Verslag Eindopdracht Vertalerbouw - Chronos
Datum: 8-7-2013

M. J. Roo – s1081217, Jacob Obrechtstraat 1, Enschede
H. Slatman – s1006274, Witbreuksweg 393 306, Enschede
Studentassistent: Ronald Meijer

Inhoudsopgave

Inleiding 3

Beschrijving programmeertaal 4

Problemen en oplossingen 5

Syntax 6

Contextbeperkingen 8

Semantiek 11

Vertaalregels - TAM 14

Vertaalregels - JAM 17

Beschrijving Java-programmatuur 20

Testplan en –verslagen 23

Specificatie verantwoordelijkheden 26

Conclusies 27

Appendix 28

- Specificatie ChronosLexer & ChronosParser 28
- Specificatie ChronosChecker 35
- Specificatie Code Generator – ChronosTAMGenerator 39
- Specificatie Code Generator – ChronosJAMAdministrator 46
- Specificatie Code Generator – ChronosJAMGenerator 50
- In- en uitvoer van testprogramma 57

Inleiding

Dit verslag beschrijft de eindopdracht van het vak Vertalerbouw (192110352), Universiteit Twente, 2012-2013. Het doel van de eindopdracht is om een programmeertaal met bijbehorende compiler te ontwikkelen. Voorkennis om deze opdracht tot een goed einde te brengen werd opgedaan bij de hoor- en werkcolleges die aan het begin van het kwartiel plaatsvonden.

De te ontwikkelen programmeertaal en bijbehorende vertaler dienden aan enkele eisen te voldoen. Zo diende er gebruik gemaakt te worden van ANTLR, een raamwerk voor het programmeren van vertalers. ANTLR neemt de programmeur veel werk uit handen, daar veel code automatisch gegenereerd wordt. Daarnaast was er een uitgebreide specificatie beschikbaar van eisen waaraan de programmeertaal syntactisch, semantisch en functioneel moest voldoen.

De ontwikkelde programmeertaal werd Chronos gedoopt; naar de figuur uit de Griekse mythologie die de personificatie van de *Tijd* is. Dit verslag bevat een korte beschrijving van de taal, syntax, contextbeperkingen en semantiek van de taal, en de vertaalregels voor de programmeertaal.
De ontwikkelde vertaler kan een Chronos programma niet enkel vertalen naar TAM code, maar ook naar Java assembly code (Jasmin).

Naast een beschrijving van de ontwikkelde taal en vertaler, zal er ook beschreven worden welke programmatuur zelf ontwikkeld is om de vertaler te realiseren. Ook komt het testen van de vertaler uitvoerig aan bod aan het eind van het verslag. Achterin bevinden zich de appendices, waar de specificaties van de verschillende elementen van de vertaler gevonden kunnen worden, alsook een uitgebreid testprogramma.

Beschrijving programmeertaal

Chronos is een programmeertaal die zich qua syntax redelijk verhoudt tot de voorbeelden die in de practicumhandleiding werden gegeven (zie practicumhandleiding, A4: Taalelementen in de eindopdracht). Naast de basiselementen die daar beschreven worden, ondersteunt Chronos ook conditionele statements en biedt het een lusinstructie in de vorm van een *while* statement. Ook deze toegevoegde elementen lijken qua syntax op de taal die beschreven wordt in de practicumhandleiding. In Chronos dienen declaraties en expressies altijd met een puntkomma afgesloten te worden.

Programma's in Chronos starten altijd met een aantal declaraties of expressies. Een programma dient altijd te eindigen met een expressie; een programma dat enkel uit een declaratie bestaat zal dus een foutmelding opleveren bij het compileren ervan. Declaraties van variabelen dienen altijd expliciet het type te definiëren; bij een constante wordt het type ervan afgeleid van de toegekende waarde.

Het is in Chronos mogelijk om een constante te declareren die zijn waarde berekent aan de hand van één of meerdere variabelen. De constante wordt dus tijdens het uitvoeren van het programma gedeclareerd. Hiermee volgt Chronos het voorbeeld van Java, waarin het mogelijk is om bijvoorbeeld in een while-loop een *final static int* te definiëren die afhankelijk is van een variabele die telkens opgehoogd wordt. Uiteraard kan een constante die eenmaal binnen een bepaalde scope gedeclareerd is, niet nogmaals een waarde toegekend krijgen; dit levert een foutmelding tijdens het compileren op.

Scopes kunnen in Chronos worden herkend aan de accolades. Alles wat binnen een set van accolades gedeclareerd en geëvalueerd wordt, is enkel binnen de accolades, de scope, beschikbaar. Binnen het conditionele statement en de lusinstructie hoeven er niet per se accolades rond de nieuwe scope geplaatst te worden; hier herkent de compiler zelf dat er een nieuwe scope geopend is. Het gebruiken van variabelen en constanten die binnen een conditioneel statement of lusinstructie zijn gedeclareerd zal dan ook tot een foutmelding leiden.

Het type en de waarde van de returnwaarde hangt in Chronos af van de laatste expressie die binnen een scope geëvalueerd wordt. Het volgt hiermee de richtlijn die in de practicumhandleiding gegeven wordt; er is geen expliciet return statement aanwezig.

Problemen en oplossingen

Het ontwikkelen van de programmeertaal Chronos, en de daarbij behorende vertaler, was een aangelegenheid die een behoorlijke hoeveelheid tijd in beslag nam. Niet op de minste plaats kwam dit door problemen die zich voordeden tijdens de ontwikkeling van de programmeertaal, waardoor er extra tijd geïnvesteerd moest worden om deze problemen op te lossen. Deze sectie beschrijft enkele van de problemen die tijdens de ontwikkeling naar voren kwamen.

Ten eerste ondersteunt Chronos, zoals in de vorige sectie beschreven, het declareren van constanten die afhankelijk zijn van een variabele. Hiermee wordt het voorbeeld van Java gevolgd, zoals eerder beschreven. Om dit soort constanten te kunnen ondersteunen, dienen de constanten tijdens het draaien van het programma pas gedefinieerd te worden, en kunnen deze dus niet tijdens het compileren ingevuld worden vanuit een andere datastructuur, zoals een *symboltable*. De constanten in Chronos gedragen zich dus eigenlijk als variabelen. Om ervoor te zorgen dat een constante slechts eenmaal een waarde toegekend kan krijgen, is dit afgedwongen in de checker.

Bij het uitvoeren van testprogramma's waarin meerdere geneste conditionele statements in elkaar verweven werden, bleek dat de uitvoer in veel gevallen niet klopte. Dit kwam omdat de labels niet correct gegenereerd werden: sommige labels kregen de verkeerde namen, en sommige andere werden helemaal niet geprint. De oorzaak hiervan was dat de structuur voor de conditionele statements grotendeels geïnspireerd was door de implementatie van conditionele statements in Calc (practicum week 3). Het bleek echter nodig om dit zeer sterk aan te passen om geneste conditionele statements te kunnen ondersteunen. Er werd een tweedimensionale matrix-structuur ontwikkeld waarmee deze labels op correcte wijze berekend en gegenereerd werden.

Bij het implementeren van de Java assembly (Jasmin) code generatie, wilde het inlezen van integers niet goed werken. Wanneer een integer ingelezen werd, werd enkel de eerste digit van deze integer gelezen en opgeslagen. Dit zorgde bij de eerste testruns voor wat kopzorgen. Uiteindelijk werd er een oplossing gevonden op http://www.ceng.metu.edu.tr/courses/ceng444/link/f3jasmintutorial.html, waarmee dit probleem opgelost leek. Echter, na een behoorlijke tijd van debugging, bleek dat er nog een onderdeel miste op de bovengenoemde URL, namelijk een *carriage return*. Deze werd toegevoegd aan de methode. Daarmee lijkt het inlezen van integers in Jasmin nu correct te werken. Wel is het nodig, dat wanneer er in een programma meer dan één integer ingelezen moet worden, deze bij de eerste aanroep van de methode ingegeven moeten worden, gescheiden door spaties. Als dit niet gedaan wordt, zal de verwachte uitkomst van het draaien van het programma niet kloppen met de daadwerkelijke uitkomst.

Een ander punt bij de ontwikkeling van de Jasmin code generator is dat het nodig bleek om de aantallen lokale variabelen en de maximale stackhoogte van functies te bepalen, aangezien een te lage waarde voor één van deze twee voor foutmeldingen zorgt bij uitvoer van het Java programma. Er werd een extra *tree grammar* ontwikkeld die enkel voor administratie bij Jasmin gebruikt wordt. Deze houdt onder andere bij hoeveel constanten en variabelen er in een Chronos programma gedeclareerd worden. Deze waarden kunnen daarna opgevraagd worden en doorgegeven worden aan de generator van de Jasmin code.

Syntax

Hieronder is de syntax van Chronos weergegeven. Er is voor gekozen om dit aan de hand van een listing te doen die veel lijkt op de listing van *Chronos.g* zoals die in Appendix: *Specificatie ChronosLexer & ChronosParser* te vinden is. Na de listing volgt een korte samenvatting van de syntax van Chronos.

```
Program                ::=    DeclStatBlocks EOF
DeclStatBlocks         ::=    ( ( Declaration ; )* Expression ; )+
IndentDeclStatBlocks   ::=    { DeclStatBlocks }
Declaration            ::=    ConstDeclaration
                       |      VarDeclaration
ConstDeclaration       ::=    const Identifier := Expression
VarDeclaration         ::=    var Identifier : Type
Type                   ::=    Integer
                       |      Char
                       |      Boolean
Expression             ::=    AssignStatement
                       |      WhileStatement
AssignStatement        ::=    ExpressionOr ( := ExpressionOr )?
ExpressionOr           ::=    ExpressionAnd ( || ExpressionAnd)*
ExpressionAnd          ::=    ExpressionRelational ( && ExpressionRelational )*

ExpressionRelational   ::=    ExpressionPlusMinus  (  (  >  |  >=  |  <  |  <=  |  ==  |  !=  )
ExpressionPlusMinus )*

ExpressionPlusMinus    ::=    ExpressionMultDiv ( ( + | - ) ExpressionMultDiv )*
ExpressionMultDiv      ::=    ExpressionUnary ( ( % | * | / ) ExpressionUnary )*
ExpressionUnary        ::=    Operand
                       |      ! Operand
                       |      + Operand
                       |      - Operand
Operand                ::=    true
                       |      false
                       |      Number
                       |      Character
                       |      Identifier
                       |      ( Operand )
                       |      IndentDeclStatBlocks
                       |      Read
                       |      Print
                       |      ExpressionIf
Read                   ::=    read ( VarList )
Print                  ::=    print ( ExpressionList )
ExpressionList         ::=    Expression ( , Expression )*
VarList                ::=    Identifier ( , Identifier )*
ExpressionIf           ::=    if DeclStatBlocks ExpressionThen fi
ExpressionThen         ::=    then DeclStatBlocks ( ExpressionElse )?
ExpressionElse         ::=    else DeclStatBlocks
WhileStatement         ::=    while DeclStatBlocks ExpressionDo od
ExpressionDo           ::=    do DeclStatBlocks
Number                 ::=    Digit+
Character              ::=    ' SingleChar '
SingleChar             ::=    Digit | Letter | Symbol
Identifier             ::=    Letter ( Letter | Digit )*
Digit                  ::=    '0'.. '9'
Symbol                 ::=    ' ' | '-'
Letter                 ::=    ( Lower | Upper )
Lower                  ::=    'a' .. 'z'
Upper                  ::=    'A' .. 'Z'
```

Chronos programma's bestaan altijd uit enkele declaraties en expressies. Een programma zal altijd eindigen in een expressie. Als dit niet het geval is, levert dit een foutmelding op. Declaraties en expressies worden altijd afgesloten met een puntkomma. Een *closed compound statement* wordt omvat door accolades. Een *closed compound statement* is een expressie, en dient dus ook afgesloten te worden met een puntkomma.

Er zijn twee verschillende declaraties, die voor variabelen en die voor constanten. Constanten worden gedeclareerd met het keyword *const*, gevolgd door een Identifier, gevolgd door *:=* (Becomes) gevolgd door de waarde en een puntkomma. Een variabele wordt gedeclareerd met het keyword *var*, dan een Identifier, een dubbele punt en daarachter het type. Dit type kan *int*, *char* of *bool* zijn.

Expressies kunnen een *AssignStatement* of een *WhileStatement* zijn. Een *AssignStatement* bestaat uit één of meerdere operands, gescheiden door := (Becomes). Hiermee is het dus mogelijk om multiple assignment toe te passen. Overige expressies worden gescheiden door de verschillende operaties (||, &&, >, >=, <, <=, ==, !=, +, -, %, *, / en !). Expressies zullen uiteindelijk reduceren tot een Operand.

Operands kunnen verschillende zaken zijn. *true* en *false* zijn twee van de mogelijkheden. Deze worden toegekend aan *bool*s. *Number*, *Character* en *Identifier* zijn drie andere mogelijkheden. Ook een andere Operand tussen normale haakjes is één van de mogelijkheden. De laatste mogelijkheden zijn de read, print, *ExpressionIf* en *closed compound statement*.

Het read statement start met keyword *read*, waarna een lijst van variabelen volgt, tussen normale haakjes. De afzonderlijke variabelen binnen de haakjes zijn gescheiden door komma's. Het print statement start met keyword *print* waarna een lijst van expressies volgt, tussen normale haakjes. De verschillende expressies tussen de haakjes zijn gescheiden door komma's.

De *ExpressionIf* start met het keyword *if*. Daarna volgt een reeks van declaraties en expressies (minstens één en eindigend in een expressie) en een *ExpressionThen*. *ExpressionThen* start met het keyword *then*. Na *then* volgt een reeks van declaraties en expressies (minstens één en eindigend in een expressie), waarop de optionele *ExpressionElse* volgt. *ExpressionElse* begint met het keyword *else*, waarop wederom een reeks van declaraties en expressies volgt. De *ExpressionIf* wordt afgesloten met het keyword *fi*. Daar *ExpressionIf* een expressie is, wordt ook deze afgesloten met een puntkomma.

Het *WhileStatement* start met het keyword *while*. Hierop volgt een reeks van declaraties en expressies (minstens één, en eindigend in een expressie). Daarop volgt de *ExpressionDo*, die start met het keyword *do*. Hierop volgt weer een reeks van declaraties en expressies (minstens één en eindigend in een expressie). Het *WhileStatement* wordt afgesloten met het keyword *od* waar bovendien een puntkomma op dient te volgen.

Een *Character* bestaat uit een enkel karakter waaromheen enkele aanhalingstekens geplaatst zijn. *Numbers* bestaan uit minstens een enkele Digit.

Contextbeperkingen

Naast de syntax van de programmeertaal, dient een programma zich ook aan de contextbeperkingen van de programmeertaal te houden. De ChronosChecker draagt er zorg voor dat een programma correct gecontroleerd wordt op deze contextbeperkingen. De contextbeperkingen van Chronos zijn hieronder per syntax regel uiteengezet. Bij de beschrijvingen van resultaat types gaan we ervan uit dat het programma zich houdt aan de contextuele beperkingen. Als dit niet het geval is zal de ChronosChecker een foutmelding genereren. Enkele grammaticaregels zijn hieronder ten opzichte van de syntax gesplitst om de contextuele beperkingen beter te kunnen beschrijven. Dit is bijvoorbeeld het geval bij *ExpressionRelational*.

```
DeclStatBlocks        ::=    ( ( Declaration ; )* Expression ; )+
```

Variabelen en constanten die gebruikt worden (*applied occurrence*) dienen daarvoor eerst gedeclareerd te zijn (*binding occurrence*) binnen de juiste scope. Het type van het *DeclStatBlock* komt overeen met het type van de laatste *Expression* binnen de *DeclStatBlocks*.

```
IndentDeclStatBlocks  ::=    { DeclStatBlocks }
```

Variabelen en constanten die gebruikt worden binnen een *closed compound statement* dienen uiteraard eerst gedeclareerd te zijn. Dit kan binnen hetzelfde *closed compound statement* zijn, maar ook in een eerdere scope. Buiten de *closed compoud statement* zijn de daarbinnen gedeclareerde variabelen en constanten niet meer beschikbaar. Het type van de *closed compound statement* is het type van de laatste expressie binnen het *DeclStatBlock*.

```
ConstDeclaration      ::=    const Identifier := Expression
```

Het type van een constante wordt dynamisch bepaald. Het type hangt af van de Expression (*compound statement*). De Expression wordt geëvalueerd, en aan de hand daarvan wordt een type toegekend aan de constante. De constante mag daarnaast slechts één keer een waarde toegekend krijgen. Een Identifier dient uniek te zijn. Declaratie van een constante (*binding occurrence*) dient altijd voor de *applied occurrence* plaats te vinden. Het resultaat type is *no_type*.

```
VarDeclaration        ::=    var Identifier : Type
```

Een variabele declaratie dient altijd met een geldig type (*int*, *bool*, *char*) gedeclareerd te worden. De Identifier dient niet eerder gebruikt te zijn. Een declaratie van een variabele (*binding occurrence*) dient altijd voor de *applied occurrence* plaats te vinden. Het resultaat type is *no_type*.

```
AssignStatement       ::=    ExpressionOr ( := ExpressionOr )?
```

Aan de linkerzijde van het *Becomes* (:=) teken dient altijd een Identifier te staan die eerder gedeclareerd is binnen de huidige of minder diepe scope. Het type van de Identifier dient overeen te komen met het type van de ExpressionOr die aan de rechterkant van het *Becomes* teken staat.

```
ExpressionOr          ::=    ExpressionAnd ( || ExpressionAnd)*
```

De operanden aan beide zijden van een || dienen van type *bool* te zijn. Het resultaat type is type *bool*.

```
ExpressionAnd  ::=    ExpressionRelational ( && ExpressionRelational )*
```

De operanden aan beide zijden van de && dienen van van type *bool* te zijn. Het resultaat type is type *bool*.

```
ExpressionRelational  ::=  ExpressionPlusMinus ( ( > | >= | < | <= ) ExpressionPlusMinus )*
```
De operanden aan beide zijden van één van de relationele operatoren dienen van het type *int* te zijn. Het resultaattype zal *bool* zijn.

```
ExpressionRelational  ::=  ExpressionPlusMinus ( ( == | != ) ExpressionPlusMinus )*
```
De operanden aan beide zijden van één van de relationele operatoren dienen allebei hetzelfde type te hebben. Dit type kan niet *void* zijn. Het resultaattype zal *bool* zijn.

```
ExpressionPlusMinus   ::=  ExpressionMultDiv ( ( + | - ) ExpressionMultDiv )*
```
Het type van de operanden aan beide zijden van een + of – teken dienen allebei van het type *int* te zijn. Het resultaat type is *int*.

```
ExpressionMultDiv     ::=  ExpressionUnary ( ( % | * | / ) ExpressionUnary )*
```
Het type van de operanden aan beide zijden van een %, * of / teken dienen allebei van het type *int* te zijn. Het resultaat type is *int.*

```
ExpressionUnary       ::=  ! Operand
```
Het type van de Operand dient *bool* te zijn. Het resultaat is type *bool*

```
ExpressionUnary       ::=  + Operand
                        |  - Operand
```
Het type van de Operand dient *int* te zijn. Het resultaat is ook van type *int*

```
Read                  ::=  read ( VarList )
```
Het resultaat type van het read statement is gelijk aan het type van de variabele die ingelezen wordt. Als er meerdere variabelen ingelezen worden, is het resultaattype *void*.

```
Print                 ::=  print ( ExpressionList )
```
Het resultaat type van het print statement is gelijk aan het type van de variabele of expressie die geprint wordt. Een expressie met type *void* kan niet geprint worden. Als er meerdere expressies en/of variabelen geprint worden, is het resultaat type *void*.

```
ExpressionIf          ::=  if DeclStatBlocks ExpressionThen fi
```
Variabelen en constanten die in *DeclStatBlocks* gedeclareerd worden zijn enkel binnen het if-statement beschikbaar (dus ook in de opvolgende *ExpressionThen* en de optionele *ExpressionElse*). Het type van de laatste expressie van het *DeclStatBlock* dient van type *bool* te zijn. Het resultaat type is gelijk aan het type van de *ExpressionThen*.

```
ExpressionThen        ::=  then DeclStatBlocks ( ExpressionElse )?
```
Variabelen en constanten die in *DeclStatBlocks* gedeclareerd worden, zijn enkel binnen die scope (*DeclStatBlocks*) beschikbaar. Het resultaat type is het type van de *ExpressionElse*.

```
ExpressionElse        ::=  else DeclStatBlocks
```
Variabelen en constanten die in *DeclStatBlocks* gedeclareerd worden, zijn enkel binnen die scope beschikbaar. Als het type van de laatste expressie binnen de *DeclStatBlock*s gelijk is aan het type van de laatste expressie binnen de *DeclStatBlocks* van de voorgaande *ExpressionThen,* dan is het resultaat type het type van die expressie. Als de types niet gelijk zijn, is het resultaat type *void*. Daar *ExpressionElse* een optionele regel is, zal het resultaat type *void* zijn als deze niet aanwezig is.

```
WhileStatement        ::=    while DeclStatBlocks ExpressionDo od
```
Variabelen en constanten die binnen *DeclStatBlocks* worden gedeclareerd, zijn enkel binnen dit *WhileStatement* (dus ook in de bijbehorende *ExpressionDo*) beschikbaar. Het type van de laatste expressie in het *DeclStatBlocks* dien van type *bool* te zijn. Het resultaat type van een *WhileStatement* is altijd *void*.

```
ExpressionDo          ::=    do DeclStatBlocks
```
Variabelen en constanten die binnen *DeclStatBlocks* gedeclareerd worden, zijn enkel binnen de huidige scope beschikbaar.

```
Identifier            ::=    Letter ( Letter | Digit )*
```
Wanneer een Identifier gebruikt wordt (*applied occurrence*), dient deze reeds op de huidige of een voorgaande scope gedeclareerd te zijn (*binding occurrence*).

Semantiek

Programma's geschreven in Chronos bestaan, zoals eerder beschreven, uit een reeks van declaraties en expressies, waarbij het laatste elementen van deze reeks altijd een expressie is. Er zijn twee soorten declaraties mogelijk, namelijk de declaratie van een constante en die van een variabele. Daarna volgende expressies. Bij sommige van deze is een splitsing gemaakt om een duidelijkere beschrijving van de semantiek te kunnen geven.

```
DeclStatBlocks        ::=    ( ( Declaration ; )* Expression ; )+
```
De waarde die opgeleverd wordt is de waarde van de laatste expressie.

```
IndentDeclStatBlocks  ::=    { DeclStatBlocks }
```
De waarde die opgeleverd wordt is de waarde van de laatste expressie in de *DeclStatBlocks*.

```
ConstDeclaration      ::=    const Identifier := Expression
```
Bij de declaratie van een constante wordt er een plaats gereserveerd voor de *Identifier* in de *symboltable*. Daarbij wordt de uitkomst van de *Expression* verbonden met deze *Identifier*: de geëvalueerde waarde van de *Expression* wordt toegekend aan de zojuist gereserveerde plek in de *symboltable*. Het type van de *Identifier* wordt aan de hand van de *Expression* geëvalueerd en toegewezen. De constante declaratie levert geen waarde op.

```
VarDeclaration        ::=    var Identifier : Type
```
Bij de declaratie van een variabele wordt er een plaats gereserveerd voor de *Identifier* in de *symboltable*. De *Identifier* krijgt het type dat door *Type* aangegeven wordt. Deze *Type* kan *int*, *bool* of *char* zijn. Ook krijgt de variabele een initialisatiewaarde toegekend; namelijk 0 voor *Type int*, 0 voor *Type bool* en 'a' voor *Type char*. De declaratie van een variabele levert geen waarde op.

```
AssignStatement       ::=    ExpressionOr ( := ExpressionOr )?
```
Het *AssignStatement* zal een waarde toekennen aan een *Identifier*. De waarde die toegekend wordt, wordt bepaald door het evalueren van de *ExpressionOr* aan de rechterkant van het *Becomes* teken. De resultaatwaarde zal daarna voor gebruik door een andere *Expressie* beschikbaar blijven. De eerste *Exp*ression*Or* zal altijd een *Identifier* zijn, want dat is wat de ChronosChecker zal afdwingen.

```
ExpressionOr          ::=    ExpressionAnd ( || ExpressionAnd)*
```
De *ExpressionAnd* wordt geëvalueerd en zal een waarde opleveren. De optionele *ExpressionAnd* zal ook geëvalueerd worden, en de || operation zal daarna op het resultaat van de twee uitgevoerd worden. Het evalueren van dit laatste zal een waarde opleveren die aan *ExpressionOr* toegekend wordt. De toegekende waarde is altijd 0 of 1 (*false* of *true*)

```
ExpressionAnd         ::=    ExpressionRelational ( && ExpressionRelational )*
```
De *ExpressionRelational* wordt geëvalueerd en zal een waarde opleveren. De optionele *ExpressionRelational* zal ook geëvalueerd worden, en de && operation zal daarna op het resultaat van de twee uitgevoerd worden. Het evalueren van dit laatste zal een waarde opleveren die aan *ExpressionAnd* toegekend wordt. De toegekende waarde is altijd 0 of 1 (*false* of *true*)

```
ExpressionRelational  ::=    ExpressionPlusMinus  (  (  >  |  >=  |  <  |  <=  |  ==  |  !=  )
ExpressionPlusMinus )*
```

Beide zijden van de relationele operators zullen geëvalueerd worden. Daarna zullen de geëvalueerde waarden gebruikt worden om de uitkomst van de relationele operatie op deze twee waarden te evalueren. Het resultaat van deze evaluatie zal toegekend worden aan de *ExpressionRelational*. De toegekende waarde is altijd 0 of 1 (*false* of *true*)

```
ExpressionPlusMinus   ::=    ExpressionMultDiv ( ( + | - ) ExpressionMultDiv )*
```

De *ExpressionMultDiv*s aan beide zijden van de + of – operator zullen geëvalueerd worden. Daarna zal de + of – operatie geëvalueerd worden. De resulterende waarde zal toegekend worden aan de *ExpressionPlusMinus*. De toegekende waarde is altijd een *int*

```
ExpressionMultDiv     ::=    ExpressionUnary ( ( % | * | / ) ExpressionUnary )*
```

De *ExpressionUnary*s aan beide zijden van de %, * of / operator zullen geëvalueerde worden. Daarna zal de %, * of / operatie geëvalueerd worden, gebruikmaken van de twee zojuist geëvalueerde waarden. De resulterende waarde zal toegekend worden aan de *ExpressionMultDiv*. De toegekende waarde is altijd een *int*.

```
ExpressionUnary       ::=    Operand
                        |    ! Operand
                        |    + Operand
                        |    - Operand
```

De *Operand* zal geëvalueerd worden. Daarna zal de (optionele) unary operator op de geëvalueerde waarde geëvalueerd worden, waarna het resultaat wordt toegekend aan de *ExpressionUnary*. De not operator zal de *bool* inverteren, de unary plus voert niks uit en de unary min zal de top van de stack met -1 vermenigvuldigen.

```
Operand               ::=    true
                        |    false
                        |    Number
                        |    Character
                        |    Identifier
```

De rechterzijde zal geëvalueerd worden. De *Operand* zal de resultaatwaarde van deze evaluatie toegekend krijgen. Dit kan *true*, *false*, een getal, een karakter of een *Identifier* zijn.

```
Read                  ::=    read ( VarList )
```

VarList zal geëvalueerd worden. Als de *VarList* een enkele *Identifier* bevat, zal de resultaatwaarde de waarde van die *Identifier* zijn. Als er meer *Identifier*s in de *VarList* voorkomen, zal er geen resultaatwaarde opgeleverd worden.

```
Print                 ::=    print ( ExpressionList )
```

ExpressionList zal geëvalueerd worden. Als de *ExpressionList* meer dan één expressie of variabele bevat, zal er geen waarde opgeleverd worden. Is dit niet het geval, dan zal de waarde van de expressie opgeleverd worden

```
ExpressionIf          ::=    if DeclStatBlocks ExpressionThen fi
ExpressionThen        ::=    then DeclStatBlocks ( ExpressionElse )?
ExpressionElse        ::=    else DeclStatBlocks
```

De laatste expressie in de *DeclStatBlocks* na het keyword *if* zal geëvalueerd worden. Deze zal type *bool* zijn. Als de expressie naar *true* evalueert, zal de *DeclStatBlocks* na het keyword *then* geëvalueerd worden. Het resultaat daarvan zal toegekend worden aan de *ExpressionIf*. Als de

expressie naar *false* evalueert en er is een *ExpressionElse* aanwezig, dan zal de *DeclStatBlocks* na het *else* keyword geëvalueerd worden. Het resultaat daarvan zal dan toegekend worden aan *ExpressionIf*. Als er geen *ExpressionElse* aanwezig is, zal de resultaatwaarde type *void* hebben.

```
WhileStatement        ::=    while DeclStatBlocks ExpressionDo od
ExpressionDo          ::=    do DeclStatBlocks
```

De laatste expressie in de *DeclStatBlocks* zal geëvalueerd worden. Als deze naar *true* evalueert, zal het programma doorgaan naar de *DeclStatBlocks* van de *ExpressionDo*. Er zal geen resultaatwaarde opgeleverd worden aan het *WhileStatement*.

De overige regels voor onder andere *Number*, *Character* en *Identifier* zullen allen geëvalueerd worden en de geëvalueerde waarde zal toegekend worden aan de linkerkant van de regel voor gebruik verderop in het programma. Zo zal een *Number* geëvalueerd worden; dit levert een resultaatwaarde op, die ergens anders in het programma, bijvoorbeeld in een *AssignStatement* gebruikt kan worden om deze waarde toe te kennen aan een *Identifier*, die ook geëvalueerd is.

Vertaalregels - TAM

De onderstaande vertaalregels zijn gebaseerd op de grammatica van het syntax gedeelte. Sommige regels zijn hierbij samengenomen. Hierdoor kan er wellicht wat onduidelijkheid bestaan over de naamgeving. We verwijzen graag naar de diverse grammatica bestanden om dit te verduidelijken, mocht dat nodig zijn. Verder is er een opdeling gemaakt in de vertaalregels voor TAM en JAM om de verschillen, ook al zijn dit er niet veel, duidelijk te kunnen aangeven.

```
Program              ::=    DeclStatBlocks EOF
```
Run [*DeclStatBlocks*] =
    Elaborate *DeclStatBlocks*
    HALT

Elaborate [*DeclStatBlocks*] =
    Elaborate *Declaration* *één van de mogelijkheden van DeclStatBlocks*

Elaborate [*DeclStatBlocks*] =
    Elaborate *Expression* *de andere mogelijkheid van DeclStatBlocks*

```
ConstDeclaration     ::=    const Identifier := Expression
```
Elaborate [*ConstDeclaration*] =
    PUSH 1 *maakt ruimte vrij*
    Evaluate *Expression* *evalueert de expressie (kan variabelen bevatten)*
    Assign *Identifier* *wijst de geëvalueerde waarde toe aan Identifier*

```
VarDeclaration       ::=    var Identifier : Type
```
Elaborate [*VarDeclaration*] =
    PUSH 1 *maakt een ruimte vrij*
    LOADL *init* *laadt een init waarde*
    STORE(1) d[SB] *d is relatief t.o.v. SB, adres van Identifier*

```
AssignStatement      ::=    ExpressionOr ( := ExpressionOr )?
```
Assign [*ExpressionOr*] =
    STORE(1) d[SB] *d is relatief aan SB, ExpressionOr is een afgedwongen Identifier*
    LOAD(1) d[SB] *d is relatief aan SB, assignment levert toegewezen waarde (mult. ass.)*

Voor de *unary Expression**** expressies geldt de volgende vertaalregel:
Evaluate [O *Expression****] =
    Evaluate *Expression****
    CALL p *p is een TAM routine die correspondeert met O*

Voor de *binary Expression**** expressies geldt de volgende vertaalregel:
Evaluate [*Expression1**** O *Expression2****] =

Evaluate *Expression1****
Evaluate *Expression2****
CALL p *p is een TAM routine die correspondeert met O*

In het geval van de *eq* en *ne* wordt er ook nog een LOADL 1 uitgevoerd.

Fetch [Identifier] =

LOAD(1) d[SB] *d is relatief aan SB, adres van Identifier*

Evaluate [*true*] =

LOADL 1 *true wordt intern gerepresenteerd als 1*

Evaluate [*false*] =

LOADL 0 *false wordt intern gerepresenteerd als 0*

Evaluate [*Number*] =

LOADL v *waarde v van het getal wordt op stack geladen*

Evaluate [*Character*] =

LOADL v *ASCII waarde v van het karakter wordt op de stack geladen*

```
Read                ::=    read ( VarList )
VarList             ::=    Identifier ( , Identifier )*
```

Execute [*Read*] =

Evaluate *Identifier*
LOADA d[SB] *d is relatief aan SB, adres van Identifier*
CALL get/getint *get of getint, gebaseerd op type van Identifier*

```
Print               ::=    print ( ExpressionList )
ExpressionList      ::=    Expression ( , Expression )*
```

Execute [*Print*] =

Evaluate *Expression*
LOADA -1[ST] *laadt het adres van de variabele of constante dat aan de top staat*
LOADI(1) *laadt de waarde van het adres; resulteert in duplicatie van top item*
CALL put/putint *put of putint, gebaseerd op type van Expression*
CALL puteol *print de eol*

```
ExpressionIf          ::=    if DeclStatBlocks ExpressionThen fi
ExpressionThen        ::=    then DeclStatBlocks ( ExpressionElse )?
ExpressionElse        ::=    else DeclStatBlocks
```

Elaborate [*ExpressionIf*] =

Evaluate *DeclStatBlocks*

JUMPIF(0) ELSE# *als DeclStatBlocks naar 0 evalueert, springen naar Else, met label #*

Elaborate [*ExpressionThen*] =

Evaluate *DeclStatBlocks*

JUMP ENDIF# *spring naar het einde, met labelnummer #*

Elaborate [*ExpressionElse*] =

Evaluate *DeclStatBlocks*

JUMP ENDIF# *spring naar het einde, met labelnummer #*

```
WhileStatement        ::=    while DeclStatBlocks ExpressionDo od
ExpressionDo          ::=    do DeclStatBlocks
```

Elaborate [*WhileStatement*] =

Evaluate *DeclStatBlocks*

JUMPIF(0) ENDWHILE# *spring naar het einde van de whileloop met labelnummer #*

Elaborate [*ExpressionDo*] =

Evaluate *DeclStatBlocks*

JUMP WHILE# *spring begin van whileloop, met labelnummer #*

Vertaalregels - JAM

```
Program              ::=    DeclStatBlocks EOF
```
Run [*DeclStatBlocks*] =
    Elaborate *DeclStatBlocks*

Elaborate [*DeclStatBlocks*] =
    Elaborate *Declaration* *één van de mogelijkheden van DeclStatBlocks*

Elaborate [*DeclStatBlocks*] =
    Elaborate *Expression* *de andere mogelijkheid van DeclStatBlocks*

```
ConstDeclaration     ::=    const Identifier := Expression
```
Elaborate [*ConstDeclaration*] =
    Evaluate *Expression* *evalueert de expressie (kan variabelen bevatten)*
    istore # *wijst de geëval. waarde toe aan local variable #*

```
VarDeclaration       ::=    var Identifier : Type
```
Elaborate [*VarDeclaration*] =
    ldc *init* *laadt een init waarde*
    istore # *slaat standaardwaarde op in local variable #*

```
AssignStatement      ::=    ExpressionOr ( := ExpressionOr )?
```
Assign [*ExpressionOr*] =
    istore # *# is adres van Identifier (ExpressionOr is een Identifier)*
    iload # *# is adres van Identifier, i voor int, a voor char*

Voor de *unary Expression**** expressies geldt de volgende vertaalregel:
Evaluate [O *Expression****] =
    Evaluate *Expression****
    p *p is een Jasmin routine die correspondeert met O en de types*

Voor de *binary Expression**** expressies geldt de volgende vertaalregel:
Evaluate [*Expression1**** O *Expression2****] =
    Evaluate *Expression1****
    Evaluate *Expression2****
    p *p is een Jasmin routine die correspondeert met O en de types*

Fetch [Identifier] =
    iload # *laadt local variable #, i voor int, a voor char*

Evaluate [*true*] =
    ldc 1 *true wordt intern gerepresenteerd als 1*

Evaluate [*false*] =

ldc 0 *false wordt intern gerepresenteerd als 0*

Evaluate [*Number*] =

ldc v *waarde v van het getal wordt op stack geladen*

Evaluate [*Character*] =

ldc "v" *stringrepresentatie van Character v wordt op stack geladen*

```
Read                ::=    read ( VarList )
VarList             ::=    Identifier ( , Identifier )*
```

Execute [*Read*] = (*Identifier* heeft type karakter)

getstatic java/lang/System/in Ljava/io/InputStream;
invokevirtual java/io/InputStream/read()I
istore #
iload #

```
Read                ::=    read ( VarList )
VarList             ::=    Identifier ( , Identifier )*
```

Execute [*Read*] = (*Identifier* heeft niet type karakter)

invokestatic "+className+".readint()I
istore #
iload #

Bovenstaande invokestatic gebruikt een zelf gedefinieerde methode. Declaratie van deze methode is te vinden in ChronosJAMGeneratorToolbox.prepareReadInt()

```
Print               ::=    print ( ExpressionList )
ExpressionList      ::=    Expression ( , Expression )*
```

Execute [*Print*] = (*Expression* heeft type karakter)

Evaluate *Expression*
dup
i2c
invokestatic java/lang/String/valueOf(C)Ljava/lang/String;
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

```
Print               ::=    print ( ExpressionList )
ExpressionList      ::=    Expression ( , Expression )*
```

Execute [*Print*] = (*Expression* heeft niet type karakter)

Evaluate *Expression*
dup
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(I)V

```
ExpressionIf          ::=    if DeclStatBlocks ExpressionThen fi
ExpressionThen        ::=    then DeclStatBlocks ( ExpressionElse )?
ExpressionElse        ::=    else DeclStatBlocks
```

Elaborate [*ExpressionIf*] =

Evaluate *DeclStatBlocks*

ifeq ELSE# *als DeclStatBlocks naar 0 evalueert, springen naar ELSE, met label #*

Elaborate [*ExpressionThen*] =

Evaluate *DeclStatBlocks*

goto ENDIF# *spring naar het einde, met labelnummer #*

Elaborate [*ExpressionElse*] =

Evaluate *DeclStatBlocks*

goto ENDIF# *spring naar het einde, met labelnummer #*

```
WhileStatement        ::=    while DeclStatBlocks ExpressionDo od
ExpressionDo          ::=    do DeclStatBlocks
```

Elaborate [*WhileStatement*] =

Evaluate *DeclStatBlocks*

ifeq ENDWHILE# *spring naar het einde van de whileloop met labelnummer #*

Elaborate [*ExpressionDo*] =

Evaluate *DeclStatBlocks*

goto WHILE# *spring naar begin van whileloop, met labelnummer #*

Beschrijving Java-programmatuur

Om de Chronos compiler te realiseren zijn er uiteraard naast de grammatica bestanden en de door ANTLR gegenereerde bronbestanden enkele andere klassen geïmplementeerd om de vertaler te realiseren. Deze klassen zijn hieronder kort beschreven. Uiteraard verwijzen we graag naar de Javadoc van de verschillende klassen als men meer wil weten over bepaalde klassen en/of functionaliteiten die deze klassen vervullen.

Chronos.java [chronos]
Dit is de klasse die de gehele compiler als het ware aan elkaar lijmt. Deze bevat de mainmethode om de vertaler te starten en zal de opties correct parsen. Er worden instanties van *ChronosLexer*, *ChronosParser*, *ChronosTAMGenerator* aangemaakt om de verschillende stadia van de vertaler te realiseren. Daarnaast is het ook een mogelijkheid dat er een *ChronosJAMAdministrator* en een *ChronosJAMGenerator* geïnstantieerd worden om Jasmin code te genereren. De klasse is sterk gebaseerd op code die beschikbaar was tijdens het practicum.

ChronosSymbolTable.java [chronos.utils.symbols]
Deze klasse wordt gebruikt om *ChronosIdentifierEntry*s in op te slaan met hun bijbehorende level. Er zijn methoden aanwezig voor het verhogen en verlagen van de scope, en het invoeren en opvragen van een *ChronosIdentifierEntry*.

ChronosIdentifierEntry.java [chronos.utils.symbols]
Een *ChronosIdentifierEntry* wordt opgeslagen in de *ChronosSymbolTable*. *ChronosIdentifierEntry*s bevatten daarnaast informatie over *Identifiers* die in een programma gedeclareerd worden. Zo kunnen de (dummy) waarde, het level waarop de *Identifier* gedeclareerd is, het type, het adres en de variabele/constante status opgeslagen worden. Deze informatie wordt dus niet in een AST node opgeslagen.

ChronosCheckerToolbox.java [chronos.utils]
Deze klasse wordt door de *ChronosChecker* gebruikt om het daadwerkelijke contextueel checken van een Chronos programma uit te voeren. Er zijn diverse methoden aanwezig om types en declaraties te checken. Zo zal er bij variabele en constanten declaraties gecheckt worden of *Identifier*s reeds gedeclareerd zijn, en zal er een foutmelding opgeleverd worden als dit inderdaad het geval is. Hierbij wordt gebruik gemaakt van de *ChronosSymbolTable*. Alle functionaliteit voor het uitvoeren van type checking zit ook in de *ChronosCheckerToolbox*. Er zijn diverse methoden die checken of een bepaald type juist is en of toewijzingen van een bepaalde expressie aan een variabele de juiste types hebben. Bevat tevens functionaliteit voor het zelf aanmaken van een foutmelding voor de *ChronosCheckerErrorReporter* op basis van een *CommonTree* of *TreeRuleReturnScope*.

ChronosTAMGeneratorToolbox.java [chronos.utils]
De klasse bevat alle functionaliteit voor het genereren van de TAM code. Voor eigenlijk elke grammaticaregel is een methode geïmplementeerd die de overeenkomstige instructie naar de output (console of bestand) print. Er wordt een *ChronosSymbolTable* bijgehouden om van *ChronosIdentifierEntry*s het type en het adres op te kunnen vragen. Er zijn diverse hulpmethoden aanwezig om het printen van TAM code te vergemakkelijken.

ChronosJAMGeneratorToolbox.java [chronos.utils]
De *ChronosJAMGeneratorToolbox* bevat alle functionaliteit die benodigd is om code voor Jasmin te genereren. Deze klasse komt grotendeels overeen met de *ChronosTAMGeneratorToolbox*, maar bevat enkele aanvullende methodes om het correct printen van bijvoorbeeld de header en de start van een Jasmin (Java) programma correcte kunnen genereren.

IChronosErrorReporter.java [chronos.utils.error]
Interface die beschrijft waar implementatie van een *ErrorReporter* aan moet voldoen. Deze moet over een methode beschikken om errors toe te voegen en om deze weer te kunnen opvragen.

ChronosParserErrorReporter.java [c *ChronosCheckerErrorReporter* **hronos.utils.error]**
Bevat functionaliteit voor het rapporteren van fouten tijdens de parser-fase van de compiler. Deze klasse wordt gebruikt door de *ChronosParser*. Aan het eind van parser-fase wordt er bekeken of er fouten zijn gevonden door deze op te vragen. Is dit het geval, dan zal executie van de compiler stoppen; anders zal de checker-fase starten.

ChronosCheckerErrorReporter.java [chronos.utils.error]
Bevat functionaliteit voor het rapporteren van fouten tijdens de checker-fase van de compiler. Deze klasse wordt gebruikt door de *ChronosChecker*. Aan het eind van de checker-fase wordt er bekeken of er fouten zijn gevonden door deze op te vragen aan de. Als dit het geval is, dan zal de executie van de compiler stoppen, anders zal er TAM dan wel Jasmin code gegenereerd worden.

ChronosException.java [chronos.utils.exceptions]
Een *ChronosException* is een onderklasse van *RecognitionException*. Deze wordt gebruikt door de *ChronosChecker* om foutmeldingen terug te geven als een programma qua context niet correct geschreven is.

ChronosLogFactory.java [chronos.utils.logging]
Bevat functionaliteit voor het aanmaken van een *Logger* voor de Chronos compiler. Deze kan naar de standaard error console schrijven, maar naar een bestand is ook een optie.

ChronosLogFormatter.java [chronos.utils.logging]
Dit is een *Formatter* voor de Logger die *ChronosLogFactory* oplevert. Deze *Formatter* zorgt ervoor dat de output van de *Logger* netjes weergegeven wordt.

ChronosTestCase.java [chronos.test]
Beschrijft een test case voor de Chronos compiler. Deze test case bestaat uit een te testen Chronos programma, de verwachte uitkomst en de gegeneereerde uitkomst. Daarnaast is er een methode aanwezig om te bepalen of de test succesvol is. Deze test hangt af van het feit of de inhoud van de bestanden voor verwachte en gegenereerde uitkomst hetzelfde is.

ChronosTester.java [chronos.test]
Dit is de tweede mainklasse van de Chronos compiler die gebruikt kan worden om de gehele compiler te testen. Er zullen *ChronosTestCase*s aangemaakt worden op basis van de vier verschillende test stages. Deze tests worden automatisch toegevoegd. Als een test stage niet

succesvol doorlopen wordt, zal het programma de volgende test stage niet uitvoeren. Als een bepaalde test stage wel goed doorlopen wordt, zal de *ChronosTester* doorgaan met het testen van de volgende test stage, totdat alle tests doorlopen zijn. Alle tests worden automatisch door *ChronosTester* aangemaakt; er dienen enkel test programma's en bijbehorende verwachte uitkomst gedefinieerd te zijn door de programmeur. De klasse heeft geen opties om mee te geven bij de start van het programma.

Testplan en –verslagen

Testen is een essentieel onderdeel bij het maken van een enigszins complex programma zoals de Chronoscompiler die in dit verslag gepresenteerd is. Hieronder staat beschreven hoe het testen aangepakt is. Vervolgens worden twee uitgebreide testprogramma's beschreven die de functionaliteit van de Chronoscompiler demonstreren.

Bij het testen is gebruik gemaakt van de hiervoor beschreven klassen ChronosTestCase en ChronosTester. Met deze klassen is een automatisch testproces op te starten waarbij de verschillende functies van de Chronoscompiler getest worden. De tests worden uit de bestandsstructuur onder "test" gehaald en de vier directe submappen van test geven de vier testfasen van de compilertest weer.

De testfasen zijn opeenvolgend: 1_syntactic, 2_contextual, 3_tamcode en 4_jamcode. Als een fase niet slaagt, worden de daarop volgende fasen niet uitgevoerd. Hiervoor is gekozen aangezien een fout in de syntactische fase eerst opgelost moet worden voordat de contextuele fase getest kan worden enzovoort. De enige uitzondering hierop zijn mogelijk fase 3 en 4, aangezien TAM en JAM code onafhankelijk van elkaar fouten kunnen bevatten. Vaak is de TAM code echter gemakkelijker te genereren en als hier fouten in zitten, blijken deze vaak ook bij de JAM code generatie voor te komen. Daar komt bij dat het geen kwaad kan om te eisen dat fouten in de TAM code generatie opgelost moeten worden voordat de JAM code generatie getest wordt, aangezien we beide implementaties uiteindelijk correct wilden hebben.

Alle testfasen bevatten testprogramma's die eindigen op de .era extensie. Van deze programma's is bekend wat de verwachte uitvoer is. Dit is vastgelegd in een .xout bestand. De uitvoer van de test van het testprogramma wordt naar een .gout bestand weggeschreven. De testklassen vergelijken de gegenereerde output vervolgens automatisch met de verwachte output en genereren een foutmelding als deze niet overeenkomen.

Er zijn verschillende soorten verwachten en gegenereerde output. Bij de syntactische en bij de contextuele fase zijn dit typisch foutmeldingen die gegenereerd horen te worden door de compiler als geprobeerd wordt het bijbehorende testbestand te compileren. In de TAM en JAM code generatiefasen is de verwachte output typisch de TAM of JAM code die gegenereerd zou moeten worden. In tegenstelling tot de meeste testbestanden in de syntactische en contextuele fasen bevatten de testbestanden voor code generatie daarom correcte programma's.

De bedoeling is dat in de syntactische fase alle syntactische fouten getest worden. In de testbestanden voor de contextuele fase zitten dus geen syntactische fouten meer. Als de syntactische fase bij het automatisch testen zonder fouten doorlopen wordt, zouden de testbestanden in de contextuele fase dus ook goed door de syntactische analyse moeten komen. Op dezelfde manier horen de testprogramma's voor de code generatie fasen zonder fouten door de syntactische en de contextuele fasen te komen.

Bij de testbestanden onder 1_syntactic worden de lexer en parser getest. Onder 1_syntactic staan daarom submappen met testbestanden waar syntactische fouten in zitten. De syntactische fouten zijn onderverdeeld in de volgende vier categorieën: 1_program_structure, 2_declarations, 3_assignments en 4_expressions. De compiler pakt de syntactische fouten hierin zonder problemen op en genereert de verwachte foutmeldingen.

De testbestanden onder 2_contextual zijn onderverdeeld in dezelfde categorieën als de syntactische fase met uitzondering van de structuur van het programma. De categorieën zijn dus 1_declarations, 2_assignments en 3_expressions. Ook de contextuele fouten in deze tests worden correct gevonden door de Chronoscompiler en de verwachte foutmeldingen worden gegenereerd.

Beide code generatiefasen, 3_tamcode en 4_jamcode, zijn onderverdeeld in de vier categorieën: 1_basic, 2_cond, 3_loop en 4_total. Bij deze fasen is de verwachte output gemaakt door de gegenereerde output handmatig te controleren en vervolgens als verwachte output te kopiëren. Dit zorgt ervoor dat als een onderdeel breekt bij toekomstige wijzigingen, dit door de test opgepakt wordt. Om te controleren of de gegenereerde code echt correct is, kan deze code uitgevoerd worden om zo te zien of de code inderdaad uitvoerbaar is en om te kijken of de output van het programma overeenkomt met wat de programmeur verwacht had.

Bij de code generatie fasen staan in de map 4_total twee uitgebreide testprogramma's genaamd test_one.era en test_two.era. Deze programma's zijn correct en ze demonstreren de functies die de Chronoscompiler ondersteunt. Bij een testrun (het uitvoeren van ChronosTester) wordt voor deze programma's de code gegenereerd. Om te controleren of deze gegenereerde code doet wat het programma hoort te doen kan deze code uitgevoerd worden.

In appendix genaamd *In- en uitvoer van testprogramma* staan de twee genoemde, uitgebreide testprogramma's evenals de in- en uitvoer die bij deze programma's hoort. Bij TAM code generatie werken beide testprogramma's naar behoren al kunnen er bij het inlezen van characters problemen optreden. Vaak treden deze problemen niet op als alle invoer (gescheiden door spaties) van het programma gegeven wordt bij de eerste read in het programma. Dit is echter een rare eis voor een normaal programma en Chronos blijft daarin in gebreke. Bij JAM code generatie werkt de read-functie niet geheel naar behoren bij het inlezen van characters en bij het inlezen van booleans. Dezelfde workaround als bij TAM code generatie werkt hiervoor ook.

In test_two.era komen de volgende onderdelen van een programmeertaal voor. Deze worden allemaal door de Chronoscompiler ondersteunt voor zowel TAM als JAM code generatie.

1. Declaraties:
    a. Constanten en variabelen zijn te declareren en krijgen de goede beginwaarde. Bij constanten kunnen ook expressies als waarde toegewezen worden.
2. Toewijzingen:
    a. Aan variabelen kunnen waarden toegewezen worden.
    b. Multiple assignments worden ondersteund.
    c. De waarde van een expressie kan aan een variabele toegewezen worden.
    d. Een read-statement kan aan een variabele toegewezen worden.
    e. Een print-statement kan aan een variabele toegewezen worden.
3. Expressies:
    a. Unary symbols kunnen gebruikt worden en deze worden samen met rekensymbolen in de goede volgorde behandeld.
    b. Vergelijkingen met <, >, =<, >=, ==, != werken correct.
    c. If-statements worden correct uitgevoerd. Deze kunnen met of zonder else-blok zijn.
    d. While-statements worden correct uitgevoerd. Ook geneste if- en while-statements werken correct.

e. Decl_stat_blocks worden goed behandeld en declaraties hierbinnen zijn mogelijk.

4. Read-functie[1]:
    a. Variabelen kunnen ingelezen worden vanaf de standaardinvoer.
    b. Meerdere variabelen kunnen met één read-aanroep ingelezen worden.
5. Print-functie:
    a. Variabelen en constanten kunnen geprint worden naar de standaarduitvoer.
    b. Expressies en toewijzingen kunnen geprint worden.
    c. Meerdere van bovenstaande mogelijkheden kunnen met één print-aanroep geprint worden.

[1] De read-functie werkt niet geheel correct bij het inlezen van een character bij TAM code generatie en bij het inlezen van characters en booleans bij JAM code generatie.

Specificatie verantwoordelijkheden

Wie	Taak
Beide	Lexer & Parser grammatica
Beide	Checker grammatica
Herman	TAM generator grammatica
Herman	JAM administrator & generator grammatica
Martijn	ChronosTester & ChronosTestCase
Martijn	Testen van de compiler
Herman	ChronosCheckerToolbox
Herman	ChronosTAMGeneratorToolbox
Herman	ChronosJAMGeneratorToolbox
Martijn	Testen en fixen van o.a. ChronosCheckerToolbox, ChronosTAMGeneratorToolbox & ChronosJAMGeneratorToolbox
Herman	Verslag: Inleiding, Syntax, Appendices, Layout
Martijn	Verslag: Beschrijving, Problemen, Contextuele Beperkingen, Semantiek, Testplan en –verslagen
Beide	Verslag: Vertaalregels, Conclusies, Stijl- en spellingscheck, Eindredactie

Conclusies

Het bouwen van de Chronos vertaler is eigenlijk zeer goed verlopen. We hebben niet met hele grote problemen gezeten, en het lukte ons aardig om eventuele obstakels vlot uit de weg te ruimen. Dit stelde ons in staat om een uitstapje te maken naar het genereren van Jasmin code. Dit leek ons een interessante keuze. Het resultaat mag er volgens ons best zijn; de Chronos vertaler is namelijk in staat om zowel TAM als Jasmin code te genereren.

De ontwikkelde programmeertaal is niet heel erg uitgebreid; hiervoor hebben we moeten kiezen omwille van de hoeveelheid beschikbare tijd aan het eind van het kwartiel. Procedures en functies leken ons wel interessant om te implementeren, maar we voorzagen dat dit ons te veel tijd zou gaan kosten om deze volledig en correct volgens de richtlijnen van de practicumhandleiding te kunnen implementeren. We hebben ons daarom zo veel mogelijk gefocust op het correct laten werken van de vertaler: de functionaliteit van de programmeertaal moest precies overeenkomen met die zoals in de practicumhandleiding beschreven was. Daarvoor was veelvuldig testen van de vertaler noodzakelijk.

Al vroeg tijdens de ontwikkeling van de vertaler werd duidelijk dat we een solide raamwerk voor het testen van de Chronos vertaler wilden gebruiken. Dit zou ervoor zorgen dat we niet veel tijd zouden kwijt zijn aan het handmatig testen van allerlei verschillende testcases. We hebben zelf een framework ontwikkeld dat naar onze mening goed zijn werk heeft gedaan.

Daar Chronos een vrij simpele programmeertaal is gebleven, is er uiteraard ruimte voor toevoegingen op het systeem. Procedures en functies zijn een logische eerste stap. Initieel zaten deze wel in de grammatica, maar in de loop van het project zagen we dat we hier niet volledig aan toe zouden komen. Verder zouden er in de Chronos vertaler nog enkele optimalisaties toegepast kunnen worden, zoals het verminderen van de hoeveelheid *pops* die voor bepaalde programma's genereerd worden.

We hebben van de eindopdracht geleerd dat het bouwen van een (simpele) vertaler best veel werk is. Daarbij werd duidelijk dat ANTLR de programmeur zeer veel werk uit handen neemt, en we zullen zeker opnieuw ANTLR, of een soortgelijk framework, gebruiken als we nog eens een vertaler of interpreter moeten bouwen.

Appendix

Specificatie ChronosLexer & ChronosParser

```
grammar Chronos;

options {
  language = Java;
  k=1;
  output = AST;
}

tokens {
  //punctuation
  PROGRAM     = 'program';
  COLON       = ':';
  COMMA       = ',';
  SEMICOLON   = ';';
  LCURLY      = '{';
  RCURLY      = '}';
  LPAREN      = '(';
  RPAREN      = ')';

  //keywords
  CONST       = 'const';
  VAR         = 'var';
  READ        = 'read';
  PRINT       = 'print';
  WHILE       = 'while';
  DO          = 'do';
  ENDDO       = 'od';
  IF          = 'if';
  ENDIF       = 'fi';
  THEN        = 'then';
  ELSE        = 'else';
  BECOMES     = ':=';

  //binary operators
  GT          = '>';
```

```
  GE          = '>=';
  LT          = '<';
  LE          = '<=';
  EQ          = '==';
  NEQ         = '!=';

  //logic operators
  NOT         = '!';
  OR          = '||';
  AND         = '&&';

  //operators
  PLUS        = '+';
  MINUS       = '-';
  MULT        = '*';
  DIV         = '/';
  MOD         = '%';

  //booleans
  TRUE        = 'true';
  FALSE       = 'false';

  //unary operators
  PLUSU       = 'plusu';
  MINUSU      = 'minusu';

  //indicator for indented block
  IDESTBLOCK  = 'indent';

  //types
  INTEGER   = 'int';
  CHAR      = 'char';
  BOOLEAN   = 'bool';
  VOID      = 'void';

  //apostrof used for single characters
  APOSTROF    = '\'';  //verwijder regel om syntax highlighting terug te krijgen

}

@lexer::header {
  package chronos;
  import chronos.utils.error.IChronosErrorReporter;
```

```
}

@header {
  package chronos;
  import chronos.utils.error.IChronosErrorReporter;
}

@members {

    private IChronosErrorReporter errorReporter = null;

    public void setErrorReporter(IChronosErrorReporter errorReporter) {
        this.errorReporter = errorReporter;
    }

    @Override
    public void displayRecognitionError(String[] tokenNames,RecognitionException e) {
        String hdr = getErrorHeader(e);
        String msg = getErrorMessage(e, tokenNames);
        if (errorReporter != null){
            errorReporter.addError(hdr, msg);
        } else {
            System.err.print("NO-ERRORREPORTER: ");
            super.displayRecognitionError(tokenNames, e);
        }
    }
}

//parser rules
program
    :   decl_stat_blocks EOF
            ->^(PROGRAM decl_stat_blocks)
    ;

decl_stat_blocks
    :   ((declaration SEMICOLON!)* expression SEMICOLON!)+
    ;

indent_decl_stat_blocks
    :   LCURLY decl_stat_blocks RCURLY
          -> ^(IDESTBLOCK decl_stat_blocks)
    ;
```

```
declaration
    :   constant_declaration
    |   variable_declaration
    ;

constant_declaration
    :   CONST^ IDENTIFIER BECOMES! expression
    ;

variable_declaration
    :   VAR^ IDENTIFIER COLON! type
    ;

type
    :   INTEGER
    |   CHAR
    |   BOOLEAN //VOID???
    ;

expression
    :   assign_statement
    |   while_statement
    ;

assign_statement
  :  expror (BECOMES^ expror)* //multiple assignment
  ;

expror
  :  exprand (OR^ exprand)*
  ;

exprand
  :   exprrel (AND^ exprrel)*
  ;

exprrel
  : exprplusminus ((GT^ | GE^ | LT^ | LE^ | EQ^ | NEQ^) exprplusminus)*
  ;

exprplusminus
    :   exprmultdiv ((PLUS^ | MINUS^) exprmultdiv)*
    ;
```

```
exprmultdiv
  : exprunary ((MOD^ | MULT^ | DIV^) exprunary)*
  ;

exprunary
  :   operand
  |   NOT^ operand
  |   PLUS operand
      -> ^(PLUSU operand)
  |   MINUS operand
      -> ^(MINUSU operand)
  ;

operand
  :   TRUE
  |   FALSE
  |   NUMBER
  |   CHARACTER
  |   IDENTIFIER
  |   LPAREN! expression RPAREN!
  |   indent_decl_stat_blocks
  |   read
  |   print
  |   exprif
  ;

read
  :   READ^ LPAREN! varlist RPAREN!
  ;

print
  :   PRINT^ LPAREN! exprlist RPAREN!
  ;

varlist
  :   IDENTIFIER (COMMA! IDENTIFIER)*
  ;

exprlist
  :   expression (COMMA! expression)*
  ;
```

```
exprif
  :   IF^ decl_stat_blocks exprthen ENDIF!
  ;

exprthen
  :   THEN^ decl_stat_blocks (exprelse)?
  ;

exprelse
  :   ELSE^ decl_stat_blocks
  ;

while_statement
    :   WHILE^ decl_stat_blocks exprdo ENDDO!
    ;

exprdo
    :   DO^ decl_stat_blocks
    ;

//lexer rules

COMMENT
  :   '//' .* '\n'
        { $channel=HIDDEN; }
  ;

WS
  :   (' ' | '\t' | '\f' | '\r' | '\n')+
        { $channel=HIDDEN; }
  ;

IDENTIFIER
  :   LETTER (LETTER | DIGIT)*
  ;

NUMBER
  :   DIGIT+
  ;

CHARACTER
  :   APOSTROF SINGLECHAR APOSTROF
  ;
```

```
fragment DIGIT        :   ('0'..'9') ;
fragment LOWER        :   ('a'..'z') ;
fragment UPPER        :   ('A'..'Z') ;
fragment LETTER       :   (LOWER | UPPER) ;
fragment SYMBOL       :   (' '|'-');
fragment SINGLECHAR   :   (LETTER | SYMBOL | DIGIT);
```

Specificatie ChronosChecker

```
tree grammar ChronosChecker;

options {
  language = Java;
  output = AST;
  tokenVocab = Chronos;
  ASTLabelType = CommonTree;
  k=1;
}

@header {
  package chronos;

  import chronos.utils.error.IChronosErrorReporter;
  import chronos.utils.ChronosCheckerToolbox;
  import chronos.utils.exceptions.ChronosException;
}

@rulecatch {

    catch (ChronosException e){
        if (errorReporter != null){
            errorReporter.addError(e.getMessage(),"");
        } else {
            throw e;
        }
    }

    catch(RecognitionException e){
        if (errorReporter != null){
            errorReporter.addError(e.getMessage(),"");
        } else {
            System.err.println("Recognition Exception Caught " + e);
        }
    }

}
```

```
@members {

    private IChronosErrorReporter errorReporter = null;

    public void setErrorReporter(IChronosErrorReporter errorReporter) {
        this.errorReporter = errorReporter;
    }

    private ChronosCheckerToolbox toolbox = new ChronosCheckerToolbox();

}

program
    : ^(PROGRAM decl_stat_blocks+)
    ;

decl_stat_blocks returns [ String type = ""; ]
  :   d=declaration                     { $type = $d.type;   }
  |   ex=expression                     { $type = $ex.type;  }
  ;

declaration returns [ String type = "";]
    :   cd=constant_declaration         { $type = $cd.type; }
    |   vd=variable_declaration         { $type = $vd.type; }
    ;

constant_declaration  returns [ String type = "";]
    :   ^(CONST id=IDENTIFIER ex=expression)      { toolbox.putConst(id, $ex.type); $type = "no_type"; }
    ;

variable_declaration returns [ String type = "";]
    :   ^(VAR id=IDENTIFIER t=type)               { toolbox.putVar(id, $t.type); $type = "no_type";    }
    ;


expression returns [ String type = ""; ]
  :   ^(PLUSU expr=expression)                    { $type = toolbox.checkInt($expr.type, expr);  }
  |   ^(MINUSU expr=expression)                   { $type = toolbox.checkInt($expr.type, expr);  }
  |   ^(NOT expr=expression)                      { $type = toolbox.checkBool($expr.type, expr); }
  |   ^(OR expr1=expression expr2=expression)     { toolbox.checkBool($expr1.type, expr1); $type = toolbox.checkBool($expr2.type, expr2); }
  |   ^(AND expr1=expression expr2=expression)    { toolbox.checkBool($expr1.type, expr1); $type = toolbox.checkBool($expr2.type, expr2); }
  |   ^(PLUS expr1=expression expr2=expression)   { toolbox.checkInt($expr1.type, expr1);  $type = toolbox.checkInt($expr2.type, expr2);  }
  |   ^(MINUS expr1=expression expr2=expression)  { toolbox.checkInt($expr1.type, expr1);  $type = toolbox.checkInt($expr2.type, expr2);  }
```

```
  |   ^(MULT expr1=expression expr2=expression)   { toolbox.checkInt($expr1.type, expr1); $type = toolbox.checkInt($expr2.type, expr2);   }
  |   ^(DIV expr1=expression expr2=expression)    { toolbox.checkInt($expr1.type, expr1); $type = toolbox.checkInt($expr2.type, expr2);   }
  |   ^(MOD expr1=expression expr2=expression)    { toolbox.checkInt($expr1.type, expr1); $type = toolbox.checkInt($expr2.type, expr2);   }
  |   ^(BECOMES id=IDENTIFIER expr=expression)    { $type = toolbox.checkBecomes(id, $expr.type); }
  |   ^(GT expr1=expression expr2=expression)     { toolbox.checkInt($expr1.type, expr1); toolbox.checkInt($expr2.type, expr2); $type = "bool"; }
  |   ^(GE expr1=expression expr2=expression)     { toolbox.checkInt($expr1.type, expr1); toolbox.checkInt($expr2.type, expr2); $type = "bool"; }
  |   ^(LT expr1=expression expr2=expression)     { toolbox.checkInt($expr1.type, expr1); toolbox.checkInt($expr2.type, expr2); $type = "bool"; }
  |   ^(LE expr1=expression expr2=expression)     { toolbox.checkInt($expr1.type, expr1); toolbox.checkInt($expr2.type, expr2); $type = "bool"; }
  |   ^(EQ expr1=expression expr2=expression)     { toolbox.compareTypes($expr1.type, expr1, $expr2.type, expr2); $type = "bool"; }
  |   ^(NEQ expr1=expression expr2=expression)    { toolbox.compareTypes($expr1.type, expr1, $expr2.type, expr2); $type = "bool"; }
  |   ^(IDESTBLOCK                                { toolbox.tbOpenScope(); }
           (dsb=decl_stat_blocks                  { $type = $dsb.type; })+
                                                  { toolbox.tbCloseScope(); })
  |   ^(PRINT expr1=expression                    { $type = $expr1.type; toolbox.checkPrintVoid($type, expr1);  }
           (expr2=expression                      { $type = "void"; toolbox.checkPrintVoid($expr2.type, expr2); })*)
  |   ^(READ id1=IDENTIFIER                       { $type = toolbox.getReadType(id1); }
           (id2=IDENTIFIER                        { if (toolbox.checkRead(id2)) {
                                                        $type = "void"; };
                                                  })*)
  |   ^(IF                                        { toolbox.tbOpenScope(); }
           d=decl_stat_blocks* ext=exprthen       { toolbox.checkBool($d.type, d);
                                                    $type = $ext.type;
                                                    toolbox.tbCloseScope();
                                                  }
  |   ^(WHILE                                     { toolbox.tbOpenScope(); }
         d=decl_stat_blocks* exprdo)              { toolbox.checkBool($d.type, d);
                                                    $type = "void";
                                                    toolbox.tbCloseScope();
                                                  }
  |   o=operand                                   { $type = $o.type; }
  ;

exprdo returns [ String type = ""; ]
  :   ^(DO                                        { toolbox.tbOpenScope();  }
         dsb=decl_stat_blocks*                    { toolbox.tbCloseScope(); })
  ;
```

```
exprthen returns [ String type = ""; ]
  :   ^(THEN                                    { toolbox.tbOpenScope(); }
           dsb=decl_stat_blocks*                { toolbox.tbCloseScope(); }
           (expr2=exprelse)?)                   {   try {
                                                          $type = toolbox.compareTypes($dsb.type, dsb, $expr2.type, expr2);
                                                      } catch(Exception e) {
                                                          $type = "void";
                                                      }
                                                }
  ;

exprelse returns [ String type = ""; ]
  :   ^(ELSE                                    { toolbox.tbOpenScope(); }
         dsb=decl_stat_blocks*)                 { $type = $dsb.type; toolbox.tbCloseScope(); }
  ;

type returns [String type = "";]
    :   INTEGER         { $type = "int";  }
    |   CHAR            { $type = "char"; }
    |   BOOLEAN         { $type = "bool"; }
    |   VOID            { $type = "void"; }
    ;

operand returns [String type = "";]
  :   FALSE             { $type = "bool"; }
  |   TRUE              { $type = "bool"; }
  |   CHARACTER         { $type = "char"; }
  |   NUMBER            { $type = "int";  }
  |   ^(id=IDENTIFIER   { $type = toolbox.getType(id); })
  ;
```

Specificatie Code Generator – ChronosTAMGenerator

```
tree grammar ChronosTAMGenerator;

options {
  language = Java;
  //output = AST;
  tokenVocab = Chronos;
  ASTLabelType = CommonTree;
}

@header {
  package chronos;

  import chronos.utils.ChronosTAMGeneratorToolbox;
  import chronos.utils.exceptions.ChronosException;
  import chronos.utils.error.IChronosErrorReporter;
}

@rulecatch {

    catch (ChronosException e){
        if (errorReporter != null){
            errorReporter.addError(e.getMessage(),"");
        } else {
            throw e;
        }
    }

}

@members {
    boolean ifCondition = false;
    private ChronosTAMGeneratorToolbox toolbox = new ChronosTAMGeneratorToolbox();

    private IChronosErrorReporter errorReporter = null;

    public void setErrorReporter(IChronosErrorReporter errorReporter) {
        this.errorReporter = errorReporter;
    }

}
```

```
program
  :  ^(PROGRAM (dsb1=decl_stat_blocks             { if ($dsb1.value != null && !$dsb1.value.equals("")){
                                                        toolbox.printClean("dsb cleanup start",1);
                                                  }
                                                 })+) { toolbox.printEnd(); }
  ;

decl_stat_blocks returns [ String value = ""; ]
  :   cd=constant_declaration                     { $value = $cd.value;   }
  |   vd=variable_declaration                     { $value = $vd.value;   }
  |   expr=expression                             { $value = $expr.value; }
  ;

constant_declaration  returns [ String value = "";]
  :   ^(CONST id=IDENTIFIER                       { toolbox.pushConst(id); }
        ex=expression)                            { toolbox.putConst(id, $ex.value); }
  ;

variable_declaration returns [ String value = "";]
  :   ^(VAR id=IDENTIFIER t=type)                 { toolbox.putVar(id, $t.type);   }
  ;


expression returns [ String value = ""; ]
  :   ^(PLUSU expr=expression)                    { $value = $expr.value;  }
  |   ^(MINUSU expr=expression)                   { toolbox.printUMinus();
                                                    $value=$expr.value;
                                                  }
  |   ^(NOT expr=expression)                      { toolbox.printNot();
                                                    $value = $expr.value.equals("0") ? "1" : "0";
                                                  }
  |   ^(OR expr1=expression expr2=expression)     { toolbox.printOr();
                                                    $value = (!$expr1.value.equals("0") && !$expr1.value.equals("") ||
                                                              !$expr2.value.equals("0") && !$expr2.value.equals("")) ? "1" : "0";
                                                  }
  |   ^(AND expr1=expression expr2=expression)    { toolbox.printAnd();
       $value = (!$expr1.value.equals("0") && !$expr1.value.equals("") && !$expr2.value.equals("0") && !$expr2.value.equals("")) ? "1" : "0"; }
  |   ^(PLUS expr1=expression expr2=expression)   { toolbox.printAdd();
                                                    int left = Integer.parseInt($expr1.value);
                                                    int right = Integer.parseInt($expr2.value);
                                                    $value = Integer.toString( left + right );
                                                  }
```

```
|   ^(MINUS expr1=expression expr2=expression)    { toolbox.printSub();
                                                    int left = Integer.parseInt($expr1.value);
                                                    int right = Integer.parseInt($expr2.value);
                                                    $value = Integer.toString( left - right );
                                                  }
|   ^(MULT expr1=expression expr2=expression)     { toolbox.printMult();
                                                    int left = Integer.parseInt($expr1.value);
                                                    int right = Integer.parseInt($expr2.value);
                                                    $value = Integer.toString(left * right);
                                                  }
|   ^(DIV expr1=expression expr2=expression)      { toolbox.printDiv();
                                                    int left = Integer.parseInt($expr1.value);
                                                    int right = Integer.parseInt($expr2.value);
                                                    $value = Integer.toString(left / right); }
|   ^(MOD expr1=expression expr2=expression)      { toolbox.printMod();
                                                    int left = Integer.parseInt($expr1.value);
                                                    int right = Integer.parseInt($expr2.value);
                                                    $value = Integer.toString(left \% right);
                                                  }
|   ^(BECOMES id=IDENTIFIER expr=expression)      { toolbox.assignValue(id, $expr.value );
                                                    $value = $expr.value;
                                                  }
|   ^(GT expr1=expression expr2=expression)       { toolbox.printGT();
                                                    $value = Integer.parseInt($expr1.value) > Integer.parseInt($expr2.value) ? "1" : "0";
                                                  }
|   ^(GE expr1=expression expr2=expression)       { toolbox.printGE();
                                                    $value = Integer.parseInt($expr1.value) >= Integer.parseInt($expr2.value) ? "1" : "0";
                                                  }
|   ^(LT expr1=expression expr2=expression)       { toolbox.printLT();
                                                    $value = Integer.parseInt($expr1.value) < Integer.parseInt($expr2.value) ? "1" : "0";
                                                  }
|   ^(LE expr1=expression expr2=expression)       { toolbox.printLE();
                                                    $value = Integer.parseInt($expr1.value) <= Integer.parseInt($expr2.value) ? "1" : "0";
                                                  }
|   ^(EQ  expr1=expression expr2=expression)       { toolbox.printEQ();
                                                    $value = $expr1.value == $expr2.value ? "1" : "0";
                                                  }
|   ^(NEQ expr1=expression expr2=expression)      { toolbox.printNEQ() ;
                                                    $value = $expr1.value != $expr2.value ? "1" : "0";
                                                  }
```

```
|   ^(IDESTBLOCK                          { toolbox.tbOpenScope(); }
        dsb=decl_stat_blocks              { $value = $dsb.value; }
                                          ({
                                              if(!$value.equals("")) {
                                                  toolbox.printClean("indent dsb cleanup", 1);
                                              }
                                          }
        dsb=decl_stat_blocks              { $value = $dsb.value; })*) {
                                              if($value.equals("")){
                                                  toolbox.tbCloseScope(0);
                                              } else {
                                                  toolbox.tbCloseScope(1);
                                              }
                                          }
|   ^(PRINT ep=exprprint                  { $value = $ep.value; }
        (ep=exprprint                     { toolbox.printClean("print", 2);
                                            $value = "";
                                          }
        (exprprint                        { toolbox.printClean("print", 1);
                                          } )*)?)
|   ^(READ er=exprread                    { $value = $er.value; }
        (exprread                         { toolbox.printClean("read", 2);
                                            $value = "";
                                          }
        (exprread                         { toolbox.printClean("read", 1);
                                          } )*)?)
|   ^(IF                                  { toolbox.tbOpenScope();  }
        dsb=decl_stat_blocks              ({ if(!$dsb.value.equals("")) {
                                                  toolbox.printClean("if cleanup start", 1);
                                              }
                                          }
        dsb=decl_stat_blocks)*            { toolbox.printIf();
                                            ifCondition = $dsb.value.equals("1");
                                          }
        et=exprthen                       { toolbox.printEndIf();
                                            $value = $et.value;
                                           if($et.value == null || $et.value.equals("")){
                                                toolbox.tbCloseScope(0);
                                            } else {
                                                toolbox.tbCloseScope(1);
                                            }
                                          })
```

```
  |  ^(WHILE                                  { Object[] whileInfo = toolbox.printWhile();
                                                toolbox.tbOpenScope();
                                              }
           dsb=decl_stat_blocks                 ({
                                                  if($dsb.value.equals("")) {
                                                      toolbox.printClean("while cleanup start", 1);
                                                  }
                                              }
           dsb=decl_stat_blocks)*               { toolbox.printWhileDo(whileInfo); }
           exprdo)                              { toolbox.printWhileEnd(whileInfo);
                                                  toolbox.tbCloseScope(0);
                                              }
  |   o=operand                               { $value = $o.value; }
  ;

exprprint returns [ String value = ""; ]
  :   expr=expression                         { toolbox.printPrint($expr.value); $value = $expr.value; }
  ;

exprread returns [ String value = ""; ]
  :   id=IDENTIFIER                           { toolbox.printRead(id); $value= toolbox.getValue(id); }
  ;

exprdo returns [ String value = ""; ]
  :^(DO                                       { toolbox.tbOpenScope(); }
           (dsb=decl_stat_blocks               { if(!$dsb.value.equals("")) {
                                                      toolbox.printClean("exprdo cleanup", 1);
                                                  }
                                              })*){ toolbox.tbCloseScope(0); }
  ;
```

```
exprthen returns [ String value = ""; ]
  :  ^(THEN                                    { toolbox.tbOpenScope(); }
            dsb=decl_stat_blocks               ({
                                                 if(!$dsb.value.equals("")) {
                                                     toolbox.printClean("exprthen cleanup", 1);
                                                 }
                                               }
            dsb=decl_stat_blocks)*             { if($dsb.value.equals("")){
                                                     toolbox.tbCloseScope(0);
                                                 } else {
                                                     toolbox.tbCloseScope(1);
                                                 }
                                                 toolbox.printIfElse();
                                               }
            (ee=exprelse)?)                    { if(ifCondition){
                                                     $value = $dsb.value;
                                                  }else{
                                                     $value = $ee.value;
                                                  }
                                               }
  ;


exprelse returns [ String value = ""; ]
  :^(ELSE                                      { toolbox.tbOpenScope(); }
            dsb=decl_stat_blocks               ({ if(!$dsb.value.equals("")) {
                                                     toolbox.printClean("exprelse cleanup", 1);
                                                  }
                                               }
            dsb=decl_stat_blocks)*             { $value = $dsb.value;
                                                  if($dsb.value.equals("")){
                                                      toolbox.tbCloseScope(0);
                                                  } else {
                                                      toolbox.tbCloseScope(1);
                                                  }
                                               })
  ;
```

```
operand returns [String value = "";]
  :   TRUE            { $value = "1"; toolbox.printTrue();             }
  |   FALSE           { $value = "0"; toolbox.printFalse();            }
  |   n=NUMBER        { $value = $n.text; toolbox.printNumber($value); }
  |   c=CHARACTER     { $value = $c.text; toolbox.printChar($value);   }
  |   id=IDENTIFIER   { $value = toolbox.getValue(id);                 }
  ;

type returns [String type = "";]
  :   INTEGER         { $type = "int";  }
  |   CHAR            { $type = "char"; }
  |   BOOLEAN         { $type = "bool"; }
  |   VOID            { $type = "void"; }
  ;
```

Specificatie Code Generator – ChronosJAMAdministrator

```
tree grammar ChronosJAMAdministrator;

options {
  language = Java;
  tokenVocab = Chronos;
  ASTLabelType = CommonTree;
}

@header {
    package chronos;
}

@members {

    int varCount = 0;
    int constCount = 0;

    int stackSize = 0;
    int maxStackSize = 0;

    boolean readFound = false;

    public boolean containsRead(){
        return readFound;
    }

    public int getVarCount(){
        return varCount;
    }

    public int getConstCount(){
        return constCount;
    }

    public int getMaxStackSize(){
        return maxStackSize;
    }

}
```

```
program
    : ^(PROGRAM  (dsb1=decl_stat_blocks)+)          { if (stackSize > maxStackSize){
                                                          maxStackSize = stackSize;
                                                      }
                                                    }
  ;

decl_stat_blocks returns [ String value = ""; ]
  :   ^(CONST id=IDENTIFIER ex=expression)          { constCount++; }
  |   ^(VAR id=IDENTIFIER t=type)                   { varCount++;   }
  |   expr=expression                               { $value = $expr.value; }
  ;

expression returns [ String value = ""; ]
  :   ^(PLUSU expr=expression)                      { $value = $expr.value; }
  |   ^(MINUSU expr=expression)                     { $value=$expr.value;   }
  |   ^(NOT expr=expression)                        { $value = $expr.value.equals("0") ? "1" : "0"; }
  |   ^(OR expr1=expression expr2=expression)       {
$value = (!$expr1.value.equals("0") && !$expr1.value.equals("") || !$expr2.value.equals("0") && !$expr2.value.equals("")) ? "1" : "0"; }
  |   ^(AND expr1=expression expr2=expression)      {
$value = (!$expr1.value.equals("0") && !$expr1.value.equals("") && !$expr2.value.equals("0") && !$expr2.value.equals("")) ? "1" : "0"; }
  |   ^(PLUS expr1=expression expr2=expression)     { int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString( left + right );
                                                    }
  |   ^(MINUS expr1=expression expr2=expression)    { int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString( left - right );
                                                    }
  |   ^(MULT expr1=expression expr2=expression)     { int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString(left * right);
                                                    }
  |   ^(DIV expr1=expression expr2=expression)      { int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString(left / right); }
  |   ^(MOD expr1=expression expr2=expression)      { int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString(left \% right); //escaping needed
                                                    }
  |   ^(BECOMES id=IDENTIFIER expr=expression)      { stackSize++; $value = $expr.value; }
  |   ^(GT expr1=expression expr2=expression)       { $value = Integer.parseInt($expr1.value) > Integer.parseInt($expr2.value) ? "1" : "0";  }
  |   ^(GE expr1=expression expr2=expression)       { $value = Integer.parseInt($expr1.value) >= Integer.parseInt($expr2.value) ? "1" : "0"; }
```

```
  |   ^(LT expr1=expression expr2=expression)       { $value = Integer.parseInt($expr1.value) < Integer.parseInt($expr2.value) ? "1" : "0";  }
  |   ^(LE expr1=expression expr2=expression)       { $value = Integer.parseInt($expr1.value) <= Integer.parseInt($expr2.value) ? "1" : "0"; }
  |   ^(EQ  expr1=expression expr2=expression)      { $value = $expr1.value == $expr2.value ? "1" : "0"; }
  |   ^(NEQ expr1=expression expr2=expression)      { $value = $expr1.value != $expr2.value ? "1" : "0"; }
  |   ^(IDESTBLOCK dsb=decl_stat_blocks             { $value = $dsb.value; }
            (dsb=decl_stat_blocks                  { $value = $dsb.value; })*)
  |   ^(PRINT ep=exprprint                         { $value = $ep.value;  }
            (ep=exprprint                          { $value = ""; }
            (exprprint)*)?)
  |   ^(READ er=exprread                           { $value = $er.value; readFound = true; }
            (exprread                              { $value = "";         }
            (exprread)*)?)
  |   ^(IF dsb=decl_stat_blocks
            (dsb=decl_stat_blocks)*
            et=exprthen                            { $value = $et.value; })
  |   ^(WHILE dsb=decl_stat_blocks
            (dsb=decl_stat_blocks)*
            exprdo)
  |   op=operand                                   { $value = $op.value; }
  ;

exprprint returns [ String value = ""; ]
  :   expr=expression                              { $value = $expr.value; stackSize++;}
  ;

exprread returns [ String value = ""; ]
  :   id=IDENTIFIER                                { $value = "1";  stackSize++; }
  ;

exprdo returns [ String value = ""; ]
  :^(DO (dsb=decl_stat_blocks)*)
  ;

exprthen returns [ String value = ""; ]
  :  ^(THEN dsb=decl_stat_blocks
            (dsb=decl_stat_blocks)*
            (ee=exprelse)?)
  ;

exprelse returns [ String value = ""; ]
  :^(ELSE dsb=decl_stat_blocks
            (dsb=decl_stat_blocks)*                { $value = $dsb.value; })
  ;
```

```
operand returns [String value = "1";]
  :   TRUE            { $value = "1"; stackSize++; }
  |   FALSE           { $value = "0"; stackSize++; }
  |   n=NUMBER        { $value = $n.text; stackSize++; }
  |   c=CHARACTER     { $value = $c.text; stackSize++; }
  |   id=IDENTIFIER   { stackSize++; }
  ;

type returns [String type = "1";]
  :   INTEGER         { $type = "int";  }
  |   CHAR            { $type = "char"; }
  |   BOOLEAN         { $type = "bool"; }
  |   VOID            { $type = "void"; }
  ;
```

Specificatie Code Generator – ChronosJAMGenerator

```
tree grammar ChronosJAMGenerator;

options {
  language = Java;
  tokenVocab = Chronos;
  ASTLabelType = CommonTree;
}

@header {
  package chronos;

  import chronos.utils.ChronosJAMGeneratorToolbox;
  import chronos.utils.exceptions.ChronosException;
  import chronos.utils.error.IChronosErrorReporter;
}

@rulecatch {

    catch (ChronosException e){
        if (errorReporter != null){
            errorReporter.addError(e.getMessage(),"");
        } else {
            throw e;
        }
    }

}

@members {

    private int numberOfVars = 0;
    private int numberOfConsts = 0;
    private int stackSize = 0;
    private boolean readFound = false;
    private String className = "";

    public void setReadFound(boolean b){
        readFound = b;
    }
```

```
    public void setNumberOfVars(int n){
        numberOfVars = n;
    }
    public void setNumberOfConsts(int n){
        numberOfConsts = n;
    }

    public void setStackSize(int n){
        stackSize = n;
    }

    public void setClassName(String cName){
        className = cName;
    }

    boolean ifCondition = false;
    int ifLabel = 0;
    private ChronosJAMGeneratorToolbox toolbox = new ChronosJAMGeneratorToolbox();
    private IChronosErrorReporter errorReporter = null;

    public void setErrorReporter(IChronosErrorReporter errorReporter) {
        this.errorReporter = errorReporter;
    }

    public void printJAMHeader(String s){
        toolbox.printJAMHeader(s);
    }
}

program
    : ^(PROGRAM                                     { toolbox.printProgramStart(numberOfVars, numberOfConsts, stackSize, readFound, className); }
              (dsb1=decl_stat_blocks                {  if (!$dsb1.value.equals("")){
                                                           toolbox.printClean("dsb cleanup start",1);
                                                     }
                                                    })+)
                                                    { toolbox.printProgramEnd(); }
  ;

decl_stat_blocks returns [ String value = ""; ]
  :   cd=constant_declaration                       { $value = $cd.value;   }
  |   vd=variable_declaration                       { $value = $vd.value;   }
  |   expr=expression                               { $value = $expr.value; }
  ;
```

```
constant_declaration  returns [ String value = "";]
    :   ^(CONST
            id=IDENTIFIER                           { toolbox.pushConst(id); }
            ex=expression)                          { toolbox.putConst(id, $ex.value); }
    ;

variable_declaration returns [ String value = "";]
    :   ^(VAR id=IDENTIFIER t=type)                 { toolbox.putVar(id, $t.type); }
    ;

expression returns [ String value = ""; ]
  :   ^(PLUSU expr=expression)                      { $value = $expr.value; /* basically, doing nothing */ }
  |   ^(MINUSU expr=expression)                     { toolbox.printUMinus();
                                                      $value=$expr.value;
                                                    }
  |   ^(NOT expr=expression)                        { toolbox.printNot();
                                                      $value = $expr.value.equals("0") ? "1" : "0";
                                                    }
  |   ^(OR expr1=expression expr2=expression)       { toolbox.printOr();
        $value = (!$expr1.value.equals("0") && !$expr1.value.equals("") || !$expr2.value.equals("0") && !$expr2.value.equals("")) ? "1" : "0"; }
  |   ^(AND expr1=expression expr2=expression)      { toolbox.printAnd();
        $value = (!$expr1.value.equals("0") && !$expr1.value.equals("") && !$expr2.value.equals("0") && !$expr2.value.equals("")) ? "1" : "0"; }
  |   ^(PLUS expre1=expression  expre2=expression)  { toolbox.printAdd();
                                                      int left = Integer.parseInt($expre1.value);
                                                      int right = Integer.parseInt($expre2.value);
                                                      $value = Integer.toString( left + right );
                                                    }
  |   ^(MINUS expre1=expression expre2=expression)  { toolbox.printMinus();
                                                      int left = Integer.parseInt($expre1.value);
                                                      int right = Integer.parseInt($expre2.value);
                                                      $value = Integer.toString( left - right );
                                                    }
  |   ^(MULT expr1=expression expr2=expression)     { toolbox.printMult();
                                                      int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString(left * right);
                                                    }
  |   ^(DIV expr1=expression expr2=expression)      {  toolbox.printDiv();
                                                      int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString(left / right);
                                                    }
```

```
|   ^(MOD expr1=expression expr2=expression)        { toolbox.printMod();
                                                      int left = Integer.parseInt($expr1.value);
                                                      int right = Integer.parseInt($expr2.value);
                                                      $value = Integer.toString(left \% right); //escaping needed
                                                    }
|   ^(BECOMES id=IDENTIFIER expr=expression)        { toolbox.assignValue(id, $expr.value );
                                                      $value = $expr.value;
                                                    }
|   ^(GT expr1=expression expr2=expression)         { toolbox.printGT();
                                                      $value = Integer.parseInt($expr1.value) > Integer.parseInt($expr2.value) ? "1" : "0";
                                                    }
|   ^(GE expr1=expression expr2=expression)         { toolbox.printGE();
                                                      $value = Integer.parseInt($expr1.value) >= Integer.parseInt($expr2.value) ? "1" : "0";
                                                    }
|   ^(LT expr1=expression expr2=expression)         { toolbox.printLT();
                                                      $value = Integer.parseInt($expr1.value) < Integer.parseInt($expr2.value) ? "1" : "0";
                                                    }
|   ^(LE expr1=expression expr2=expression)         { toolbox.printLE();
                                                      $value = Integer.parseInt($expr1.value) <= Integer.parseInt($expr2.value) ? "1" : "0";
                                                    }
|   ^(EQ  expr1=expression expr2=expression)        { toolbox.printEQ();
                                                      $value = $expr1.value == $expr2.value ? "1" : "0";
                                                    }
|   ^(NEQ expr1=expression expr2=expression)        { toolbox.printNEQ();
                                                      $value = $expr1.value != $expr2.value ? "1" : "0";
                                                    }
|   ^(IDESTBLOCK                                   { toolbox.tbOpenScope();  }
        dsb=decl_stat_blocks                       { $value = $dsb.value; }(
                                                   { if(!$value.equals("")) {
                                                       toolbox.printClean("indent dsb cleanup", 1);
                                                     }
                                                   }
        dsb=decl_stat_blocks                       { $value = $dsb.value; })*)
                                                   { if($value.equals("")){
                                                       toolbox.tbCloseScope(0);
                                                     } else {
                                                       toolbox.tbCloseScope(1);
                                                     }
                                                   }
```

```
|   ^(PRINT ep=exprprint                    { $value = $ep.value; }
        (ep=exprprint                       { toolbox.printClean("print", 2);
                                              $value = "";
                                            }
        (exprprint                          { toolbox.printClean("print", 1);
                                            })*)?)
|   ^(READ erread=exprread                  { $value = $erread.value; /*System.err.println("value: " + $value);*/ }
        (exprread                           { toolbox.printClean("read", 2);
                                              $value = "";  //$value leeg zetten, want void
                                            }
        (exprread                           { toolbox.printClean("read", 1);
                                            })*)?)
|   ^(IF                                    { toolbox.tbOpenScope(); }
        dsb=decl_stat_blocks (              { if(!$dsb.value.equals("")) {
                                                  toolbox.printClean("if cleanup start", 1);
                                              }
                                            }
        dsb=decl_stat_blocks)*              { toolbox.printIf();
                                              ifCondition = $dsb.value.equals("1");
                                            }
        et=exprthen                         { toolbox.printEndIf();
                                              $value = $et.value;
                                              if($et.value.equals("")){
                                                  toolbox.tbCloseScope(0); //0
                                              } else {
                                                  toolbox.tbCloseScope(1); //1
                                              }
                                            })
|   ^(WHILE                                 { Object[] whileInfo = toolbox.printWhile();
                                              toolbox.tbOpenScope();
                                            }
        dsb=decl_stat_blocks (              { if($dsb.value.equals("")) {
                                                  toolbox.printClean("while cleanup start", 1);
                                              }
                                            }
        dsb=decl_stat_blocks)*              { toolbox.printWhileDo(whileInfo);
                                            }
        exprdo)                             { toolbox.printWhileEnd(whileInfo);
                                              toolbox.tbCloseScope(0); //0
                                            }
|   oper=operand                            { $value = $oper.value; }
;
```

```
exprprint returns [ String value = ""; ]
  :   expres=expression                          { toolbox.printPrint($expres.value);
                                                   $value = $expres.value;
                                                 }

  ;

exprread returns [ String value = ""; ]
  :   id=IDENTIFIER                              { toolbox.printRead(id);
                                                   $value= toolbox.getValue(id, false);
                                                 }

  ;

exprdo returns [ String value = ""; ]
  :^(DO                                          { toolbox.tbOpenScope(); }
            (dsb=decl_stat_blocks                { if(!$dsb.value.equals("")) {
                                                       toolbox.printClean("exprdo cleanup", 1);
                                                   }
                                                 })*)
                                                 { toolbox.tbCloseScope(0); }

  ;

exprthen returns [ String value = ""; ]
  :  ^(THEN                                      { toolbox.tbOpenScope(); }
            dsb=decl_stat_blocks(                { if($dsb.value.equals("")) {
                                                       toolbox.printClean("exprthen cleanup", 1);
                                                   }
                                                 }
            dsb=decl_stat_blocks)*               { if($dsb.value.equals("")){
                                                       toolbox.tbCloseScope(0); //0
                                                   } else {
                                                       toolbox.tbCloseScope(1); //1
                                                   }
                                                   toolbox.printIfElse();
                                                 }
            (ee=exprelse)?)                      { if(ifCondition){
                                                       $value = $dsb.value;
                                                   } else {
                                                       $value = $ee.value;
                                                   }
                                                    if (ee == null){
                                                       toolbox.printNoElse();
                                                   }
                                                 }  ;
```

```
exprelse returns [ String value = ""; ]
  :^(ELSE                                                { toolbox.tbOpenScope(); }
            dsb=decl_stat_blocks(                        { if(!$dsb.value.equals("")) {
                                                               toolbox.printClean("exprelse cleanup", 1);
                                                           }
                                                         }
            dsb=decl_stat_blocks)*                       { $value = $dsb.value;
                                                           if($dsb.value.equals("")){
                                                               toolbox.tbCloseScope(0); //0
                                                           } else {
                                                               toolbox.tbCloseScope(1); //1
                                                           }
                                                         })
  ;


operand returns [String value = "";]
  :   TRUE            { $value = "1"; toolbox.printTrue();             }
  |   FALSE           { $value = "0"; toolbox.printFalse();            }
  |   n=NUMBER        { $value = $n.text; toolbox.printNumber($value); }
  |   c=CHARACTER     { $value = $c.text; toolbox.printChar($value);   }
  |   id=IDENTIFIER   { $value = toolbox.getValue(id, true);           }
  ;

type returns [String type = "";]
    :   INTEGER         { $type = "int";  }
    |   CHAR            { $type = "char"; }
    |   BOOLEAN         { $type = "bool"; }
    |   VOID            { $type = "void"; }
    ;
```

In- en uitvoer van testprogramma

test_one.era

```
var i: int;
const a:= 'y';

print(i,a);
i := {var d: int; {d := 5;}; d := d-2;};
i := print(i)+2;

const b:= 10;
if i < b-6; then print(a); else var c: int; c := 10; i := 100 + c; fi;

print(i,a,b);

i := if b > i; then
       i := b-2;
else
       var help: int;
       while i > 100; do
              help := i % 8 + help;
              i := i-1;
       od;
       print(help);
fi;

var d: bool;
if !d; then d := !d; fi;

print(d);
d := 5<6 && i>b;
print(d);
```

test_one.era.tasm

```
           PUSH        1             ; push variable i on the stack
           LOADL       0             ; load init value of i
           STORE(1)    0[SB]         ; store in variable i
           PUSH        1             ; push constant a on the stack
           LOADL       121           ; load literal value 121 representing 'y'
           STORE(1)    1[SB]         ; store in variable a
           LOAD(1)     0[SB]         ; load the variable i
           LOADA       -1[ST]        ; load address of int on top of stack
           LOADI(1)                  ; get the int on the stack, again, for printing
           CALL        putint        ; print the integer value of
           CALL        puteol        ; put a newline
           LOAD(1)     1[SB]         ; load the constant a
           LOADA       -1[ST]        ; load address of char on top of stack
           LOADI(1)                  ; get the char on the stack, again, for printing
           CALL        put           ; print the character
           CALL        puteol        ; put a newline
           POP(0)      2             ; pop the resulting value of print
           PUSH        1             ; push variable d on the stack
           LOADL       0             ; load init value of d
           STORE(1)    2[SB]         ; store in variable d
           LOADL       5             ; load literal value 5
           STORE(1)    2[SB]         ; store in variable d
           LOAD(1)     2[SB]         ; load the variable d on the stack
           POP(1)      0             ; pop 0 local variables on closing a scope
           POP(0)      1             ; pop the resulting value of indent dsb cleanup
           LOAD(1)     2[SB]         ; load the variable d
           LOADL       2             ; load literal value 2
           CALL        sub           ; subtract the top entries of the stack from eachother
           STORE(1)    2[SB]         ; store in variable d
           LOAD(1)     2[SB]         ; load the variable d on the stack
           POP(1)      1             ; pop 1 local variables on closing a scope
           STORE(1)    0[SB]         ; store in variable i
           LOAD(1)     0[SB]         ; load the variable i on the stack
           POP(0)      1             ; pop the resulting value of dsb cleanup start
           LOAD(1)     0[SB]         ; load the variable i
           LOADA       -1[ST]        ; load address of int on top of stack
```

```
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
            LOADL       2           ; load literal value 2
            CALL        add         ; add up the entries on the top of the stack
            STORE(1)    0[SB]       ; store in variable i
            LOAD(1)     0[SB]       ; load the variable i on the stack
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            PUSH        1           ; push constant b on the stack
            LOADL       10          ; load literal value 10
            STORE(1)    2[SB]       ; store in variable b
            LOAD(1)     0[SB]       ; load the variable i
            LOAD(1)     2[SB]       ; load the constant b
            LOADL       6           ; load literal value 6
            CALL        sub         ; subtract the top entries of the stack from eachother
            CALL        lt          ; less than statement
            JUMPIF(0)   ELSE11[CB]  ; jump to the else clause
            LOAD(1)     1[SB]       ; load the constant a
            LOADA       -1[ST]      ; load address of char on top of stack
            LOADI(1)                ; get the char on the stack, again, for printing
            CALL        put         ; print the character
            CALL        puteol      ; put a newline
            POP(1)      0           ; pop 0 local variables on closing a scope
            JUMP        ENDIF11[CB] ; jump past the else clause
ELSE11:     PUSH        1           ; push variable c on the stack
            LOADL       0           ; load init value of c
            STORE(1)    3[SB]       ; store in variable c
            LOADL       10          ; load literal value 10
            STORE(1)    3[SB]       ; store in variable c
            LOAD(1)     3[SB]       ; load the variable c on the stack
            POP(0)      1           ; pop the resulting value of exprelse cleanup
            LOADL       100         ; load literal value 100
            LOAD(1)     3[SB]       ; load the variable c
            CALL        add         ; add up the entries on the top of the stack
            STORE(1)    0[SB]       ; store in variable i
            LOAD(1)     0[SB]       ; load the variable i on the stack
            POP(1)      1           ; pop 1 local variables on closing a scope
ENDIF11:    POP(1)      0           ; pop 0 local variables on closing a scope
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOAD(1)     0[SB]       ; load the variable i
            LOADA       -1[ST]      ; load address of int on top of stack
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
            LOAD(1)     1[SB]       ; load the constant a
            LOADA       -1[ST]      ; load address of char on top of stack
            LOADI(1)                ; get the char on the stack, again, for printing
            CALL        put         ; print the character
            CALL        puteol      ; put a newline
            POP(0)      2           ; pop the resulting value of print
            LOAD(1)     2[SB]       ; load the constant b
            LOADA       -1[ST]      ; load address of int on top of stack
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
            POP(0)      1           ; pop the resulting value of print
            LOAD(1)     2[SB]       ; load the constant b
            LOAD(1)     0[SB]       ; load the variable i
            CALL        gt          ; greater than statement
            JUMPIF(0)   ELSE12[CB]  ; jump to the else clause
            LOAD(1)     2[SB]       ; load the constant b
            LOADL       2           ; load literal value 2
            CALL        sub         ; subtract the top entries of the stack from eachother
            STORE(1)    0[SB]       ; store in variable i
            LOAD(1)     0[SB]       ; load the variable i on the stack
            POP(1)      0           ; pop 0 local variables on closing a scope
            JUMP        ENDIF12[CB] ; jump past the else clause
ELSE12:     PUSH        1           ; push variable help on the stack
            LOADL       0           ; load init value of help
            STORE(1)    3[SB]       ; store in variable help
WHILE0:     LOAD(1)     0[SB]       ; load the variable i
            LOADL       100         ; load literal value 100
            CALL        gt          ; greater than statement
            JUMPIF(0)   ENDWHILE0[CB] ; jump past the while body
            LOAD(1)     0[SB]       ; load the variable i
            LOADL       8           ; load literal value 8
            CALL        mod         ; modulus of the two top entries on the stack
```

```
            LOAD(1)     3[SB]          ; load the variable help
            CALL        add            ; add up the entries on the top of the stack
            STORE(1)    3[SB]          ; store in variable help
            LOAD(1)     3[SB]          ; load the variable help on the stack
            POP(0)      1              ; pop the resulting value of exprdo cleanup
            LOAD(1)     0[SB]          ; load the variable i
            LOADL       1              ; load literal value 1
            CALL        sub            ; subtract the top entries of the stack from eachother
            STORE(1)    0[SB]          ; store in variable i
            LOAD(1)     0[SB]          ; load the variable i on the stack
            POP(0)      1              ; pop the resulting value of exprdo cleanup
            POP(0)      0              ; pop 0 local variables on closing a scope
            JUMP        WHILE0[CB]     ; jump to the while
ENDWHILE0:  POP(0)      0              ; pop 0 local variables on closing a scope
            LOAD(1)     3[SB]          ; load the variable help
            LOADA       -1[ST]         ; load address of int on top of stack
            LOADI(1)                   ; get the int on the stack, again, for printing
            CALL        putint         ; print the integer value of
            CALL        puteol         ; put a newline
            POP(1)      1              ; pop 1 local variables on closing a scope
ENDIF12:    POP(1)      0              ; pop 0 local variables on closing a scope
            STORE(1)    0[SB]          ; store in variable i
            LOAD(1)     0[SB]          ; load the variable i on the stack
            POP(0)      1              ; pop the resulting value of dsb cleanup start
            PUSH        1              ; push variable d on the stack
            LOADL       0              ; load init value of d
            STORE(1)    3[SB]          ; store in variable d
            LOAD(1)     3[SB]          ; load the variable d
            CALL        not            ; negate the statement
            JUMPIF(0)   ELSE13[CB]     ; jump to the else clause
            LOAD(1)     3[SB]          ; load the variable d
            CALL        not            ; negate the statement
            STORE(1)    3[SB]          ; store in variable d
            LOAD(1)     3[SB]          ; load the variable d on the stack
            POP(1)      0              ; pop 0 local variables on closing a scope
            JUMP        ENDIF13[CB]    ; jump past the else clause
ELSE13:     JUMP        ENDIF13[CB]    ; jump to the end
ENDIF13:    POP(1)      0              ; pop 0 local variables on closing a scope
            POP(0)      1              ; pop the resulting value of dsb cleanup start
            LOAD(1)     3[SB]          ; load the variable d
            LOADA       -1[ST]         ; load address of int on top of stack
            LOADI(1)                   ; get the int on the stack, again, for printing
            CALL        putint         ; print the integer value of
            CALL        puteol         ; put a newline
            POP(0)      1              ; pop the resulting value of dsb cleanup start
            LOADL       5              ; load literal value 5
            LOADL       6              ; load literal value 6
            CALL        lt             ; less than statement
            LOAD(1)     0[SB]          ; load the variable i
            LOAD(1)     2[SB]          ; load the constant b
            CALL        gt             ; greater than statement
            CALL        and            ; and the values on the top of the stack with eachother
            STORE(1)    3[SB]          ; store in variable d
            LOAD(1)     3[SB]          ; load the variable d on the stack
            POP(0)      1              ; pop the resulting value of dsb cleanup start
            LOAD(1)     3[SB]          ; load the variable d
            LOADA       -1[ST]         ; load address of int on top of stack
            LOADI(1)                   ; get the int on the stack, again, for printing
            CALL        putint         ; print the integer value of
            CALL        puteol         ; put a newline
            POP(0)      1              ; pop the resulting value of dsb cleanup start
            POP(0)      4              ; free 4 variables at end of program
            HALT                       ; ends the program
```

test_one.era.jasm:

```
.source Test_one.jasm
.class public Test_one
.super java/lang/Object

; standard class initializer
.method public <init>()V
       aload_0
       invokenonvirtual java/lang/Object/<init>()V
       return
.end method

.method public static main([Ljava/lang/String;)V
       .limit locals 7
       .limit stack 61


       ; 'start' of generated code

       ldc 0
       istore 0
       ldc 121
       istore 1

       ; loading of a variable/constant
       iload 0
       dup

       ; push reference to system.out on stack
       getstatic java/lang/System/out Ljava/io/PrintStream;
       swap

       ; print the output
       invokevirtual java/io/PrintStream/println(I)V


       ; loading of variable/constant character
       iload 1
       dup
       i2c
       invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

       ; push reference to system.out on stack
       getstatic java/lang/System/out Ljava/io/PrintStream;
       swap

       ; print the output
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


       ; pop 2 values from the stack for cleaning up print
       pop
       pop
       ldc 0
       istore 2
       ldc 5
       istore 2
       iload 2

       ; pop 1 value from the stack for cleaning up indent dsb cleanup
       pop

       ; loading of a variable/constant
       iload 2
       ldc 2
```

```
        isub
        istore 2
        iload 2
        istore 0
        iload 0

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop

        ; loading of a variable/constant
        iload 0
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V

        ldc 2
        iadd
        istore 0
        iload 0

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 10
        istore 3

        ; loading of a variable/constant
        iload 0

        ; loading of a variable/constant
        iload 3
        ldc 6
        isub

        ; less-than statement
        if_icmplt LabelLT0
        iconst_0
        goto LabelLT1

LabelLT0:
        iconst_1

LabelLT1:
        ifeq ELSE11

        ; loading of variable/constant character
        iload 1
        dup
        i2c
        invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

        goto ENDIF11

ELSE11:
        ldc 0
        istore 4
```

```
        ldc 10
        istore 4
        iload 4

        ; pop 1 value from the stack for cleaning up exprelse cleanup
        pop
        ldc 100

        ; loading of a variable/constant
        iload 4
        iadd
        istore 0
        iload 0


ENDIF11:
        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop

        ; loading of a variable/constant
        iload 0
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V


        ; loading of variable/constant character
        iload 1
        dup
        i2c
        invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


        ; pop 2 values from the stack for cleaning up print
        pop
        pop

        ; loading of a variable/constant
        iload 3
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V


        ; pop 1 value from the stack for cleaning up print
        pop

        ; loading of a variable/constant
        iload 3
```

```
        ; loading of a variable/constant
        iload 0

        ; greater-than statement
        if_icmpgt LabelGT2
        iconst_0
        goto LabelGT3

LabelGT2:
        iconst_1

LabelGT3:
        ifeq ELSE12

        ; loading of a variable/constant
        iload 3
        ldc 2
        isub
        istore 0
        iload 0
        goto ENDIF12

ELSE12:
        ldc 0
        istore 5


WHILE4:
        ; loading of a variable/constant
        iload 0
        ldc 100

        ; greater-than statement
        if_icmpgt LabelGT5
        iconst_0
        goto LabelGT6

LabelGT5:
        iconst_1

LabelGT6:
        ifeq ENDWHILE4

        ; loading of a variable/constant
        iload 0
        ldc 8
        irem

        ; loading of a variable/constant
        iload 5
        iadd
        istore 5
        iload 5

        ; pop 1 value from the stack for cleaning up exprdo cleanup
        pop

        ; loading of a variable/constant
        iload 0
        ldc 1
        isub
        istore 0
        iload 0

        ; pop 1 value from the stack for cleaning up exprdo cleanup
        pop
        goto WHILE4
```

```
ENDWHILE4:
        ; loading of a variable/constant
        iload 5
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V


ENDIF12:
        istore 0
        iload 0

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 0
        istore 6

        ; loading of a variable/constant
        iload 6

        ; processing the NOT operation
        iconst_1
        ixor
        ifeq ELSE13

        ; loading of a variable/constant
        iload 6

        ; processing the NOT operation
        iconst_1
        ixor
        istore 6
        iload 6
        goto ENDIF13

ELSE13:
        ldc 0
        goto ENDIF13


ENDIF13:
        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop

        ; loading of a variable/constant
        iload 6
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V


        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 5
        ldc 6
```

```
        ; less-than statement
        if_icmplt LabelLT7
        iconst_0
        goto LabelLT8

LabelLT7:
        iconst_1

LabelLT8:

        ; loading of a variable/constant
        iload 0

        ; loading of a variable/constant
        iload 3

        ; greater-than statement
        if_icmpgt LabelGT9
        iconst_0
        goto LabelGT10

LabelGT9:
        iconst_1

LabelGT10:
        iand
        istore 6
        iload 6

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop

        ; loading of a variable/constant
        iload 6
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V


        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop


        ; 'end' of generated code
        return
.end method
```

Input test_one.era.tam:

Geen input benodigd.

Output test_one.era.tam:

```
0
y
3
110
y
```

```
10
39
1
1
```

test_two.era

```
// _____________________________________________

// DECLARATIONS
// Constants
const cc := 'p';
const ic := 5;
const bc := true;

// Variables
var cv: char;
var iv: int;
var bv: bool;

// Use assignments in constant declaration
const ic2 := 3+ic*2;
const bc2 := !bc || iv < 2;

// _____________________________________________

// ASSIGNMENTS
cv := 'r';
iv := 3;
bv := false;

// Multiple assignment
var iv2: int;
iv2 := 3;
iv := iv2 := 6;

// Expressions in assignments
iv := 2+3*ic-iv;
bv := 10>iv && bc;

// Print in assignment
cv := print('z');
iv := print(iv2)+1;
bv := !print(true);

// _____________________________________________

// EXPRESSIONS
// Unary symbols combined with +, -, *, /, %
print(+16+2*-8/2+19%3);

// Less than , greater than, equal to, not equal to
5<3 && iv == iv2 || iv >= 4 || 8<=iv2 && iv2 != 10 && 5>iv;

// If-expression
if 5>iv2 && 3<2 || cv == 'z'; then iv := 18; 8+8; else iv := 3; fi;

// If-expression without else part
if 3<8; then bv := 5==iv2; fi;

iv := 0;
// While-expression
while iv < 10; do iv := iv + 1; iv2 := 15/iv + iv2; od;
```

```
bv := true;
// Assign if-statement to variable and use nested if- and while-expressions
iv := if bv; then
       while iv > 0 && bv; do
              iv := iv-1;
              if iv%2 == 0; then
                     iv2 := 5+iv;
              else
                     if 8-iv > 0; then
                            iv2 := 10;
                     else
                            13-iv2;
                     fi;
              fi;
       od;
       8*iv%7;
else
       iv2 := 6;
fi;

// Decl_stat_blocks
iv2 := {var temp: int; {temp:=16; iv:=temp%9-1;}; iv + 3;};

// ____________________________________________

// READ
print('-',1);
read(cv);
print(cv);
print('-',2);
read(iv);
print('-',3,4);
read(bv, iv);

// Read in assignment
print('-',5);
iv := read(iv2)-1;

// ____________________________________________

// PRINT
print(cv, iv, bv);
print(iv:=15-3*-2+7%4);
print('d','o','n','e');
```

test_two.era.tasm:

```
           PUSH        1             ; push constant cc on the stack
           LOADL       112           ; load literal value 112 representing 'p'
           STORE(1)    0[SB]         ; store in variable cc
           PUSH        1             ; push constant ic on the stack
           LOADL       5             ; load literal value 5
           STORE(1)    1[SB]         ; store in variable ic
           PUSH        1             ; push constant bc on the stack
           LOADL       1             ; load literal value 1
           STORE(1)    2[SB]         ; store in variable bc
           LOAD(1)     0[SB]         ; load the constant cc
           LOADA       -1[ST]        ; load address of char on top of stack
           LOADI(1)                  ; get the char on the stack, again, for printing
           CALL        put           ; print the character
           CALL        puteol        ; put a newline
           LOAD(1)     1[SB]         ; load the constant ic
           LOADA       -1[ST]        ; load address of int on top of stack
           LOADI(1)                  ; get the int on the stack, again, for printing
           CALL        putint        ; print the integer value of
           CALL        puteol        ; put a newline
           POP(0)      2             ; pop the resulting value of print
           LOAD(1)     2[SB]         ; load the constant bc
           LOADA       -1[ST]        ; load address of int on top of stack
           LOADI(1)                  ; get the int on the stack, again, for printing
```

```
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
POP(0)       1              ; pop the resulting value of print
PUSH         1              ; push variable cv on the stack
LOADL        97             ; load init value of cv
STORE(1)     3[SB]          ; store in variable cv
PUSH         1              ; push variable iv on the stack
LOADL        0              ; load init value of iv
STORE(1)     4[SB]          ; store in variable iv
PUSH         1              ; push variable bv on the stack
LOADL        0              ; load init value of bv
STORE(1)     5[SB]          ; store in variable bv
LOAD(1)      3[SB]          ; load the variable cv
LOADA        -1[ST]         ; load address of char on top of stack
LOADI(1)                    ; get the char on the stack, again, for printing
CALL         put            ; print the character
CALL         puteol         ; put a newline
LOAD(1)      4[SB]          ; load the variable iv
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
POP(0)       2              ; pop the resulting value of print
LOAD(1)      5[SB]          ; load the variable bv
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
POP(0)       1              ; pop the resulting value of print
PUSH         1              ; push constant ic2 on the stack
LOADL        3              ; load literal value 3
LOAD(1)      1[SB]          ; load the constant ic
LOADL        2              ; load literal value 2
CALL         mult           ; multiplication of the top entries on the stack
CALL         add            ; add up the entries on the top of the stack
STORE(1)     6[SB]          ; store in variable ic2
PUSH         1              ; push constant bc2 on the stack
LOAD(1)      2[SB]          ; load the constant bc
CALL         not            ; negate the statement
LOAD(1)      4[SB]          ; load the variable iv
LOADL        2              ; load literal value 2
CALL         lt             ; less than statement
CALL         or             ; or the values on the top of the stack with eachother
STORE(1)     7[SB]          ; store in variable bc2
LOAD(1)      6[SB]          ; load the constant ic2
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
LOAD(1)      7[SB]          ; load the constant bc2
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
POP(0)       2              ; pop the resulting value of print
LOADL        114            ; load literal value 114 representing 'r'
STORE(1)     3[SB]          ; store in variable cv
LOAD(1)      3[SB]          ; load the variable cv on the stack
POP(0)       1              ; pop the resulting value of dsb cleanup start
LOADL        3              ; load literal value 3
STORE(1)     4[SB]          ; store in variable iv
LOAD(1)      4[SB]          ; load the variable iv on the stack
POP(0)       1              ; pop the resulting value of dsb cleanup start
LOADL        0              ; load literal value 0
STORE(1)     5[SB]          ; store in variable bv
LOAD(1)      5[SB]          ; load the variable bv on the stack
POP(0)       1              ; pop the resulting value of dsb cleanup start
LOAD(1)      3[SB]          ; load the variable cv
LOADA        -1[ST]         ; load address of char on top of stack
LOADI(1)                    ; get the char on the stack, again, for printing
CALL         put            ; print the character
CALL         puteol         ; put a newline
LOAD(1)      4[SB]          ; load the variable iv
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
```

```
POP(0)      2             ; pop the resulting value of print
LOAD(1)     5[SB]         ; load the variable bv
LOADA       -1[ST]        ; load address of int on top of stack
LOADI(1)                  ; get the int on the stack, again, for printing
CALL        putint        ; print the integer value of
CALL        puteol        ; put a newline
POP(0)      1             ; pop the resulting value of print
PUSH        1             ; push variable iv2 on the stack
LOADL       0             ; load init value of iv2
STORE(1)    8[SB]         ; store in variable iv2
LOADL       6             ; load literal value 6
STORE(1)    8[SB]         ; store in variable iv2
LOAD(1)     8[SB]         ; load the variable iv2 on the stack
STORE(1)    4[SB]         ; store in variable iv
LOAD(1)     4[SB]         ; load the variable iv on the stack
POP(0)      1             ; pop the resulting value of dsb cleanup start
LOAD(1)     8[SB]         ; load the variable iv2
LOADA       -1[ST]        ; load address of int on top of stack
LOADI(1)                  ; get the int on the stack, again, for printing
CALL        putint        ; print the integer value of
CALL        puteol        ; put a newline
LOAD(1)     4[SB]         ; load the variable iv
LOADA       -1[ST]        ; load address of int on top of stack
LOADI(1)                  ; get the int on the stack, again, for printing
CALL        putint        ; print the integer value of
CALL        puteol        ; put a newline
POP(0)      2             ; pop the resulting value of print
LOADL       2             ; load literal value 2
LOADL       3             ; load literal value 3
LOAD(1)     1[SB]         ; load the constant ic
CALL        mult          ; multiplication of the top entries on the stack
CALL        add           ; add up the entries on the top of the stack
LOAD(1)     4[SB]         ; load the variable iv
CALL        sub           ; subtract the top entries of the stack from eachother
STORE(1)    4[SB]         ; store in variable iv
LOAD(1)     4[SB]         ; load the variable iv on the stack
POP(0)      1             ; pop the resulting value of dsb cleanup start
LOADL       10            ; load literal value 10
LOAD(1)     4[SB]         ; load the variable iv
CALL        gt            ; greater than statement
LOAD(1)     2[SB]         ; load the constant bc
CALL        and           ; and the values on the top of the stack with eachother
STORE(1)    5[SB]         ; store in variable bv
LOAD(1)     5[SB]         ; load the variable bv on the stack
POP(0)      1             ; pop the resulting value of dsb cleanup start
LOAD(1)     4[SB]         ; load the variable iv
LOADA       -1[ST]        ; load address of int on top of stack
LOADI(1)                  ; get the int on the stack, again, for printing
CALL        putint        ; print the integer value of
CALL        puteol        ; put a newline
LOAD(1)     5[SB]         ; load the variable bv
LOADA       -1[ST]        ; load address of int on top of stack
LOADI(1)                  ; get the int on the stack, again, for printing
CALL        putint        ; print the integer value of
CALL        puteol        ; put a newline
POP(0)      2             ; pop the resulting value of print
LOADL       122           ; load literal value 122 representing 'z'
LOADA       -1[ST]        ; load address of char on top of stack
LOADI(1)                  ; get the char on the stack, again, for printing
CALL        put           ; print the character
CALL        puteol        ; put a newline
STORE(1)    3[SB]         ; store in variable cv
LOAD(1)     3[SB]         ; load the variable cv on the stack
POP(0)      1             ; pop the resulting value of dsb cleanup start
LOAD(1)     8[SB]         ; load the variable iv2
LOADA       -1[ST]        ; load address of int on top of stack
LOADI(1)                  ; get the int on the stack, again, for printing
CALL        putint        ; print the integer value of
CALL        puteol        ; put a newline
LOADL       1             ; load literal value 1
CALL        add           ; add up the entries on the top of the stack
STORE(1)    4[SB]         ; store in variable iv
LOAD(1)     4[SB]         ; load the variable iv on the stack
POP(0)      1             ; pop the resulting value of dsb cleanup start
LOADL       1             ; load literal value 1
LOADA       -1[ST]        ; load address of int on top of stack
LOADI(1)                  ; get the int on the stack, again, for printing
```

```
CALL        putint          ; print the integer value of
CALL        puteol          ; put a newline
CALL        not             ; negate the statement
STORE(1)    5[SB]           ; store in variable bv
LOAD(1)     5[SB]           ; load the variable bv on the stack
POP(0)      1               ; pop the resulting value of dsb cleanup start
LOAD(1)     3[SB]           ; load the variable cv
LOADA       -1[ST]          ; load address of char on top of stack
LOADI(1)                    ; get the char on the stack, again, for printing
CALL        put             ; print the character
CALL        puteol          ; put a newline
LOAD(1)     4[SB]           ; load the variable iv
LOADA       -1[ST]          ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL        putint          ; print the integer value of
CALL        puteol          ; put a newline
POP(0)      2               ; pop the resulting value of print
LOAD(1)     5[SB]           ; load the variable bv
LOADA       -1[ST]          ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL        putint          ; print the integer value of
CALL        puteol          ; put a newline
POP(0)      1               ; pop the resulting value of print
LOADL       16              ; load literal value 16
LOADL       2               ; load literal value 2
LOADL       8               ; load literal value 8
LOADL       -1              ; load literal value -1
CALL        mult            ; multiply top of the stack with -1
CALL        mult            ; multiplication of the top entries on the stack
LOADL       2               ; load literal value 2
CALL        div             ; division of the top entries on the stack
CALL        add             ; add up the entries on the top of the stack
LOADL       19              ; load literal value 19
LOADL       3               ; load literal value 3
CALL        mod             ; modulus of the two top entries on the stack
CALL        add             ; add up the entries on the top of the stack
LOADA       -1[ST]          ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL        putint          ; print the integer value of
CALL        puteol          ; put a newline
POP(0)      1               ; pop the resulting value of dsb cleanup start
LOADL       5               ; load literal value 5
LOADL       3               ; load literal value 3
CALL        lt              ; less than statement
LOAD(1)     4[SB]           ; load the variable iv
LOAD(1)     8[SB]           ; load the variable iv2
LOADL       1               ; load literal value 1
CALL        eq              ; equals statement
CALL        and             ; and the values on the top of the stack with eachother
LOAD(1)     4[SB]           ; load the variable iv
LOADL       4               ; load literal value 4
CALL        ge              ; greater or equal statement
CALL        or              ; or the values on the top of the stack with eachother
LOADL       8               ; load literal value 8
LOAD(1)     8[SB]           ; load the variable iv2
CALL        le              ; less or equal statement
LOAD(1)     8[SB]           ; load the variable iv2
LOADL       10              ; load literal value 10
LOADL       1               ; load literal value 1
CALL        ne              ; not equals statement
CALL        and             ; and the values on the top of the stack with eachother
LOADL       5               ; load literal value 5
LOAD(1)     4[SB]           ; load the variable iv
CALL        gt              ; greater than statement
CALL        and             ; and the values on the top of the stack with eachother
CALL        or              ; or the values on the top of the stack with eachother
LOADA       -1[ST]          ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL        putint          ; print the integer value of
CALL        puteol          ; put a newline
POP(0)      1               ; pop the resulting value of dsb cleanup start
LOADL       5               ; load literal value 5
LOAD(1)     8[SB]           ; load the variable iv2
CALL        gt              ; greater than statement
LOADL       3               ; load literal value 3
LOADL       2               ; load literal value 2
CALL        lt              ; less than statement
```

```
            CALL        and             ; and the values on the top of the stack with eachother
            LOAD(1)     3[SB]           ; load the variable cv
            LOADL       122             ; load literal value 122 representing 'z'
            LOADL       1               ; load literal value 1
            CALL        eq              ; equals statement
            CALL        or              ; or the values on the top of the stack with eachother
            JUMPIF(0)   ELSE11[CB]      ; jump to the else clause
            LOADL       18              ; load literal value 18
            STORE(1)    4[SB]           ; store in variable iv
            LOAD(1)     4[SB]           ; load the variable iv on the stack
            POP(0)      1               ; pop the resulting value of exprthen cleanup
            LOADL       8               ; load literal value 8
            LOADL       8               ; load literal value 8
            CALL        add             ; add up the entries on the top of the stack
            POP(1)      0               ; pop 0 local variables on closing a scope
            JUMP        ENDIF11[CB]     ; jump past the else clause
ELSE11:     LOADL       3               ; load literal value 3
            STORE(1)    4[SB]           ; store in variable iv
            LOAD(1)     4[SB]           ; load the variable iv on the stack
            POP(1)      0               ; pop 0 local variables on closing a scope
ENDIF11:    POP(1)      0               ; pop 0 local variables on closing a scope
            POP(0)      1               ; pop the resulting value of dsb cleanup start
            LOAD(1)     3[SB]           ; load the variable cv
            LOADA       -1[ST]          ; load address of char on top of stack
            LOADI(1)                    ; get the char on the stack, again, for printing
            CALL        put             ; print the character
            CALL        puteol          ; put a newline
            LOAD(1)     4[SB]           ; load the variable iv
            LOADA       -1[ST]          ; load address of int on top of stack
            LOADI(1)                    ; get the int on the stack, again, for printing
            CALL        putint          ; print the integer value of
            CALL        puteol          ; put a newline
            POP(0)      2               ; pop the resulting value of print
            LOADL       3               ; load literal value 3
            LOADL       8               ; load literal value 8
            CALL        lt              ; less than statement
            JUMPIF(0)   ELSE12[CB]      ; jump to the else clause
            LOADL       5               ; load literal value 5
            LOAD(1)     8[SB]           ; load the variable iv2
            LOADL       1               ; load literal value 1
            CALL        eq              ; equals statement
            STORE(1)    5[SB]           ; store in variable bv
            LOAD(1)     5[SB]           ; load the variable bv on the stack
            POP(1)      0               ; pop 0 local variables on closing a scope
            JUMP        ENDIF12[CB]     ; jump past the else clause
ELSE12:     JUMP        ENDIF12[CB]     ; jump to the end
ENDIF12:    POP(1)      0               ; pop 0 local variables on closing a scope
            POP(0)      1               ; pop the resulting value of dsb cleanup start
            LOAD(1)     5[SB]           ; load the variable bv
            LOADA       -1[ST]          ; load address of int on top of stack
            LOADI(1)                    ; get the int on the stack, again, for printing
            CALL        putint          ; print the integer value of
            CALL        puteol          ; put a newline
            POP(0)      1               ; pop the resulting value of dsb cleanup start
            LOADL       0               ; load literal value 0
            STORE(1)    4[SB]           ; store in variable iv
            LOAD(1)     4[SB]           ; load the variable iv on the stack
            POP(0)      1               ; pop the resulting value of dsb cleanup start
WHILE0:     LOAD(1)     4[SB]           ; load the variable iv
            LOADL       10              ; load literal value 10
            CALL        lt              ; less than statement
            JUMPIF(0)   ENDWHILE0[CB]   ; jump past the while body
            LOAD(1)     4[SB]           ; load the variable iv
            LOADL       1               ; load literal value 1
            CALL        add             ; add up the entries on the top of the stack
            STORE(1)    4[SB]           ; store in variable iv
            LOAD(1)     4[SB]           ; load the variable iv on the stack
            POP(0)      1               ; pop the resulting value of exprdo cleanup
            LOADL       15              ; load literal value 15
            LOAD(1)     4[SB]           ; load the variable iv
            CALL        div             ; division of the top entries on the stack
            LOAD(1)     8[SB]           ; load the variable iv2
            CALL        add             ; add up the entries on the top of the stack
            STORE(1)    8[SB]           ; store in variable iv2
            LOAD(1)     8[SB]           ; load the variable iv2 on the stack
            POP(0)      1               ; pop the resulting value of exprdo cleanup
            POP(0)      0               ; pop 0 local variables on closing a scope
```

```
            JUMP        WHILE0[CB]     ; jump to the while
ENDWHILE0:  POP(0)      0              ; pop 0 local variables on closing a scope
            LOAD(1)     4[SB]          ; load the variable iv
            LOADA       -1[ST]         ; load address of int on top of stack
            LOADI(1)                   ; get the int on the stack, again, for printing
            CALL        putint         ; print the integer value of
            CALL        puteol         ; put a newline
            LOAD(1)     8[SB]          ; load the variable iv2
            LOADA       -1[ST]         ; load address of int on top of stack
            LOADI(1)                   ; get the int on the stack, again, for printing
            CALL        putint         ; print the integer value of
            CALL        puteol         ; put a newline
            POP(0)      2              ; pop the resulting value of print
            LOADL       1              ; load literal value 1
            STORE(1)    5[SB]          ; store in variable bv
            LOAD(1)     5[SB]          ; load the variable bv on the stack
            POP(0)      1              ; pop the resulting value of dsb cleanup start
            LOAD(1)     5[SB]          ; load the variable bv
            JUMPIF(0)   ELSE13[CB]     ; jump to the else clause
WHILE1:     LOAD(1)     4[SB]          ; load the variable iv
            LOADL       0              ; load literal value 0
            CALL        gt             ; greater than statement
            LOAD(1)     5[SB]          ; load the variable bv
            CALL        and            ; and the values on the top of the stack with eachother
            JUMPIF(0)   ENDWHILE1[CB]  ; jump past the while body
            LOAD(1)     4[SB]          ; load the variable iv
            LOADL       1              ; load literal value 1
            CALL        sub            ; subtract the top entries of the stack from eachother
            STORE(1)    4[SB]          ; store in variable iv
            LOAD(1)     4[SB]          ; load the variable iv on the stack
            POP(0)      1              ; pop the resulting value of exprdo cleanup
            LOAD(1)     4[SB]          ; load the variable iv
            LOADL       2              ; load literal value 2
            CALL        mod            ; modulus of the two top entries on the stack
            LOADL       0              ; load literal value 0
            LOADL       1              ; load literal value 1
            CALL        eq             ; equals statement
            JUMPIF(0)   ELSE51[CB]     ; jump to the else clause
            LOADL       5              ; load literal value 5
            LOAD(1)     4[SB]          ; load the variable iv
            CALL        add            ; add up the entries on the top of the stack
            STORE(1)    8[SB]          ; store in variable iv2
            LOAD(1)     8[SB]          ; load the variable iv2 on the stack
            POP(1)      0              ; pop 0 local variables on closing a scope
            JUMP        ENDIF51[CB]    ; jump past the else clause
ELSE51:     LOADL       8              ; load literal value 8
            LOAD(1)     4[SB]          ; load the variable iv
            CALL        sub            ; subtract the top entries of the stack from eachother
            LOADL       0              ; load literal value 0
            CALL        gt             ; greater than statement
            JUMPIF(0)   ELSE71[CB]     ; jump to the else clause
            LOADL       10             ; load literal value 10
            STORE(1)    8[SB]          ; store in variable iv2
            LOAD(1)     8[SB]          ; load the variable iv2 on the stack
            POP(1)      0              ; pop 0 local variables on closing a scope
            JUMP        ENDIF71[CB]    ; jump past the else clause
ELSE71:     LOADL       13             ; load literal value 13
            LOAD(1)     8[SB]          ; load the variable iv2
            CALL        sub            ; subtract the top entries of the stack from eachother
            POP(1)      0              ; pop 0 local variables on closing a scope
ENDIF71:    POP(1)      0              ; pop 0 local variables on closing a scope
            POP(1)      0              ; pop 0 local variables on closing a scope
ENDIF51:    POP(1)      0              ; pop 0 local variables on closing a scope
            POP(0)      1              ; pop the resulting value of exprdo cleanup
            POP(0)      0              ; pop 0 local variables on closing a scope
            JUMP        WHILE1[CB]     ; jump to the while
ENDWHILE1:  POP(0)      0              ; pop 0 local variables on closing a scope
            LOADL       8              ; load literal value 8
            LOAD(1)     4[SB]          ; load the variable iv
            CALL        mult           ; multiplication of the top entries on the stack
            LOADL       7              ; load literal value 7
            CALL        mod            ; modulus of the two top entries on the stack
            POP(1)      0              ; pop 0 local variables on closing a scope
            JUMP        ENDIF13[CB]    ; jump past the else clause
ELSE13:     LOADL       6              ; load literal value 6
            STORE(1)    8[SB]          ; store in variable iv2
            LOAD(1)     8[SB]          ; load the variable iv2 on the stack
```

```
            POP(1)      0           ; pop 0 local variables on closing a scope
ENDIF13:    POP(1)      0           ; pop 0 local variables on closing a scope
            STORE(1)    4[SB]       ; store in variable iv
            LOAD(1)     4[SB]       ; load the variable iv on the stack
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOAD(1)     4[SB]       ; load the variable iv
            LOADA       -1[ST]      ; load address of int on top of stack
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
            LOAD(1)     8[SB]       ; load the variable iv2
            LOADA       -1[ST]      ; load address of int on top of stack
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
            POP(0)      2           ; pop the resulting value of print
            PUSH        1           ; push variable temp on the stack
            LOADL       0           ; load init value of temp
            STORE(1)    9[SB]       ; store in variable temp
            LOADL       16          ; load literal value 16
            STORE(1)    9[SB]       ; store in variable temp
            LOAD(1)     9[SB]       ; load the variable temp on the stack
            POP(0)      1           ; pop the resulting value of indent dsb cleanup
            LOAD(1)     9[SB]       ; load the variable temp
            LOADL       9           ; load literal value 9
            CALL        mod         ; modulus of the two top entries on the stack
            LOADL       1           ; load literal value 1
            CALL        sub         ; subtract the top entries of the stack from eachother
            STORE(1)    4[SB]       ; store in variable iv
            LOAD(1)     4[SB]       ; load the variable iv on the stack
            POP(1)      0           ; pop 0 local variables on closing a scope
            POP(0)      1           ; pop the resulting value of indent dsb cleanup
            LOAD(1)     4[SB]       ; load the variable iv
            LOADL       3           ; load literal value 3
            CALL        add         ; add up the entries on the top of the stack
            POP(1)      1           ; pop 1 local variables on closing a scope
            STORE(1)    8[SB]       ; store in variable iv2
            LOAD(1)     8[SB]       ; load the variable iv2 on the stack
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOAD(1)     8[SB]       ; load the variable iv2
            LOADA       -1[ST]      ; load address of int on top of stack
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOADA       3[SB]       ; load the address of cv
            CALL        get         ; get a character value for cv
            LOAD(1)     3[SB]       ; load the variable cv
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOAD(1)     3[SB]       ; load the variable cv
            LOADA       -1[ST]      ; load address of char on top of stack
            LOADI(1)                ; get the char on the stack, again, for printing
            CALL        put         ; print the character
            CALL        puteol      ; put a newline
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOADA       4[SB]       ; load the address of iv
            CALL        getint      ; read a numeric value for iv
            LOAD(1)     4[SB]       ; load the variable iv
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOAD(1)     4[SB]       ; load the variable iv
            LOADA       -1[ST]      ; load address of int on top of stack
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
            POP(0)      1           ; pop the resulting value of dsb cleanup start
            LOADA       5[SB]       ; load the address of bv
            CALL        getint      ; read a numeric value for bv
            LOAD(1)     5[SB]       ; load the variable bv
            LOADA       4[SB]       ; load the address of iv
            CALL        getint      ; read a numeric value for iv
            LOAD(1)     4[SB]       ; load the variable iv
            POP(0)      2           ; pop the resulting value of read
            LOAD(1)     5[SB]       ; load the variable bv
            LOADA       -1[ST]      ; load address of int on top of stack
            LOADI(1)                ; get the int on the stack, again, for printing
            CALL        putint      ; print the integer value of
            CALL        puteol      ; put a newline
```

```
LOAD(1)      4[SB]          ; load the variable iv
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
POP(0)       2              ; pop the resulting value of print
LOADA        8[SB]          ; load the address of iv2
CALL         getint         ; read a numeric value for iv2
LOAD(1)      8[SB]          ; load the variable iv2
LOADL        1              ; load literal value 1
CALL         sub            ; subtract the top entries of the stack from eachother
STORE(1)     4[SB]          ; store in variable iv
LOAD(1)      4[SB]          ; load the variable iv on the stack
POP(0)       1              ; pop the resulting value of dsb cleanup start
LOAD(1)      4[SB]          ; load the variable iv
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
LOAD(1)      8[SB]          ; load the variable iv2
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
POP(0)       2              ; pop the resulting value of print
LOADL        15             ; load literal value 15
LOADL        3              ; load literal value 3
LOADL        2              ; load literal value 2
LOADL        -1             ; load literal value -1
CALL         mult           ; multiply top of the stack with -1
CALL         mult           ; multiplication of the top entries on the stack
CALL         sub            ; subtract the top entries of the stack from eachother
LOADL        7              ; load literal value 7
LOADL        4              ; load literal value 4
CALL         mod            ; modulus of the two top entries on the stack
CALL         add            ; add up the entries on the top of the stack
STORE(1)     4[SB]          ; store in variable iv
LOAD(1)      4[SB]          ; load the variable iv on the stack
LOADA        -1[ST]         ; load address of int on top of stack
LOADI(1)                    ; get the int on the stack, again, for printing
CALL         putint         ; print the integer value of
CALL         puteol         ; put a newline
POP(0)       1              ; pop the resulting value of dsb cleanup start
LOADL        100            ; load literal value 100 representing 'd'
LOADA        -1[ST]         ; load address of char on top of stack
LOADI(1)                    ; get the char on the stack, again, for printing
CALL         put            ; print the character
CALL         puteol         ; put a newline
LOADL        111            ; load literal value 111 representing 'o'
LOADA        -1[ST]         ; load address of char on top of stack
LOADI(1)                    ; get the char on the stack, again, for printing
CALL         put            ; print the character
CALL         puteol         ; put a newline
POP(0)       2              ; pop the resulting value of print
LOADL        110            ; load literal value 110 representing 'n'
LOADA        -1[ST]         ; load address of char on top of stack
LOADI(1)                    ; get the char on the stack, again, for printing
CALL         put            ; print the character
CALL         puteol         ; put a newline
POP(0)       1              ; pop the resulting value of print
LOADL        101            ; load literal value 101 representing 'e'
LOADA        -1[ST]         ; load address of char on top of stack
LOADI(1)                    ; get the char on the stack, again, for printing
CALL         put            ; print the character
CALL         puteol         ; put a newline
POP(0)       1              ; pop the resulting value of print
POP(0)       9              ; free 9 variables at end of program
HALT                        ; ends the program
```

test_two.era.jasm:

```
.source Test_two.jasm
.class public Test_two
.super java/lang/Object

; standard class initializer
```

```
.method public <init>()V
       aload_0
       invokenonvirtual java/lang/Object/<init>()V
       return
.end method

.method public static main([Ljava/lang/String;)V
       .limit locals 10
       .limit stack 171


       ; 'start' of generated code

       ldc 112
       istore 0
       ldc 5
       istore 1
       ldc 1
       istore 2
       ldc 97
       istore 3
       ldc 0
       istore 4
       ldc 0
       istore 5
       ldc 3

       ; loading of a variable/constant
       iload 1
       ldc 2
       imul
       iadd
       istore 6

       ; loading of a variable/constant
       iload 2

       ; processing the NOT operation
       iconst_1
       ixor

       ; loading of a variable/constant
       iload 4
       ldc 2

       ; less-than statement
       if_icmplt LabelLT0
       iconst_0
       goto LabelLT1

LabelLT0:
       iconst_1

LabelLT1:
       ior
       istore 7
       ldc 114
       istore 3
       iload 3

       ; pop 1 value from the stack for cleaning up dsb cleanup start
       pop
       ldc 3
       istore 4
       iload 4

       ; pop 1 value from the stack for cleaning up dsb cleanup start
       pop
       ldc 0
       istore 5
       iload 5

       ; pop 1 value from the stack for cleaning up dsb cleanup start
       pop
       ldc 0
       istore 8
       ldc 3
```

```
        istore 8
        iload 8

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 6
        istore 8
        iload 8
        istore 4
        iload 4

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 2
        ldc 3

        ; loading of a variable/constant
        iload 1
        imul
        iadd

        ; loading of a variable/constant
        iload 4
        isub
        istore 4
        iload 4

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 10

        ; loading of a variable/constant
        iload 4

        ; greater-than statement
        if_icmpgt LabelGT2
        iconst_0
        goto LabelGT3

LabelGT2:
        iconst_1

LabelGT3:

        ; loading of a variable/constant
        iload 2
        iand
        istore 5
        iload 5

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 122
        dup
        i2c
        invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

        istore 3
        iload 3

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop

        ; loading of a variable/constant
        iload 8
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
```

```
        ; print the output
        invokevirtual java/io/PrintStream/println(I)V

        ldc 1
        iadd
        istore 4
        iload 4

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 1
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V


        ; processing the NOT operation
        iconst_1
        ixor
        istore 5
        iload 5

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 16
        ldc 2
        ldc 8
        ineg
        imul
        ldc 2
        idiv
        iadd
        ldc 19
        ldc 3
        irem
        iadd
        dup

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(I)V


        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 5
        ldc 3

        ; less-than statement
        if_icmplt LabelLT4
        iconst_0
        goto LabelLT5

LabelLT4:
        iconst_1

LabelLT5:

        ; loading of a variable/constant
        iload 4

        ; loading of a variable/constant
        iload 8

        ; equals statement
        if_icmpeq LabelEQ6
        iconst_0
        goto LabelEQ7
```

```
LabelEQ6:
        iconst_1

LabelEQ7:
        iand

        ; loading of a variable/constant
        iload 4
        ldc 4

        ; greater-equals statement
        if_icmpge LabelGE8
        iconst_0
        goto LabelGE9

LabelGE8:
        iconst_1

LabelGE9:
        ior
        ldc 8

        ; loading of a variable/constant
        iload 8

        ; less-equals statement
        if_icmple LabelLE10
        iconst_0
        goto LabelLE11

LabelLE10:
        iconst_1

LabelLE11:

        ; loading of a variable/constant
        iload 8
        ldc 10

        ; not-equals statement
        if_icmpne LabelNE12
        iconst_0
        goto LabelNE13

LabelNE12:
        iconst_1

LabelNE13:
        iand
        ldc 5

        ; loading of a variable/constant
        iload 4

        ; greater-than statement
        if_icmpgt LabelGT14
        iconst_0
        goto LabelGT15

LabelGT14:
        iconst_1

LabelGT15:
        iand
        ior

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 5

        ; loading of a variable/constant
        iload 8

        ; greater-than statement
        if_icmpgt LabelGT16
        iconst_0
```

```
        goto LabelGT17

LabelGT16:
        iconst_1

LabelGT17:
        ldc 3
        ldc 2

        ; less-than statement
        if_icmplt LabelLT18
        iconst_0
        goto LabelLT19

LabelLT18:
        iconst_1

LabelLT19:
        iand

        ; loading of variable/constant character
        iload 3
        ldc 122

        ; equals statement
        if_icmpeq LabelEQ20
        iconst_0
        goto LabelEQ21

LabelEQ20:
        iconst_1

LabelEQ21:
        ior
        ifeq ELSE11
        ldc 18
        istore 4
        iload 4
        ldc 8
        ldc 8
        iadd
        goto ENDIF11

ELSE11:
        ldc 3
        istore 4
        iload 4


ENDIF11:
        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 3
        ldc 8

        ; less-than statement
        if_icmplt LabelLT22
        iconst_0
        goto LabelLT23

LabelLT22:
        iconst_1

LabelLT23:
        ifeq ELSE12
        ldc 5

        ; loading of a variable/constant
        iload 8

        ; equals statement
        if_icmpeq LabelEQ24
        iconst_0
        goto LabelEQ25

LabelEQ24:
        iconst_1
```

```
LabelEQ25:
        istore 5
        iload 5
        goto ENDIF12

ELSE12:
        ldc 0
        goto ENDIF12


ENDIF12:
        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop
        ldc 0
        istore 4
        iload 4

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop


WHILE26:
        ; loading of a variable/constant
        iload 4
        ldc 10

        ; less-than statement
        if_icmplt LabelLT27
        iconst_0
        goto LabelLT28

LabelLT27:
        iconst_1

LabelLT28:
        ifeq ENDWHILE26

        ; loading of a variable/constant
        iload 4
        ldc 1
        iadd
        istore 4
        iload 4

        ; pop 1 value from the stack for cleaning up exprdo cleanup
        pop
        ldc 15

        ; loading of a variable/constant
        iload 4
        idiv

        ; loading of a variable/constant
        iload 8
        iadd
        istore 8
        iload 8

        ; pop 1 value from the stack for cleaning up exprdo cleanup
        pop
        goto WHILE26

ENDWHILE26:
        ldc 1
        istore 5
        iload 5

        ; pop 1 value from the stack for cleaning up dsb cleanup start
        pop

        ; loading of a variable/constant
        iload 5
        ifeq ELSE13


WHILE29:
```

```
        ; loading of a variable/constant
        iload 4
        ldc 0

        ; greater-than statement
        if_icmpgt LabelGT30
        iconst_0
        goto LabelGT31

LabelGT30:
        iconst_1

LabelGT31:

        ; loading of a variable/constant
        iload 5
        iand
        ifeq ENDWHILE29

        ; loading of a variable/constant
        iload 4
        ldc 1
        isub
        istore 4
        iload 4

        ; pop 1 value from the stack for cleaning up exprdo cleanup
        pop

        ; loading of a variable/constant
        iload 4
        ldc 2
        irem
        ldc 0

        ; equals statement
        if_icmpeq LabelEQ32
        iconst_0
        goto LabelEQ33

LabelEQ32:
        iconst_1

LabelEQ33:
        ifeq ELSE51
        ldc 5

        ; loading of a variable/constant
        iload 4
        iadd
        istore 8
        iload 8
        goto ENDIF51

ELSE51:
        ldc 8

        ; loading of a variable/constant
        iload 4
        isub
        ldc 0

        ; greater-than statement
        if_icmpgt LabelGT34
        iconst_0
        goto LabelGT35

LabelGT34:
        iconst_1

LabelGT35:
        ifeq ELSE71
        ldc 10
        istore 8
        iload 8
        goto ENDIF71
```

```
ELSE71:
       ldc 13

       ; loading of a variable/constant
       iload 8
       isub

ENDIF71:
       goto ENDIF51


ENDIF51:
       ; pop 1 value from the stack for cleaning up exprdo cleanup
       pop
       goto WHILE29


ENDWHILE29:
       ; pop 1 value from the stack for cleaning up exprthen cleanup
       pop
       ldc 8

       ; loading of a variable/constant
       iload 4
       imul
       ldc 7
       irem
       goto ENDIF13

ELSE13:
       ldc 6
       istore 8
       iload 8

ENDIF13:
       istore 4
       iload 4

       ; pop 1 value from the stack for cleaning up dsb cleanup start
       pop
       ldc 0
       istore 9
       ldc 16
       istore 9
       iload 9

       ; pop 1 value from the stack for cleaning up indent dsb cleanup
       pop

       ; loading of a variable/constant
       iload 9
       ldc 9
       irem
       ldc 1
       isub
       istore 4
       iload 4

       ; pop 1 value from the stack for cleaning up indent dsb cleanup
       pop

       ; loading of a variable/constant
       iload 4
       ldc 3
       iadd
       istore 8
       iload 8

       ; pop 1 value from the stack for cleaning up dsb cleanup start
       pop
       ldc 45
       dup
       i2c
       invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

       ; push reference to system.out on stack
       getstatic java/lang/System/out Ljava/io/PrintStream;
```

```
swap

; print the output
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

ldc 1
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 2 values from the stack for cleaning up print
pop
pop
getstatic java/lang/System/in Ljava/io/InputStream;
invokevirtual java/io/InputStream/read()I
istore 3
iload 3

; pop 1 value from the stack for cleaning up dsb cleanup start
pop

; loading of variable/constant character
iload 3
dup
i2c
invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


; pop 1 value from the stack for cleaning up dsb cleanup start
pop
ldc 45
dup
i2c
invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

ldc 2
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 2 values from the stack for cleaning up print
pop
pop
invokestatic Test_two.readint()I
istore 4
iload 4

; pop 1 value from the stack for cleaning up dsb cleanup start
pop
ldc 45
dup
```

```
i2c
invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

ldc 3
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 2 values from the stack for cleaning up print
pop
pop
ldc 4
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 1 value from the stack for cleaning up print
pop
invokestatic Test_two.readint()I
istore 5
iload 5
invokestatic Test_two.readint()I
istore 4
iload 4

; pop 2 values from the stack for cleaning up read
pop
pop
ldc 45
dup
i2c
invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

ldc 5
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 2 values from the stack for cleaning up print
pop
pop
invokestatic Test_two.readint()I
istore 8
iload 8
ldc 1
```

```
isub
istore 4
iload 4

; pop 1 value from the stack for cleaning up dsb cleanup start
pop

; loading of variable/constant character
iload 3
dup
i2c
invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


; loading of a variable/constant
iload 4
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 2 values from the stack for cleaning up print
pop
pop

; loading of a variable/constant
iload 5
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 1 value from the stack for cleaning up print
pop
ldc 15
ldc 3
ldc 2
ineg
imul
isub
ldc 7
ldc 4
irem
iadd
istore 4
iload 4
dup

; push reference to system.out on stack
getstatic java/lang/System/out Ljava/io/PrintStream;
swap

; print the output
invokevirtual java/io/PrintStream/println(I)V


; pop 1 value from the stack for cleaning up dsb cleanup start
pop
ldc 100
dup
```

```
        i2c
        invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

        ldc 111
        dup
        i2c
        invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


        ; pop 2 values from the stack for cleaning up print
        pop
        pop
        ldc 110
        dup
        i2c
        invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


        ; pop 1 value from the stack for cleaning up print
        pop
        ldc 101
        dup
        i2c
        invokestatic java/lang/String/valueOf(C)Ljava/lang/String;

        ; push reference to system.out on stack
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap

        ; print the output
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


        ; pop 1 value from the stack for cleaning up print
        pop


        ; 'end' of generated code
        return
.end method

; int readint() function
.method public static readint()I
        .limit locals 2
        .limit stack 10
        ldc 0
        istore 0

LabelRead0:
        getstatic java/lang/System/in Ljava/io/InputStream;
        invokevirtual java/io/InputStream/read()I
        istore 1
        iload 1
        ldc 10
        isub
        ifeq LabelRead1
```

```
        iload 1
        ldc 13
        isub
        ifeq LabelRead1
        iload 1
        ldc 32
        isub
        ifeq LabelRead1
        iload 1
        ldc 48
        isub
        ldc 10
        iload 0
        imul
        iadd
        istore 0
        goto LabelRead0

LabelRead1:
        ; local variable 0 holds the read integer
        iload 0
        ireturn

.end method
```

Input test_two.era:

character
integer 1
boolean
integer 2
integer 3

Output test_two.era:

```
p
5
1
a
0
0
13
1
r
3
0
6
6
11
0
z
6
1
z
7
0
9
1
z
18
0
10
46
```

```
0
5
9
ingevoerd character wordt geprint
eerste ingevoerde integer wordt geprint
ingevoerde boolean wordt geprint
tweede ingevoerde integer wordt geprint
derde ingevoerde integer verminderd met 1 wordt geprint
derde ingevoerde integer wordt geprint
24
d
o
n
e
```