst weergegeven zal worden. Zorg dat alles ingevuld is volgens onderstaande afbeelding.



Vertaler configureren

Als je straks je PSD wil omzetten in code, moet je aangeven wat op welke manier 'vertaald' moet worden. Als je dit niet doet, zal de code die je krijgt niet uitvoerbaar zijn. Ga naar **Preferences** > **Parser** en stel ook hier alles in zoals in onderstaande afbeelding.



4.1.3. Gebruik

Je kan op internet heel wat informatie vinden over het gebruik van Structorizer. Als je in de loop van de cursus op problemen stuit, kan je altijd op volgende website een kijkje gaan nemen: [site van Structorizer](http://structorizer.fisch.lu/) (<http://structorizer.fisch.lu/>) en kies dan links **User Guide**

Hieronder vind je de belangrijkste zaken terug. Als je verder in de cursus iets speciaals nodig hebt, zal dit op dat moment uitgelegd worden.

Knop(pen)	Betekenis
	voegt een item toe boven het geselecteerde item
	voegt een item toe onder het geselecteerde item
	verplaatst het geselecteerde item één item naar boven / naar beneden

Nog enkele richtlijnen:

- De ??? bovenaan vervang je door de programmanaam door erop te dubbelklikken.
 - de programmanaam zet je in **UpperCamelCase** (dus met beginhoofdletter)
 - deze mag niet dezelfde zijn als één van de variabelen (alleen bij een subroutine mag dit wel)
 - vermijd spaties
- Voor de namen van variabelen gebruiken we **lowerCamelCase**.

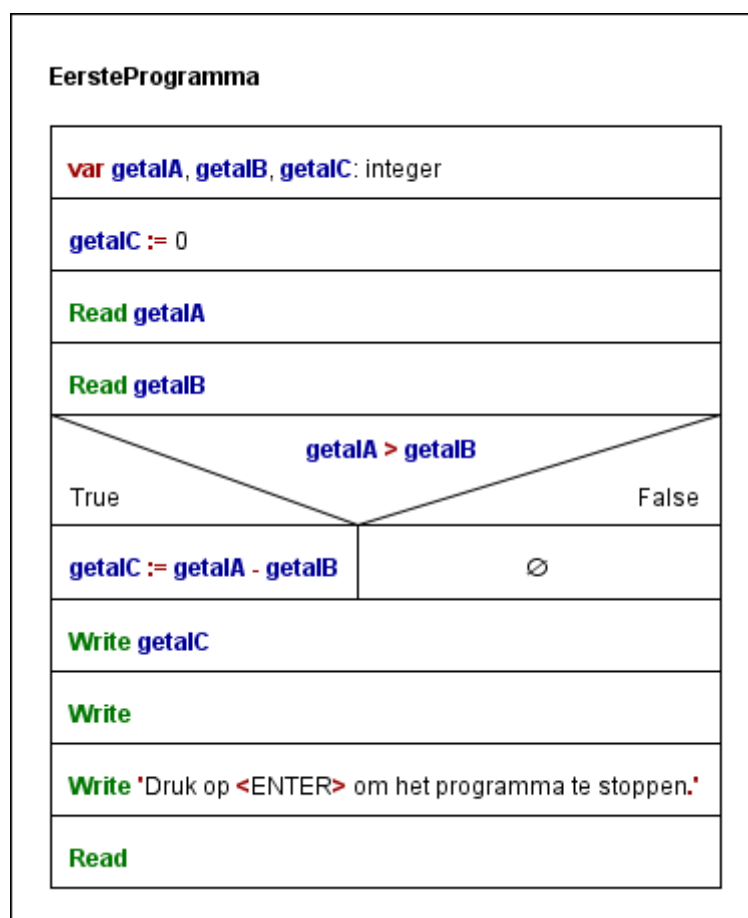
- De declaraties van de variabelen zet je bovenaan in je PSD.
- Let er op dat elke variabele geïnitieerd wordt.
- Om een item van je PSD aan te passen, dubbelklik je erop. Het invulvenster zal dan weer getoond worden.
- Om een item te verwijderen, selecteer je het en druk je op *Delete*.
- Om je PSD te **exporteren**:
 - kies je in het menu *File* de optie *Export*
 - kies je de optie *Code*
 - tenslotte kies je *Pascal/Delphi*
 - geef een bestandsnaam en plaats
 - bevestig met *Save*.
- Van de basishandelingen, zoals nieuw, openen, opslaan, kopiëren, plakken veronderstellen we dat deze gekend zijn.

4.2. Opgaven1

De oplossingen van deze opgaven vind je terug in de webcursus.

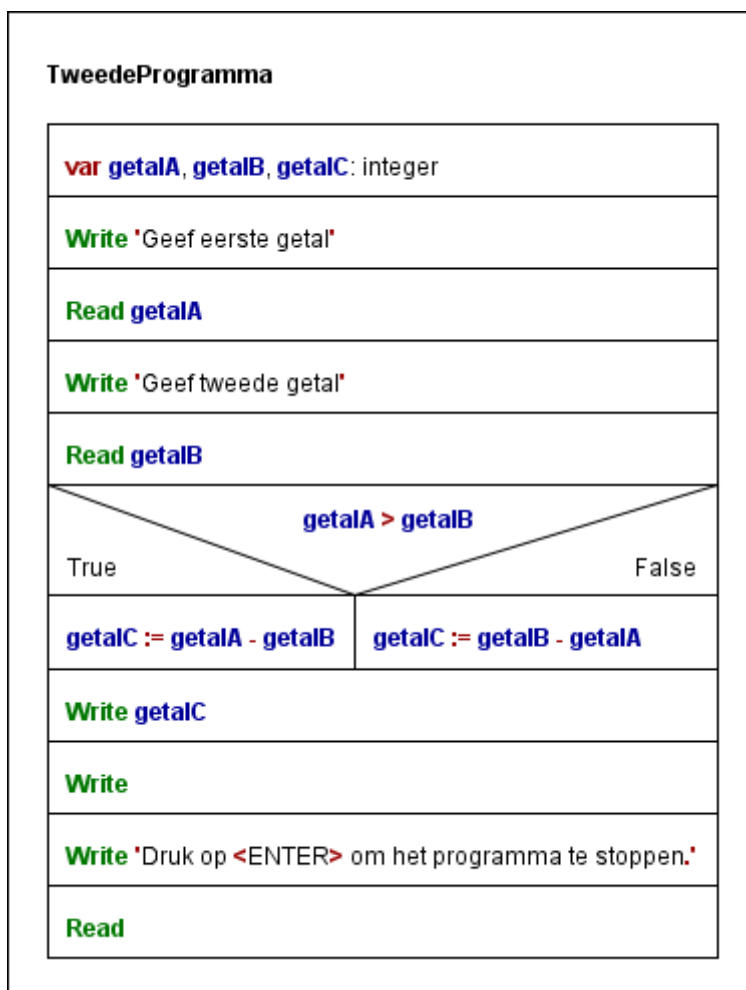
4.2.1. Opgave 1

Probeer onderstaand PSD na te maken in Structorizer. Bewaar je resultaat onder de naam *EersteProgramma.nsd*.



4.2.2. Opgave 2

Open, indien nodig, het bestand *EersteProgramma.nsd*. Doe de nodige aanpassingen zodat je onderstaand programma krijgt. Bewaar je resultaat onder de naam *TweedeProgramma.nsd*.



4.3. Lazarus

Met Structorizer kan je een PSD maken. Met Lazarus kan je controleren of je PSD doet wat het moet doen.

In dit onderdeel bekijken we

- waar je Lazarus kan downloaden,
- hoe je het moet installeren en
- hoe je je code dan kan testen.

4.3.1. Installatie

Belangrijk voor Mac-gebruikers !!!

Er zijn een aantal stappen die je moet uitvoeren voor je Lazarus kan installeren. De **Xcode Command Line Tools** moeten geïnstalleerd zijn

- Vanaf Mac OS X 10.9 (Mavericks):

- Open hiervoor de Terminal en typ volgende instructie:
xcode-select --install
- Je hebt **gdb** nodig om in Lazarus te kunnen debuggen.
Dit wordt niet meer mee geïnstalleerd vanaf MacOS X 10.9 (Mavericks) en XCode 5.

Controleer of volgende bestanden aanwezig zijn:

- */usr/bin/gdb*
- */usr/libexec/gdb/gdb-i386-apple-darwin*

Als dit niet het geval is, download de bestanden in de webcursus en zet ze op de juiste plaats.

- Werk je met een eerder versie van Mac OS:
 - dan kan je meer informatie vinden
http://wiki.freepascal.org/Installing_Lazarus_on_MacOS_X.
Als het je lukt te werken met XCode 4.6.3, dan worden de **gdb**-bestanden automatisch op de juiste plaats gezet.
 - Bekijk ook het YouTube-filmpje via deze link:
https://www.youtube.com/watch?v=CN0o_qX4A8c

Nadien moet je drie files downloaden (kies de juiste versie: intel of powerpc):

- fpc
- fpcsrc
- lazarus

Deze bestanden moeten ook **in deze volgorde geïnstalleerd** worden.

Algemeen

Je kan het programma Lazarus gratis downloaden op de lazarus.freepascal.org. Kies de juiste versie (Windows, Linux of Mac). Bewaar het bestand op je PC. Dubbelklik op het bestand en volg de stappen van de installatiewizard. Na installatie kan je op je bureaublad het icoontje van Lazarus terugvinden:



Je kan heel wat handleidingen terugvinden op de volgende websites:

- [handleiding lazarus](#)
- [verzameling handleidingen lazarus](#)

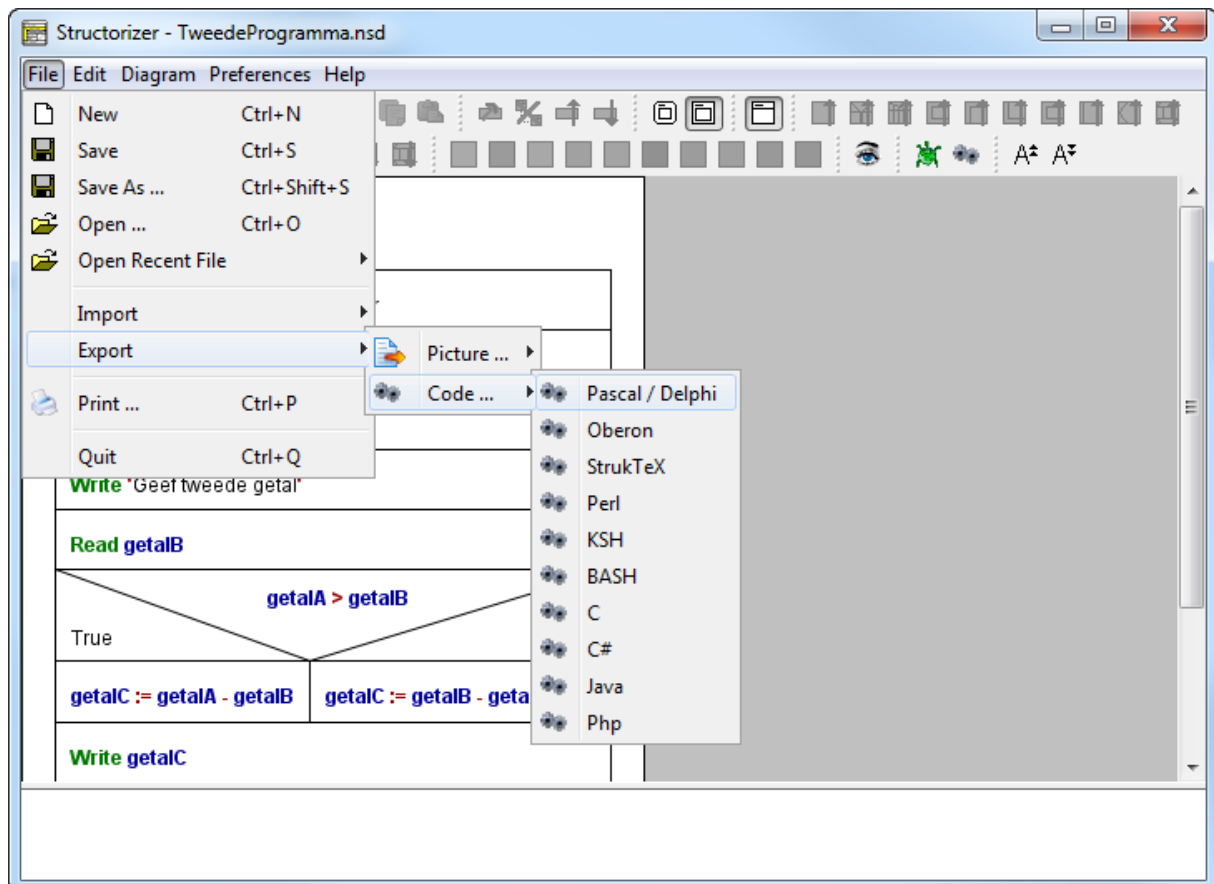
4.3.2. Gebruik

Om je PSD te testen in Lazarus moet je:

- het exporteren uit Structorizer naar *Pascal/Delphi*
- het geëxporteerde bestand openen in Lazarus

Exporteren uit Structorizer

Open het menu **File** en kies voor **Export > Code ... > Pascal / Delphi**



Geef een bestandsnaam en bepaal waar het bestand bewaard moet worden. Klik op **Save**.

Openen in Lazarus

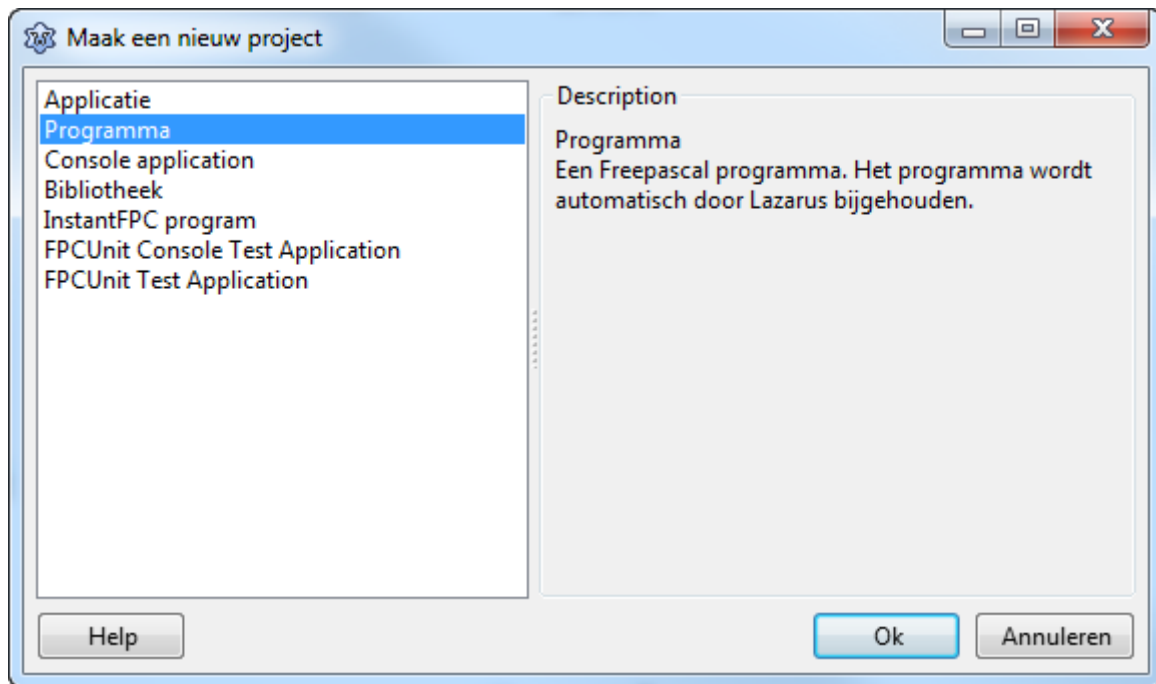
Open Lazarus en kies in het menu **Bestand** voor **Openen ...**

Blader naar de juiste map, selecteer je .pas-bestand en klik op **Openen**.

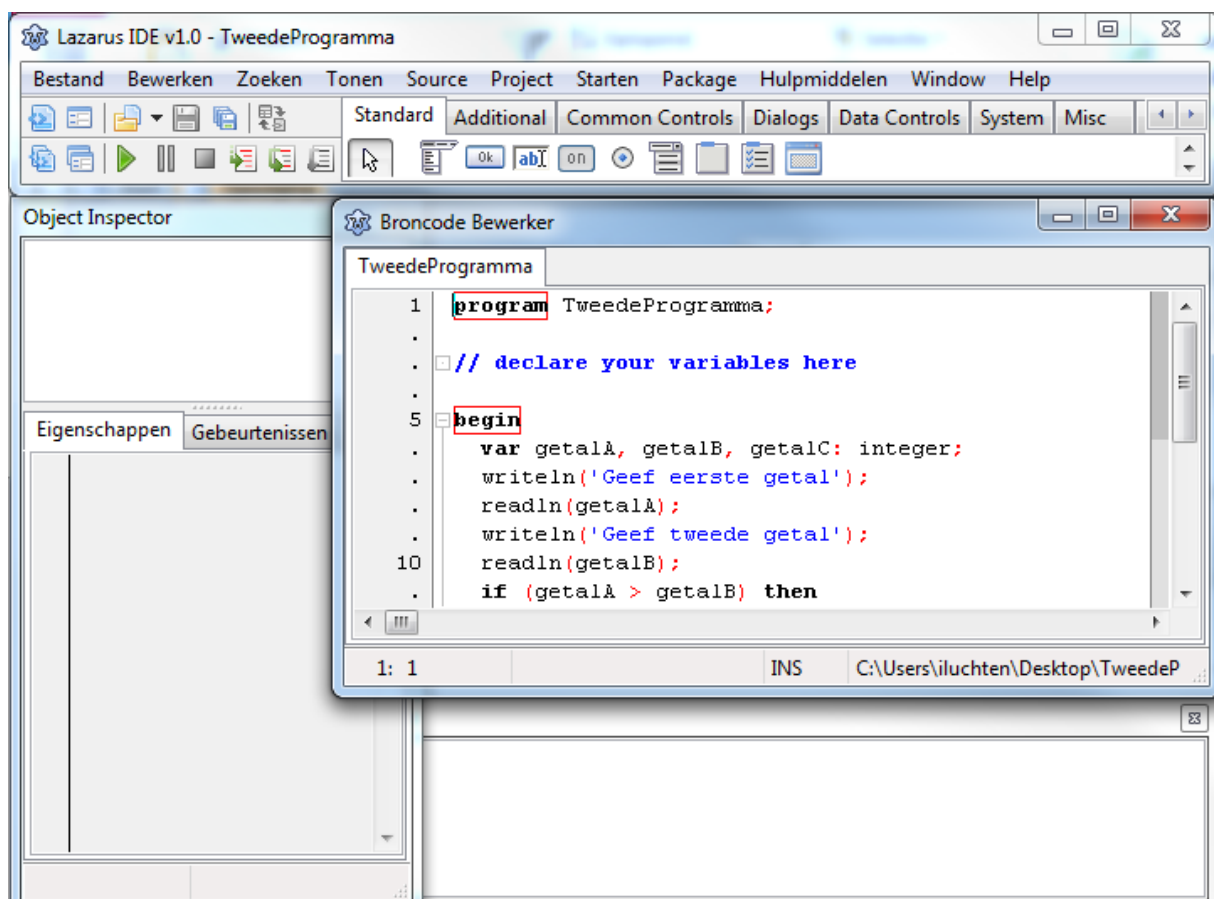
Je krijgt onderstaand venster. Bevestig met **Ja**.



In het scherm *Maak een nieuw project* kies je voor de optie **Programma** en bevestig met **Ok**.



Resultaat



4.3.3. Variabelen declareren

De variabelendeclaratie moet tussen de naam van het programma en het woordje **'begin'** komen te staan.

Bij het exporteren uit Structorizer staat dit nog niet goed:

```
program TweedeProgramma;

// declare your variables here


begin
  var getalA, getalB, getalC: integer;
  writeln('Geef eerste getal');
  readln(getalA);
  writeln('Geef tweede getal');
```

Verplaats het naar de juiste plaats:

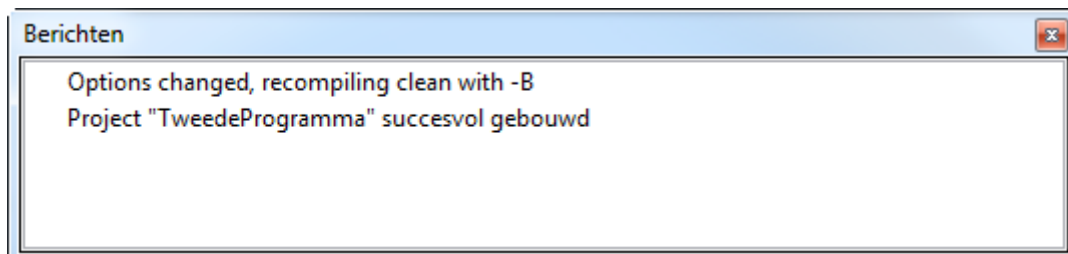
```
program TweedeProgramma;

// declare your variables here
var getalA, getalB, getalC: integer;

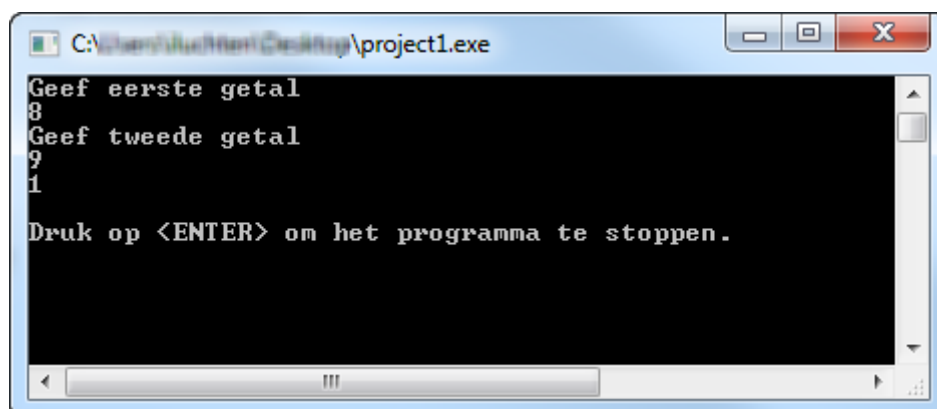
begin
  writeln('Geef eerste getal');
  readln(getalA);
  writeln('Geef tweede getal');
```

 Klik op de knop **Starten** om het programma te starten. Eerst zal lazarus je code controleren op syntaxfouten. Nadien wordt de code uitgevoerd.

Als er fouten zijn, worden die onderaan in een *Berichtvenster* getoond.



Het resultaat van het programma wordt in een zwart dialoogvenster getoond.



4.4. Opgaven2

4.4.1. Opgave 1

Probeer het programma *TweedeProgramma* uit te testen in Lazarus:

- exporteer de code
- zet de declaratie van je variabelen op de juiste plaats
- voer het programma uit

Bekijk de animatie met de oplossing in de webcursus.

Hoofdstuk 5. Basisstructuren

We onderscheiden drie basisstructuren om een gestructureerd programma op te stellen:

- de **sequentie**:
een opeenvolging van procesinstructies; deze worden gewoon één na één uitgevoerd, waarbij de volgorde van belang kan zijn.
- de **selectie**:
de keuze tussen twee (of eventueel meer) verschillende reeksen procesinstructies, op basis van een voorwaarde (selectie-criterium).
- de **iteratie**:
de herhaling van een reeks procesinstructies.

In dit hoofdstuk bekijken we de sequentie en maken we hierop enkele oefeningen.

5.1. Sequentie

Sequentie is één van de drie basisstructuren van programmatielogica. Het is de meest elementaire basisstructuur, nl. een opeenvolging van meerdere instructies. Alle voorbeelden die we tot hier toe zagen, waren eenvoudige sequenties. In een PSD wordt dit weergegeven als een aantal rechthoeken onder elkaar; elke rechthoek staat voor één proces-instructie.

5.1.1. Voorbeeld 1

Het onderstaand voorbeeld berekent het eindkapitaal van een belegging na 5 jaar. Eerst wordt het beginkapitaal en de rentevoet gevraagd (5% geef je in als 5). Vervolgens wordt het kapitaal ieder jaar verhoogd met de rente.

Het is duidelijk dat dit programma langer en langer gaat worden naarmate de belegging langer loopt. Nochtans zal er wezenlijk niets veranderen: voor elk bijkomend jaar zal dezelfde instructie steeds opnieuw worden tussengevoegd.

Eindkapitaal

```
var kapitaal, rentevoet: real
```

```
Write 'Beginkapitaal?'
```

```
Read kapitaal
```

```
Write 'Rentevoet?'
```

```
Read rentevoet
```

```
kapitaal := kapitaal * (1 + rentevoet / 100)
```

```
kapitaal := kapitaal * (1 + rentevoet / 100)
```

```
kapitaal := kapitaal * (1 + rentevoet / 100)
```

```
kapitaal := kapitaal * (1 + rentevoet / 100)
```

```
kapitaal := kapitaal * (1 + rentevoet / 100)
```

```
Write 'Eindkapitaal na 5 jaar: ', kapitaal:0:2
```

```
Write
```

```
Write 'Druk op <ENTER> om het programma te stoppen.'
```

```
Read
```

5.1.2. Voorbeeld 2

Met onderstaand structogram gaan we:

- uurloon en aantal uren inlezen
- het brutoloon berekenen
- het brutoloon tonen



Opdracht

1. Maak het structogram na in Structorizer
2. Exporteer de code in Pascal-formaat.
3. Open het pas-bestand in Lazarus.
4. Test je code uit.
Vb. uurloon: € 12.50 en aantal uren: 38 → € 475.00

Bekijk zeker ook de animaties in de webcursus.

5.2. Opgaven1

In dit onderdeel maak je 5 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!

5.2.1. Opgave 1: Kwadraat

Vraag de gebruiker om een getal. Lees dat getal in en bereken het kwadraat. Toon het kwadraat.

5.2.2. Opgave 2: Som en gemiddelde

Lees twee willekeurige getallen in. Bereken de som en het gemiddelde van die twee getallen, en toon die resultaten.

5.2.3. Opgave 3: Loon

Lees het aantal uren, het aantal overuren en het bruto uurloon in.

Bepaal het effectieve brutoloon, als je weet dat overuren aan 150% worden uitbetaald.

(Bijv.: 38 uren, 6 overuren, € 25,00 uurloon geeft een brutoloon van € 1 175,00)

5.2.4. Opgave 4: Duurtijd

Lees een begintijd in (uren en minuten afzonderlijk).

Lees een duurtijd in (ook in uren en minuten).

Bereken dan de eindtijd, in uren en minuten.

5.2.5. Opdracht voor je coach: Draagkracht

Een stalen balk met een hoogte en breedte, die voor een lengte uit een (stevige) constructie steekt, kan aan het uiteinde maximaal een bepaald gewicht dragen. Dat gewicht wordt berekend als

$$gewicht = \frac{19 \times breedte \times hoogte^2}{lengte}$$

Hierbij worden de lengtematen breedte, hoogte en lengte in cm uitgedrukt en het gewicht in kg.

Vraag:

Maak een PSD dat de drie nodige maten inleest en het maximale gewicht berekent en toont.

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

5.3. Selectie

De **selectie** is de tweede basisstructuur van programmatielogica.

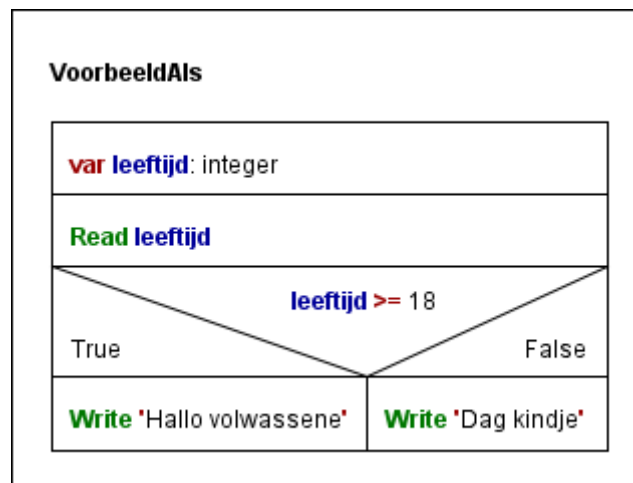
Bij een selectie of een keuzediagram wordt een andere reeks instructies uitgevoerd al naargelang al dan niet aan een voorwaarde voldaan is. Dit wordt algemeen weergegeven onder de vorm van een IF... THEN ... ELSE ... redenering:

```

IF (als) aan een bepaalde voorwaarde is voldaan
THEN (dan)
  wordt deze eerste reeks instructies uitgevoerd
ELSE (anders)
  wordt deze tweede reeks instructies uitgevoerd
END IF (einde als)
  
```

Het "ELSE ..." gedeelte kan eventueel worden weggelaten. Indien niet aan de voorwaarde is voldaan, wordt naar de eerste instructie na de selectie gesprongen.

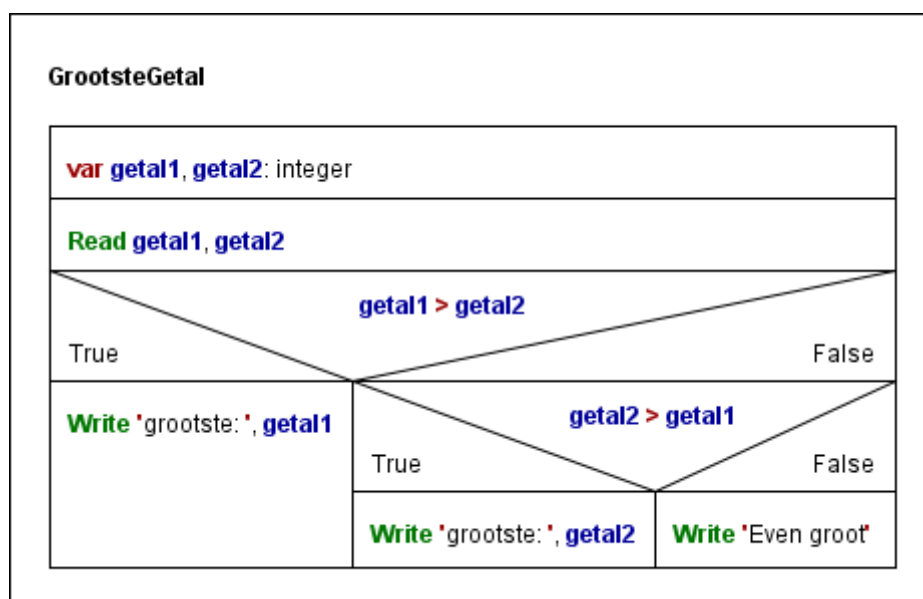
In een PSD ziet dit er als volgt uit:



Afhankelijk van de waarde van de uitdrukking **leeftijd >= 18**, een logische test die altijd TRUE of FALSE oplevert, wordt een ander deel van de programmacode uitgevoerd.

Als er alleen iets moet gebeuren wanneer de voorwaarde waar is, laat je FALSE-blok leeg. Andersom kan ook: als er alleen iets moet gebeuren wanneer de voorwaarde niet waar is, blijft het TRUE-blok leeg.

Een ander voorbeeld:



Indien het eerste ingelezen getal het grootste is, wordt dat afgedrukt; indien het tweede het grootste is, wordt dat afgedrukt; en in het andere geval zijn ze even groot en wordt een passende boodschap afgedrukt.

5.3.1. Samengestelde voorwaarde

In voorgaande voorbeelden hebben we enkel eenvoudige voorwaarden gebruikt: bvb. **leeftijd >= 18**.

Maar in de echte wereld zijn er acties die enkel mogen uitgevoerd worden indien er voldaan wordt aan **meerdere voorwaarden**.

Bekijk even onderstaand voorbeeld:

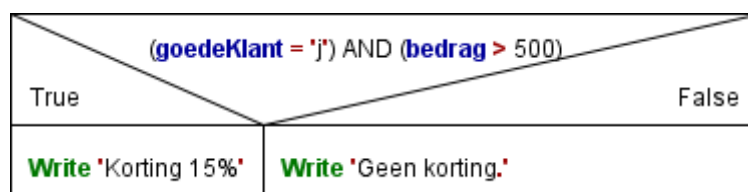


Om een korting te krijgen moet er voldaan worden aan 2 voorwaarden:

- genoteerd staan als “goede klant” en
- voor meer dan € 500,00 aankopen doen.

Dit noemen we een **samengestelde voorwaarde**.

Niet onbelangrijk: merk op dat er slechts 2 mogelijke uitkomsten zijn: *korting 15%* of *geen korting*. In dat geval kunnen we de 2 voorwaarden samen plaatsen in één selectieblok:



De voorwaarden worden verbonden met het woordje and. Dit geeft aan dat beide voorwaarden true moeten zijn alvorens het True-blok wordt uitgevoerd. In alle andere gevallen wordt het False-blok uitgevoerd.

Naast het verbindingswoord AND bestaat er ook nog het verbindingswoord OR. OR geeft aan dat tenminste één van beide voorwaarden TRUE moet zijn om het TRUE-blok uit te voeren. Het FALSE-blok wordt enkel uitgevoerd als geen enkele voorwaarde TRUE is.

Schematisch ziet er dit als volgt uit:

Voorwaarde1		Voorwaarde2	Resultaat	
<code>goedeKlant = 'j'</code>		<code>bedrag > 500</code>		
TRUE	AND	TRUE	TRUE	TRUE-blok
TRUE	AND	FALSE	FALSE	FALSE-blok
FALSE	AND	TRUE	FALSE	FALSE-blok
FALSE	AND	FALSE	FALSE	FALSE-blok
TRUE	OR	TRUE	TRUE	TRUE-blok
TRUE	OR	FALSE	TRUE	TRUE-blok
FALSE	OR	TRUE	TRUE	TRUE-blok
FALSE	OR	FALSE	FALSE	FALSE-blok

Een samengestelde voorwaarde kan bestaan uit **meer dan 2 voorwaarden**. Zorg er voor dat alle voorwaarden verbonden zijn met and of or.

Een aantal voorbeelden:

- `(leeftijd > 18) and (goedeKlant = 'j') and (bedrag > 500)`
- `(leeftijd > 18) or (goedeKlant = 'j')`
- `(leeftijd > 18) or (goedeKlant = 'j') and (bedrag > 500)`

AND en OR worden logische operatoren genoemd, en kennen net zoals rekenkundige operatoren (+, -, /, *) een voorrangsregeling. Zo worden de voorwaarden verbonden met AND eerst uitgewerkt en vervolgens de voorwaarden verbonden met OR. Afwijken kan door het gebruik van haakjes.

5.4. Opgaven2

In dit onderdeel maak je 7 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!

5.4.1. Opgave 1: Temperatuur

Tussen een temperatuur gemeten in Kelvin en een temperatuur gemeten in graden Celsius bestaat het volgende verband:

$$\text{celsius} := \text{kelvin} - 273,15$$

Hierbij mag **kelvin** niet negatief zijn. Ontwerp een programma dat een temperatuur in Kelvin inleest en de temperatuur in graden Celsius afdruckt.

5.4.2. Opgave 2: Factuur

Lees een eindbedrag (bijv. van een factuur) in.

Indien het bedrag groter is dan € 5 000,00, dan wordt een korting van 3% toegestaan.

Toon het uiteindelijk te betalen bedrag.

5.4.3. Opgave 3: Rechthoek

Lees de oppervlakte en de breedte van een rechthoek in.

Bereken de lengte van de rechthoek (oppervlakte = lengte x breedte).

5.4.4. Opgave 4: Examens

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

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.

5.4.5. Opgave 5: Deling

Lees twee getallen in.

Deel het grootste door het kleinste en toon het resultaat op het scherm.

5.4.6. Opgave 6: Eenvoudige sortering

Lees drie getallen in.

Rangschik ze van groot naar klein en druk het resultaat af.

5.4.7. Opdracht voor je coach: Kindergeld

Een moeder heeft recht op € 25,00 kindergeld per kind. Voor het derde kind (en elk volgend kind) krijgt ze een toeslag van € 12,50. Voor het vijfde (en elk volgend) kind krijgt ze nog eens een toeslag van € 7,50.

Als het maandloon van de moeder kleiner is dan of gelijk aan € 500,00, dan krijgt ze 25% toeslag op het kindergeld. Maar als haar maandloon groter is dan € 2000,00, dan krijgt ze 25% minder kindergeld.

Minimaal heeft een moeder altijd recht op € 25,00 per kind.

Vraag:

Lees het aantal kinderen en maandloon in, en toon het kindergeld waar de moeder recht op heeft.
(Bijv. : 6 kinderen en € 1 500,00 maandloon geeft € 215,00 kindergeld)

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

5.5. Iteratie

De **iteratie** is de derde en laatste basisstructuur van programmatielogica.

Om te vermijden dat we bij een belegging over 50 jaar vijftig herhalingen van dezelfde instructie onder mekaar zouden moeten zetten, kunnen we gebruik maken van een iteratie of herhalingsconstructie.

met iteratie	zonder iteratie
<div>Eindkapitaal</div> <div> <pre> var kapitaal, rentevoet: real Write "Beginkapitaal?" Read kapitaal Write "Rentevoet?" Read rentevoet Voer onderstaande code 50 keer uit kapitaal := kapitaal * (1 + rentevoet / 100) Write "Eindkapitaal na 50 jaar: ", kapitaal:0:2 Write Write "Druk op <ENTER> om het programma te stoppen." Read </pre> </div>	<div>Eindkapitaal</div> <div> <pre> var kapitaal, rentevoet: real Write "Beginkapitaal?" Read kapitaal Write "Rentevoet?" Read rentevoet kapitaal := kapitaal * (1 + rentevoet / 100) kapitaal := kapitaal * (1 + rentevoet / 100) kapitaal := kapitaal * (1 + rentevoet / 100) </pre> <div>Deze instructie wordt 50 keer herhaald</div> <pre> kapitaal := kapitaal * (1 + rentevoet / 100) kapitaal := kapitaal * (1 + rentevoet / 100) Write "Eindkapitaal na 50 jaar: ", kapitaal:0:2 Write Write "Druk op <ENTER> om het programma te stoppen." Read </pre> </div>

Eigenlijk houdt dat in dat we aangeven dat een bepaalde instructie (of reeks instructies) een **aantal keer moet herhaald** worden. In plaats van "50" hadden we hier ook een variabele kunnen opgeven, zodat pas bij het uitvoeren van het programma bepaald zal worden hoe vaak de code moet worden herhaald.

In een PSD wordt de herhaling weergegeven met een winkelhaak-diagram, zoals op bovenstaand schema te zien is. De horizontale balk kan boven- en/of onderaan komen, al naargelang welk soort iteratie gewenst is. We komen hier later op terug.

5.5.1. For ... to ...

Een eerste vorm van iteratie is de **FOR-lus**: hierbij wordt aangegeven dat een **reeks instructies een exact aantal keren herhaald** moet worden. Dat gebeurt aan de hand van een teller die oploopt van 1 tot **aantal**. Het doen oplopen van de **teller** wordt door de computer afgehandeld: daar is geen extra programma-code voor nodig.

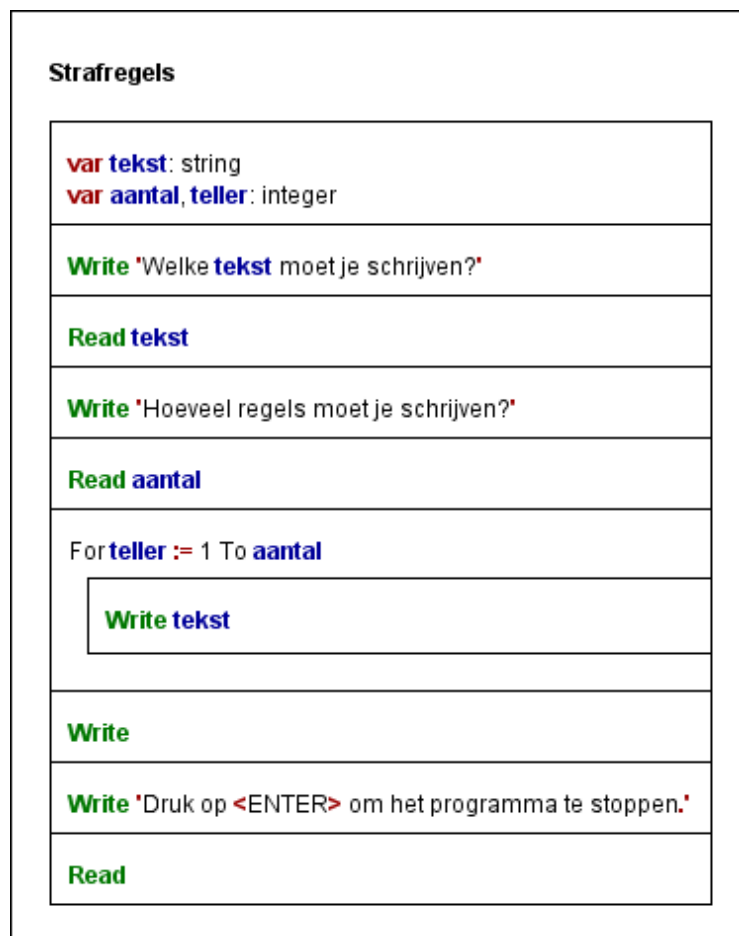
Hieronder zie je een voorbeeld van een FOR-lus die 10 keer doorlopen wordt.



Voorbeeld

Je hebt strafwerk gekregen. Je moet een bepaalde zin een aantal keer schrijven. Laat de computer dit werk voor je doen.

Oplossing



5.5.2. While ...

Een tweede vorm van iteratie is de **WHILE-lus**. Hierbij wordt **eerst** getest of aan een **bepaalde (logische) voorwaarde** is voldaan. Als dat het geval is, worden de instructies binnen de lus uitgevoerd. Voor de lus opnieuw start, wordt de test opnieuw uitgevoerd. Vanaf het moment dat de voorwaarde NIET meer waar is, wordt de lus beëindigd.

In onderstaand voorbeeld wordt een getal ingelezen. Zolang dit getal kleiner of gelijk is aan 10, worden er een aantal instructies uitgevoerd. Tenslotte wordt een nieuw getal ingelezen. Als het getal groter is dan 10, stopt de iteratie.



Voorbeeld

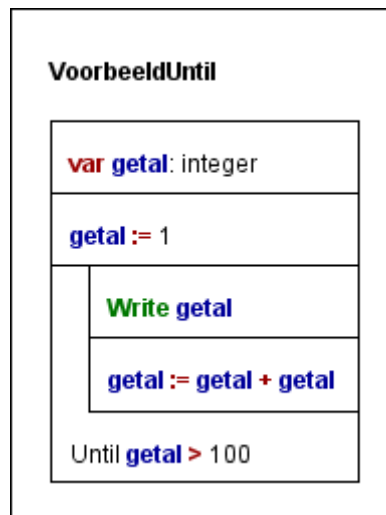
Bij het inlezen van gegevens weet je nooit exact wat je gaat krijgen. Daarom is het belangrijk om de waarde die je krijgt te testen. Lees een leeftijd in en controleer of je een geldige waarde hebt binnengekregen.

Oplossing



5.5.3. Until ...

Een derde iteratie-vorm, de **UNTIL-lus**, plaatst de **test achteraan**, en wordt herhaald totdat de test WAAR is.



Deze constructie is grotendeels analoog met de WHILE, enkel wordt in dit geval de reeks instructies binnen de lus MINSTENS 1 keer uitgevoerd, terwijl bij de WHILE de mogelijkheid bestaat dat de voorwaarde al van bij het begin onwaar is en de lus dus geen enkele keer wordt uitgevoerd.

Voorbeeld

Gebruik een totdat-lus om een rechthoekige driehoek van sterretjes te tekenen. In onderstaand voorbeeld lezen we het aantal sterretjes in dat op de eerste regel geschreven wordt. In elke volgende regel wordt telkens een sterretje minder geschreven.

Oplossing

DriehoekTeken

```
var zijde, teller: integer
```

```
Write 'Geef de lengte van de rechthoekszijde.'
```

```
Read zijde
```

```
While (zijde < 0) or (zijde > 50)
```

```
    Write 'Geef een waarde tussen 0 en 50'
```

```
    Read zijde
```

```
Write 'Een rechthoekige driehoek: '
```

```
Write
```

```
    For teller:=1 To zijde
```

```
        Write '*'
```

```
    Write
```

```
    zijde := zijde - 1
```

```
Until zijde <= 0
```

```
Write
```

```
Write 'Druk op <ENTER> om het programma te stoppen.'
```

```
Read
```


5.5.4. Vergelijking

Wat zijn de verschillen tussen de verschillende iteraties? Wat gebruik je wanneer?

	While ...	Until ...	For ... To ...
Aantal herhalingen	vooraf niet gekend mogelijk dat code niet uitgevoerd wordt	vooraf niet gekend code wordt minstens 1 maal uitgevoerd	vooraf gekend code wordt welbepaald aantal keren uitgevoerd
Einde herhaling	als de voorwaarde niet meer voldaan is	als de voorwaarde voldaan is	automatisch
Oneindige lus mogelijk?	ja	ja	nee

5.6. Opgaven3

In dit onderdeel maak je 7 oefeningen. De oplossingen van deze oefeningen, uitgezonderd de laatste, vind je in de webcursus.

Tracht vanaf nu bij alle volgende oefeningen zoveel mogelijk de invoer van de gegevens te valideren:

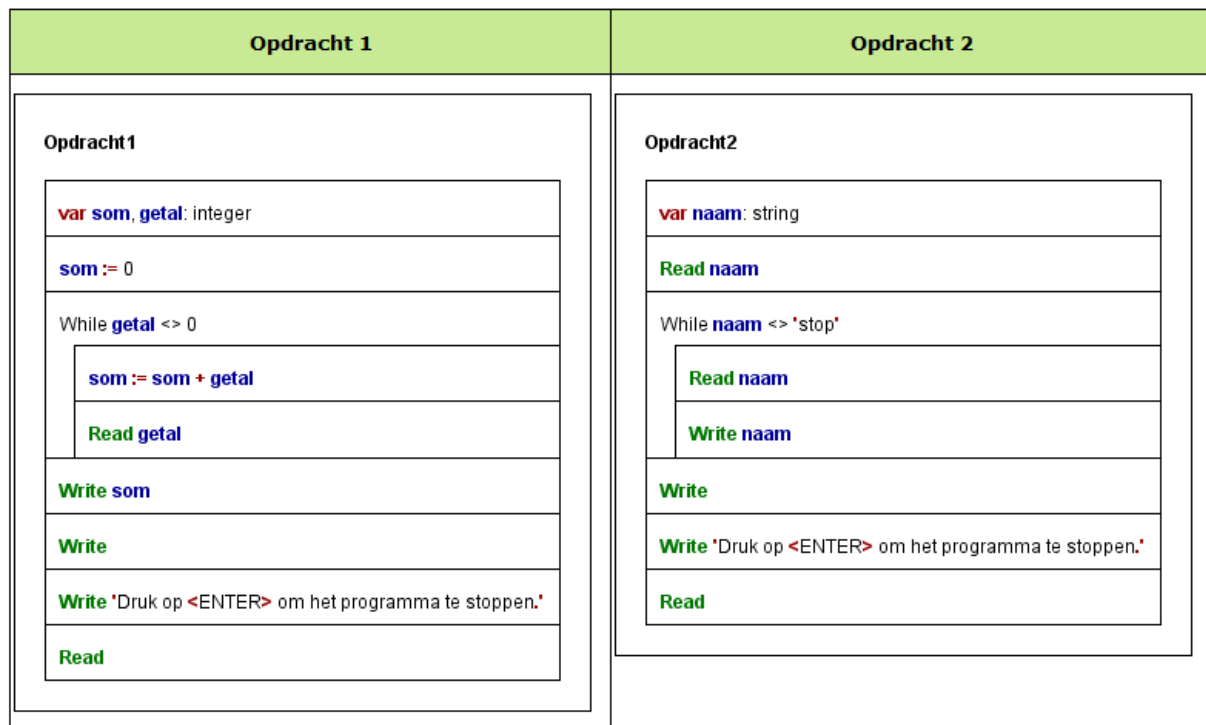
- controle op de invoer van een gegeven,
- opnieuw ingeven van dit gegeven tot de invoer juist is.

Op deze manier wordt er enkel met juiste gegevens gewerkt. Doch laat de essentie van de oefening niet verloren gaan door eindeloze validaties. Zoek een gulden middenweg!

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

5.6.1. Opgave 1: Waar loopt het fout?

Waarom zijn volgende structogrammen fout?



5.6.2. Opgave 2: Iteratie toevoegen

Bekijk onderstaand structogram. Breng een iteratie aan zodat het structogram korter geschreven kan worden.

Iteratie

var grondtal, product: integer
grondtal := 0
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
grondtal := grondtal + 1
product := grondtal * grondtal
Write grondtal, ' ', product
Write
Write "Druk op <ENTER> om het programma te stoppen."
Read

5.6.3. Opgave 3: Som van 10

Druk de getallen 1 t.e.m. 10 af. Toon daaronder de som van deze 10 getallen.

5.6.4. Opgave 4: Temperatuur

Het KMI ontvangt iedere dag temperatuurwaarnemingen uit het hele land. Deze gegevens worden per computer verwerkt. Er zijn niet elke dag evenveel waarnemingen.

Lees een willekeurig aantal temperaturen in (wanneer 99 wordt ingetikt, stopt de ingave). Druk hierna volgende resultaten af:

- aantal positieve temperaturen,
- aantal negatieve temperaturen,
- gemiddelde van de positieve temperaturen,
- gemiddelde van de negatieve temperaturen,
- aantal nul-temperaturen.

5.6.5. Opgave 5: Examens2

We hernemen opnieuw de opgave 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 meer dan 12/20.

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

5.6.6. Opgave 6: Getal raden

Maak een programma waarbij iemand naar een vooraf gegenereerd getal (tussen 1 en 100) moet raden.

De computer zegt of te hoog of te laag geraden is.

Na het juist raden van het getal, toon je hoeveel beurten de speler nodig had om het getal te raden.

5.6.7. Opgave 7: Min en max

Lees 10 getallen in. Positieve én negatieve getallen zijn toegelaten.

Druk aan het eind het grootste en het kleinste getal af.

5.6.8. Opgave 8: 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 afdrukt.

5.6.9. Opgave 9: Fibonacci

De “rij van Fibonacci” is een wiskundige rij, waarbij elk element de som van de twee voorgaande is. De eerste twee elementen zijn 0 en 1. Het begin van de rij is dus:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Maak een programma dat deze rij afdrukt voor alle getallen kleiner dan 1000.

5.6.10. Opgave 10: Tafel van vermenigvuldiging

Lees een getal in en druk de tafel van vermenigvuldiging van dat getal af.

5.6.11. Opgave 11: Sparen

Lees een beginkapitaal, een intrestpercentage en een looptijd in. Druk voor elk jaar tot het einde van de looptijd af welk bedrag tot dan toe gekapitaliseerd is (beginkapitaal + intrest).

5.6.12. Opdracht voor je coach: Lidgeld

Een vereniging vraagt € 10,00 lidgeld per jaar. Leden ouder dan 50 jaar krijgen € 2,00 reductie. Per kind ten laste wordt € 1,00 reductie gegeven (met een maximum van € 5,00). Indien het jaarinkomen onder € 12 500,00 ligt, wordt € 2,50 korting gegeven. De maximale reductie per lid is € 8,50.

Vraag:

Maak een programma dat per lid de nodige gegevens inleest en het te betalen lidgeld toont. Het programma eindigt wanneer de naam ‘stop’ of 'STOP' ingegeven wordt.

Laat op het einde ook weten voor hoeveel leden je het lidgeld hebt berekend en geef ook het totaal en het gemiddelde van de lidgelden.

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

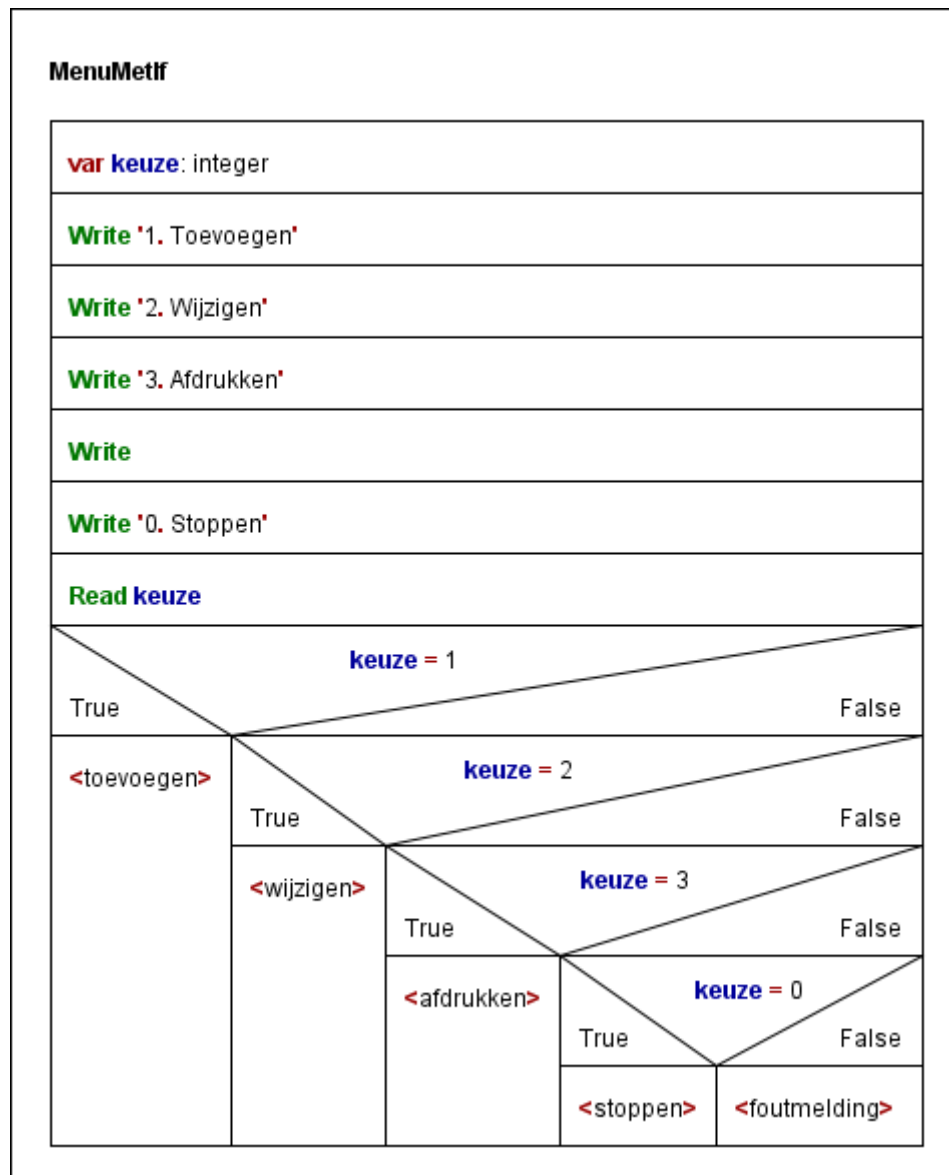
5.7. Meervoudige selectie

5.7.1. Variant op selectie

Een variant op de selectie is de **meervoudige selectie**. In een meervoudige selectie wordt de inhoud van een variabele (of een uitdrukking) vergeleken met verschillende mogelijke waarden. Bij elke mogelijkheid is een sequentieblok voorzien. Slechts één sequentieblok wordt uitgevoerd, afhankelijk van de waarde van de selectievariabele of uitdrukking.

Als **voorwaarde** wordt dus geen logische test opgegeven, maar een **expressie** die een eindig aantal welbepaalde waarden kan aannemen.

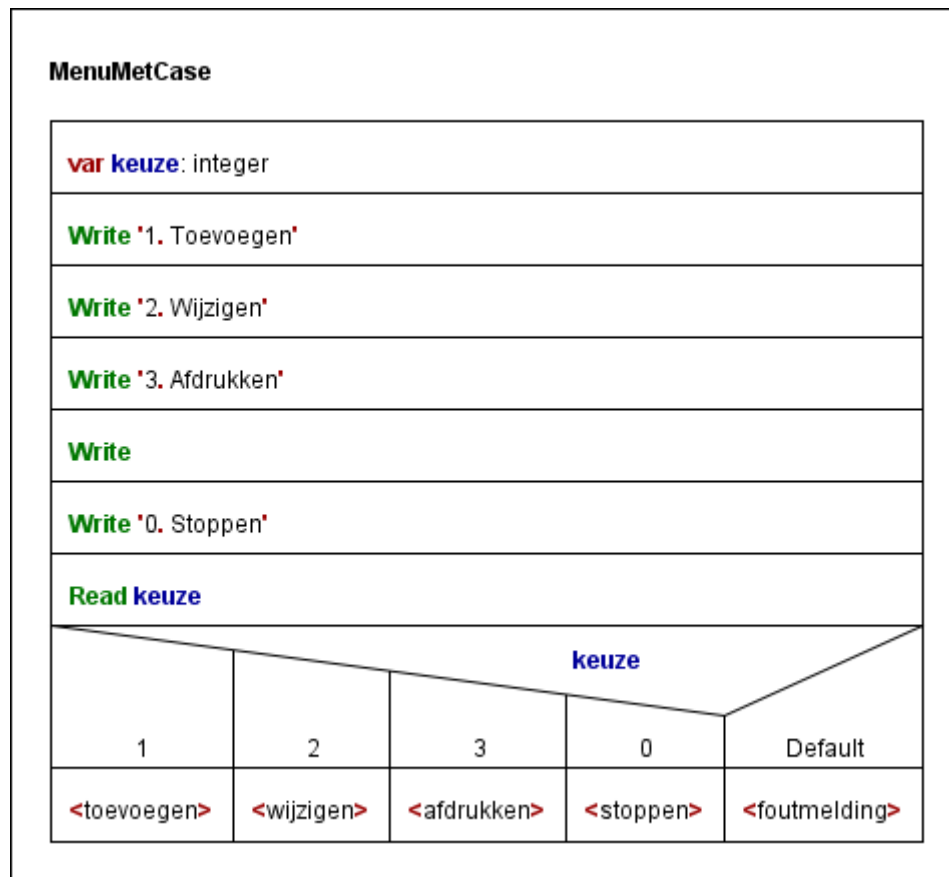
Een mooi voorbeeld is een menustructuur. Als je een menu van 4 items (“toevoegen”, “wijzigen”, “afdrukken”, “stoppen”) wil aanmaken met de tot hiertoe bekende structuren, dan zou je iets als volgt kunnen krijgen:



Voor elke menukeuze die je wil toevoegen, zal je een niveau dieper in de structuur moeten gaan.

Oplossing

Omdat we in dit voorbeeld met een eindig aantal welbepaalde keuzes te doen hebben (1,2,3,0 of de rest), kunnen we dit in een meervoudige selectie of **Case** onderbrengen.



De selectie wordt gemaakt op basis van de waarde van **keuze**. Voor elke aangegeven waarde wordt de opgegeven set instructies (kolom in het PSD) uitgevoerd. Een bijzonder geval is de kolom DEFAULT, te vergelijken met de ELSE uit de gewone selectiestructuur IF...THEN...ELSE....

Ook bij de meervoudige selectie is het **DEFAULT-onderdeel** facultatief, maar hou er rekening mee dat het weglaten ervan een foutsituatie kan opleveren.

(bijv. wanneer in bovenstaand voorbeeld "4" zou worden ingetikt. Dit vermijdt je wanneer de invoer van de keuze gevalideerd wordt en er dus enkel een waarde 0-3 toegelaten wordt. Dan kan het default-onderdeel zonder problemen weggelaten worden).

5.7.2. Meerdere waarden

Het is mogelijk dat bij een aantal verschillende waarden van de keuze-variabele dezelfde opdrachten moeten worden uitgevoerd. Deze verschillende waarden kunnen dan opgesomd worden (gescheiden door komma's), of als een bereik worden opgegeven (met een streepje ertussen).

MenuMetMeerdereWaarden

var keuze: integer			
Read keuze			
keuze			
1	2, 5	6 .. 8	Default
Write 'Een'	Write 'Twee of vijf'	Write 'Zes, zeven of acht'	Write 'alle andere waarden'

Een ander voorbeeld van een opsomming van mogelijke waardes is een keuze op basis van letters, waarbij kleine en grote letters gelijk behandeld moeten worden.

MenuMetMeerdereKarakters

var keuze: char				
Write 'T. toevoegen'				
Write 'W. wijzigen'				
Write 'A. afdrukken'				
Write				
Write 'X. stoppen'				
Read keuze				
keuze				
'T', 't'	'W', 'w'	'A', 'a'	'X', 'x'	Default
Write 'toevoegen'	Write 'wijzigen'	Write 'afdrukken'	Write 'stoppen'	Write 'fout'

De selectie wordt gemaakt op basis van de waarde van **keuze**. Voor elke aangegeven waarde wordt de opgegeven set instructies (kolom in het PSD) uitgevoerd. Een bijzonder geval is de kolom DEFAULT, te vergelijken met de ELSE uit de gewone selectiestructuur IF...THEN...ELSE....

Ook bij de meervoudige selectie is het **default-onderdeel** facultatief, maar hou er rekening mee dat het weglaten ervan een foutsituatie kan opleveren.

(bijv. wanneer in bovenstaand voorbeeld "4" zou worden ingetikt. Dit vermijdt je wanneer de invoer van de keuze gevalideerd wordt en er dus enkel een waarde 0-3 toegelaten wordt. Dan kan het DEFAULT-onderdeel zonder problemen weggelaten worden).

5.8. Opgaven4

In dit onderdeel maak je 3 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!

5.8.1. Opgave 1: Omrekenen

Maak een programma dat toelaat een aantal getallen van de éne eenheid naar de andere eenheid om te rekenen: kilogram naar pond en omgekeerd, en cm naar inches en omgekeerd. Bij het begin wordt een menu getoond met de vier omrekenmogelijkheden, en een mogelijkheid om het programma te stoppen:

1. KILOGRAM naar POND
2. POND naar KILOGRAM
3. CENTIMETER naar INCH
4. INCH naar CENTIMETER
5. STOP

Eerst dient een getal voor de menu-keuze ingelezen te worden. Bij ingave van een cijfer van 1 t.e.m. 4 wordt vervolgens een basisgetal ingelezen (d.i. het aantal kg/pond/cm/inch dat omgerekend moet worden). Dit basisgetal wordt omgerekend en de uitkomst wordt getoond; vervolgens wordt opnieuw het menu getoond.

Wanneer in het menu optie 5 wordt gekozen, stopt het programma. Wanneer een andere (foute) waarde wordt gelezen, wordt een passende foutmelding getoond en wordt een nieuwe keuze gevraagd.

5.8.2. Opgave 2: Graad

Voor een aantal studenten wordt voor een aantal vakken de punten ingelezen. Deze punten staan telkens op 20. Het programma begint met het aantal vakken te vragen waarvoor punten moeten worden gelezen (bijv. 5 vakken). Vervolgens wordt telkens de naam van de student gevraagd en worden de punten voor het gegeven aantal vakken ingelezen. Is de naam "stop", dan eindigt het programma.

Nadat alle punten voor één student zijn gelezen, drukt het programma af welke graad de student behaald heeft (en hoeveel percent hij in totaal heeft):

$\geq 90\%$ én alle vakken $\geq 10/20$	grootste onderscheiding
$\geq 80\%$ én alle vakken $\geq 10/20$	grote onderscheiding
$\geq 70\%$ én alle vakken $\geq 10/20$	onderscheiding
$\geq 60\%$ én alle vakken $\geq 10/20$	voldoening
$< 60\%$ of minstens 1 vak $< 10/20$	niet geslaagd

5.8.3. Opdracht voor je coach: Rekenmachine

Schrijf een eenvoudige rekenmachine. Het programma vraagt afwisselend een getal en een bewerking. Toegelaten bewerkingen zijn +, -, * en /. Wanneer = als bewerking wordt ingetypt, wordt het resultaat getoond en stopt het programma.

Je hoeft geen rekening te houden met reken-prioriteiten (eerst * en /, dan pas + en -): je voert de bewerkingen gewoon uit zoals ze worden ingevoerd.

Een voorbeeld:

7 + 5 -> 12, vervolgens terug een bewerking ingeven, bijv. de -, daarna een getal,
 12 - 3 -> 9, vervolgens terug een bewerking ingeven, bijv. de *, daarna een getal,
 9 * 7 -> 63, vervolgens terug een bewerking ingeven, bijv. de +, daarna een getal,
 63 + 14 -> 77, vervolgens terug een bewerking ingeven, bijv. de /, daarna een getal,
 77 / 7 -> 11, dan ingave van een = -> resultaat wordt getoond
 = 11

Vandaar de opmerking dat je geen rekening dient te houden met de rekenprioriteiten, vermits je de bewerking niet in zijn geheel uitrekent. Wiskundig gezien heeft de vermenigvuldiging en de deling voorrang op de optelling en de aftrekking en zou het resultaat van dit voorbeeld anders zijn, nl:

$$\begin{aligned} &7 + 5 - 3 * 7 + 14 / 7 \\ &= 7 + 5 - 21 + 2 \\ &= -7 \end{aligned}$$

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

5.9. Gestructureerd programmeren

In dit hoofdstuk bekijken we wat gestructureerd programmeren is en waarom we dit zo belangrijk vinden.

5.9.1. Efficiënt werken

In een werksituatie ga je je taken altijd **zo efficiënt mogelijk** uitvoeren. Hierbij zijn twee zaken belangrijk:

- beperk je tot **zinnvolle taken**,
- **organiseer** je werk.

Stel je volgende situatie eens voor:

Je collega haalt een verslag uit het klassement. Hij neemt een kopie van dit verslag. De kopie steekt hij terug in het klassement. Het origineel gooit hij weg.

Deze stappen zijn volledig overbodig en je zal je ook afvragen waar hij mee bezig is. Het is tijdverlies en verspilling van materiaal.

Een ander voorbeeld:

Je hebt een klassement, gesorteerd per klant. Als de klant contact met je opneemt, wil je meteen de juiste prijzen bij de hand hebben. Je hebt ook de gewoonte het dossier van de klant er op dat moment bij te nemen.

Toch is het niet interessant om de prijslijst bij te houden in het dossier van de klant:

- Je zou voor elke klant een kopie moeten maken van dezelfde lijst.
- Bij elke wijziging van de prijzen moet je per klant een kopie maken en elk klantendossier aanpassen.
- Doe je dit niet, dan krijgen bepaalde klanten een verkeerde prijs.

Je merkt hier dat een **verkeerde organisatie** van je werk voor veel **overbodig werk** zorgt en **fouten veroorzaakt**.

Als je deze voorbeelden leest, lijkt dit allemaal voor de hand te liggen.

We zoeken altijd manieren om ons werk zo **gestructureerd** en **efficiënt** mogelijk aan te pakken.

5.9.2. Efficiënt programmeren

Bij het programmeren moet je hier ook rekening mee houden. Je wil niet dat de computer zinloze taken uitvoert en fouten genereert. Je wil dat de computer **zo efficiënt mogelijk** functioneert.

Toch zie je dit sneller over het hoofd dan je denkt.

Voorbeeld:

Je voorziet in je programma 100 variabelen. Tijdens het programma lees je getallen in tot de gebruiker 'stop' ingeeft. Je leest bijvoorbeeld 6 getallen in. Daarna ga je eenzelfde verwerking uitvoeren op al de ingelezen getallen.

In je programma kan een iteratie zitten die 100 keer doorlopen wordt (het aantal variabelen dat voorzien is) of 6 keer (het aantal getallen dat werkelijk ingelezen werd). Dit maakt een groot verschil.

5.9.3. Onderhoud

De meeste programma's zullen ooit aangepast moeten worden. De grootste kost hierin is de menselijke kost. Hoeveel werk en tijd vraagt het om de aanpassing door te voeren? Als je bij die aanpassingen ook nog eens fouten gaat maken, loopt de rekening helemaal hoog op.

Om dit onderhoud te vergemakkelijken - en dus de kosten te beperken - is het belangrijk dat:

- een aanpassing maar **op 1 plaats** hoeft te gebeuren.
Denk aan de prijslijst die in elk klantendossier bijgehouden werd. Dit zorgt voor extra werk en problemen.
- het programma **duidelijk en overzichtelijk** is.
Het moet na geruime tijd nog leesbaar zijn. Je moet op een snelle en makkelijke manier de juiste code terug kunnen vinden.

5.10. Opgaven5

In dit onderdeel maak je 2 oefeningen. De oplossingen van deze oefeningen vind je in de webcursus.

5.10.1. Opgave 1: Fibonacci structureren

We hernemen de oefening rond de rij van Fibonacci.

We geven je hier een mogelijke oplossing, maar deze is niet erg gestructureerd. Waarom niet? Welke dingen zou jij anders doen en waarom?

Nog even ter herhaling de opgave:

De “rij van Fibonacci” is een wiskundige rij, waarbij elk element de som van de twee voorgaande is.

De eerste twee elementen zijn 0 en 1. Het begin van de rij is dus:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Maak een programma dat deze rij afdrukt voor alle getallen kleiner dan 1000.

RijVanFibonacci

```
var eerste, tweede, derde: integer
```

```
eerste := 0
```

```
Write eerste
```

```
tweede := 1 + eerste
```

```
Write ', ', tweede
```

```
derde := eerste + tweede
```

```
Write ', ', derde
```

```
While (eerste < 1000) AND (tweede < 1000) AND (derde < 1000)
```

```
    eerste := tweede + derde
```

```
    eerste < 1000
```

```
    True
```

```
    False
```

```
    Write ', ', eerste
```

```
    tweede := derde + eerste
```

```
    tweede < 1000
```

```
    True
```

```
    False
```

```
    Write ', ', tweede
```

```
    derde := eerste + tweede
```

```
    derde < 1000
```

```
    True
```

```
    False
```

```
    Write ', ', derde
```

```
Write
```

```
Write 'Typ <ENTER> om het programma te stoppen.'
```

```
Read
```

5.10.2. Opgave 2: Steen - Papier - Schaar

Steen, papier, schaar, ook gekend als *Blad, steen, schaar* of *Schaar, steen, papier*, is een wijdverbreid nulsomspel dat met de hand gespeeld wordt.

Twee spelers tellen af en steken tegelijk zonder aarzeling de hand uit in de vorm van een vuist (=steen), een vlakke hand (=papier) of twee gespreide vingers (=schaar). Volgende regels bepalen wie wint:

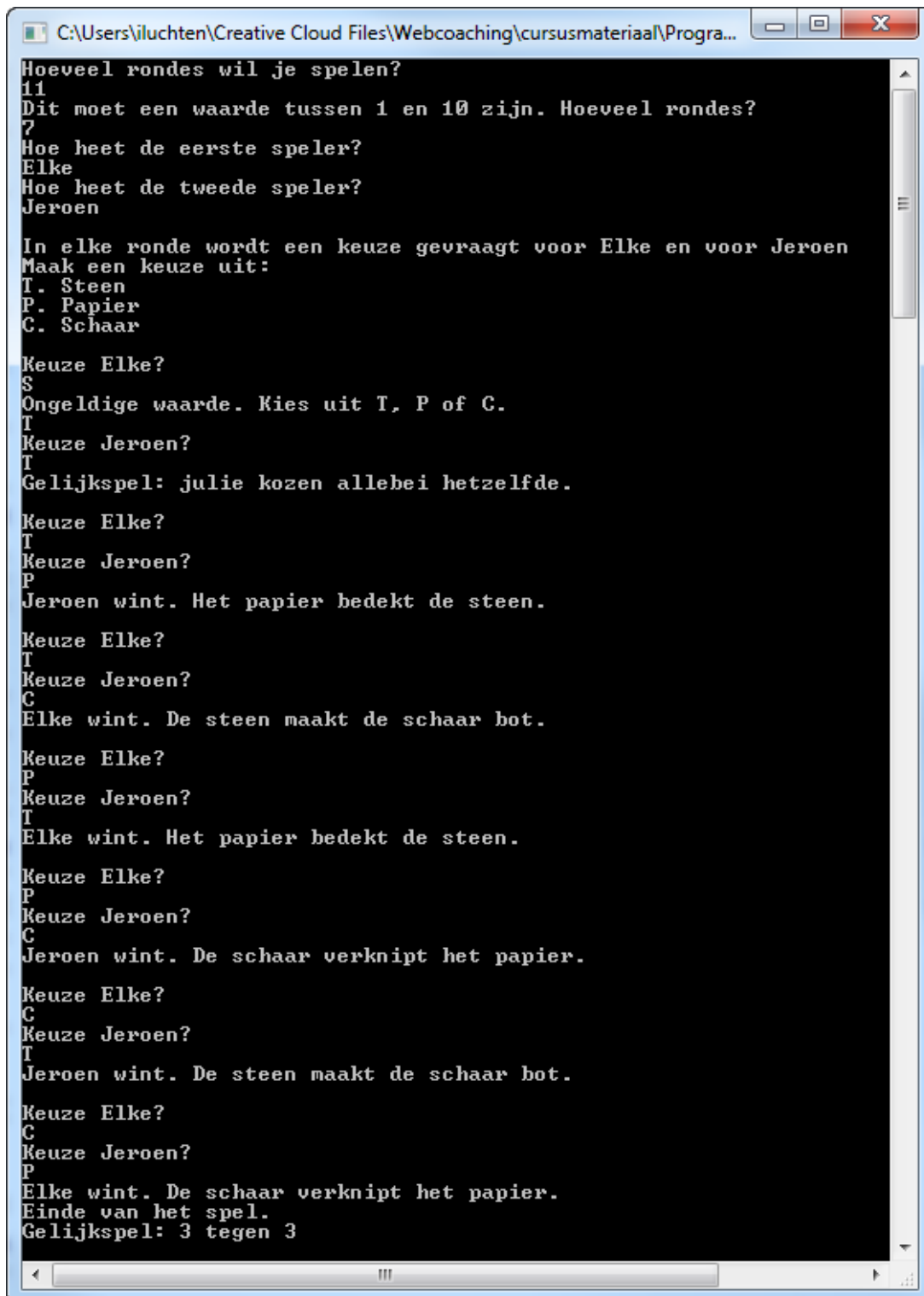
- De steen maakt de schaar bot.
- Het papier bedekt de steen.
- De schaar verknipt het papier.

Opgave:

Een spelletje bestaat uit een aantal opeenvolgende rondes. Vraag hoeveel rondes er gespeeld zullen worden en welke twee spelers meedoen. Per ronde wordt voor beide spelers een keuze gemaakt. Bepaal adhv bovenstaande regels wie de ronde heeft gewonnen. Laat weten wie de rond wint en waarom. Hou de score bij en laat op het einde van het spel weten wie heeft gewonnen en wat de scores zijn.

Bouw je programma zo efficiënt en gestructureerd mogelijk op.

Mogelijke uitvoer:



```
C:\Users\iluchten\Creative Cloud Files\Webcoaching\cursusmateriaal\Progra...
Hoeveel rondes wil je spelen?
11
Dit moet een waarde tussen 1 en 10 zijn. Hoeveel rondes?
7
Hoe heet de eerste speler?
Elke
Hoe heet de tweede speler?
Jeroen

In elke ronde wordt een keuze gevraagd voor Elke en voor Jeroen
Maak een keuze uit:
T. Steen
P. Papier
C. Schaar

Keuze Elke?
S
Ongeldige waarde. Kies uit T, P of C.
T
Keuze Jeroen?
T
Gelijkspel: jullie kozen allebei hetzelfde.

Keuze Elke?
T
Keuze Jeroen?
P
Jeroen wint. Het papier bedekt de steen.

Keuze Elke?
T
Keuze Jeroen?
C
Elke wint. De steen maakt de schaar bot.

Keuze Elke?
P
Keuze Jeroen?
T
Elke wint. Het papier bedekt de steen.

Keuze Elke?
P
Keuze Jeroen?
C
Jeroen wint. De schaar verknijpt het papier.

Keuze Elke?
C
Keuze Jeroen?
T
Jeroen wint. De steen maakt de schaar bot.

Keuze Elke?
C
Keuze Jeroen?
P
Elke wint. De schaar verknijpt het papier.
Einde van het spel.
Gelijkspel: 3 tegen 3
```

Hoofdstuk 6. Einde cursus

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

6.1. 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 basisstructuren**".

6.2. Wat nu?

Je hebt nu het eerste deel van programmatielogica doorgenomen.

Na dit deel kan je kiezen uit een van volgende onderdelen:

- **Procedures en functies**: procedures en functies maken en gebruiken
- **Arrays en Strings**: werken met arrays

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** door.