# Simple and Tranquil Imperative Language

Jarno van Leeuwen
s1203134

Bas Veeling
s1084151

9 juli 2014

# Inhoudsopgave

# Hoofdstuk 1

# Introductie

In dit verslag staat de Stil programmeertaal beschreven. Stil – Simple and Tranquil Imperative Language – is ontwikkeld voor het vak Vertalerbouw als onderdeel van de bachelor Technische Informatica. De syntax, ontwikkelkeuzes, en implementatie van de parser, checker en code generator komen aan bod.

## 1.1 Beknopte beschrijving

De hoofdgedachte van Stil is dat de programmeertaal de ontwikkelaar zo min mogelijk in de weg moet zitten. Met sterk-verbose talen als Java en Objective-C lijkt men soms door de syntax de software niet meer te zien. Stil wil een tegengeluid bieden voor deze ontwikkeling, door de syntax zo kort en schoon mogelijk te houden.

De syntax van Stil is geïnspireerd door C, Java, Ruby en Python. De taal kent drie types: `char`, `int` en `bool`. Deze types kunnen zich als variabelen of constanten voortdoen. De syntax omvat de vereiste "basic expression syntax" met input/output, compound, logische en wiskundige expressies. Daarnaast bevat de syntax ook `if/else` en `while` statements.

Stil compileert naar de Java Virtual Machine (JVM). Dit gebeurt met twee stappen. Een programma wordt eerst gecompileerd naar de tussentaal Jasmin [1], die vervolgens compileert naar JVM code. Door het gebruik van "StringTemplates" [3] is het relatief gemakkelijk om de vertaler te herschrijven voor een andere (virtuele) machine.

In hoofdstuk 2 wordt de syntax van Stil uitgebreid beschreven. De vertaalregels voor het compileren naar de JVM worden toegelicht in hoofdstuk 3. De programmatuur wordt beschreven in hoofdstuk 4. Het testplan wordt behandeld in hoofdstuk 5.

## 1.2 Problemen en oplossingen

Bij het ontwikkelen van de Stil vertaler deden zich een aantal problemen voort. Dit hoofdstuk zal een aantal noemenswaardige uitdagingen behandelen en onze oplossingen beschrijven.

### 1.2.1 Grammatica

**Chained assignment** Een van de eisen voor de *basic expression language* was de mogelijkheid om variabelen geketend toe te wijzen met een syntax als: `a := b := c := 5`. Dit leverde een LL* probleem op voor de grammatica van de taal. Om dit te omzeilen is er gebruik gemaakt van een *Syntactic Predicate*. Hierbij wordt er in de `expression` regel een *lookahead* gedaan voor een `BECOMES` token. Dit is terug te vinden in regel 102 van `Stil.g`.

### 1.2.2 Checker

**Unassigned variable checking** Een extra functionaliteit die we wilde toevoegen aan de checker is zorgen dat unassigned variabelen niet gebruikt kunnen worden in expressies anders dan `read` of `becomes`. Door het toevoegen van conditionele statements als `if` en `while` ontstaat er echter enige ambiguïteit of een variabele wel assigned is binnen een bepaalde scope. Een taal als Java lost dit op door logischerwijs te concluderen of een variabele gegarandeerd assigned is. (een assignement binnen een 'if(true)' wordt daar als gegarandeerd beschouwd). Deze functionaliteit is wenselijk, maar werd te complex bevonden voor de scope van het vak. Er is een versimpelde assignment checking geïmplementeerd, die per scope bepaalt of een variabele geassigned is. Deze wordt verder behandeld in hoofdstuk 4.

### 1.2.3 Codegeneratie

**Return values van expressies** Een van de eisen voor de taal is dat bijna elk statement een waarde doorgeeft. Zo moet een expressie als `a := print(b)` de waarde van `b` printen en tevens toewijzen aan `a`. Om de code generatie simpel te houden is dit geïmplementeerd door elke `expression` een waarde op de stack te laten staan. Na elke compleet statement (eindigend met een `;`) wordt er een waarde van de stack gepopt. Dit heeft als voordeel dat de code-templates simpel blijven van aard. Een belangrijk nadeel van deze keuze is dat het enige overhead oplevert in het vertaalde programma. Oorspronkelijk was het plan om deze overtollige statements (vaak een extra `dup` en `pop`) weer weg te halen door een optimalisatie stap toe te voegen die de overtollige statements weer weghaalt, maar hier is helaas geen tijd meer voor gevonden in de realisatie van het project.

**Constanten** Er zijn een aantal mogelijke scenario's om constanten te implementeren in de JVM. Zo is er overwogen om compile-time constanten direct te vervangen met de letterlijke waarden. Ook is het mogelijk om gebruik te maken van de *constant pool* van de JVM. Wederom met als doel de codegeneratie simpel te houden is er besloten om constanten te implementeren als gewone variabelen. Dit levert het gemak op om geen onderscheid te hoeven maken tot compile-time constanten en run-time constanten. Wederom stond er in de planning om variabelen die constant blijven gedurende de runtime te vervangen met literal waarden in een optimalisatie stap, maar dit is niet gerealiseerd.

**Input met JVM** Voor het `read` statement wordt gebruik gemaakt van de `Scanner` class van Java. Tijdens het testen van de code liepen we tegen het probleem aan dat de `nextX()` methoden van de scanner het new line symbool niet leest. Dit leverde zeer onvoorspelbaar resultaat op bij geautomatiseerde input/output tests. Uiteindelijk hebben we dit opgelost door af te sluiten met een `scanner.readLine()` aanroep.

Een tweede probleem met het `read()` statement trad op bij meerdere read statements achter elkaar. Dit had te maken met het telkens opnieuw openen van een scanner die luistert op de standaardinvoer. Daarom wordt er nu aan het begin van het programma een scanner geopend en opgeslagen op een gereserveerde positie. Bij alle read statements wordt gebruik gemaakt van deze scanner.

# Hoofdstuk 2

# Syntax, contextbeperkingen en semantiek

## 2.1 Syntax

De syntax van Stil in Extended BackusNaur Form (EBNF) vorm:

⟨*program*⟩ ::= ⟨*instructions*⟩

⟨*instructions*⟩ ::= ((((⟨*declaration*⟩ | ⟨*expression*⟩) ‘`;`’) | ⟨*statement*⟩))*

⟨*declaration*⟩ ::= ⟨*constant_declaration*⟩
| ⟨*var_declaration*⟩

⟨*constant_declaration*⟩ ::= ‘`const`’ ⟨*type*⟩ ⟨*identifier*⟩ ‘`:=`’ ⟨*expression*⟩

⟨*var_declaration*⟩ ::= ‘`var`’ ⟨*type*⟩ ⟨*identifier*⟩

⟨*statement*⟩ ::= ⟨*if_statment*⟩
| ⟨*while_statement*⟩

⟨*if_statement*⟩ ::= ‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`{`’ ⟨*instructions*⟩ ‘`}`’ ( ⟨*empty*⟩ | ‘`else`’ ‘`{`’ ⟨*instructions*⟩ ‘`}`’)

⟨*while_statement*⟩ ::= ‘`while`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`{`’ ⟨*instructions*⟩ ‘`}`’

⟨*expression*⟩ ::= ⟨*assignment_statement*⟩
| ⟨*arithmetic_expression*⟩

⟨*compound*⟩ ::= ((((⟨*declaration*⟩ ‘`;`’) | ⟨*statement*⟩)* ⟨*expression*⟩ ‘`;`’)+

⟨*closed_compound*⟩ ::= ‘`{`’ ⟨*compound*⟩ ‘`}`’

⟨*math_expr*⟩ ::= ⟨*math_expr_pr5*⟩ (‘`||`’ ⟨*math_expr_pr5*⟩)*

⟨*math_expr_pr5*⟩ ::= ⟨*math_expr_pr4*⟩ (‘`&&`’ ⟨*math_expr_pr4*⟩)*

⟨*math_expr_pr4*⟩ ::= ⟨*math_expr_pr3*⟩ ((‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’ | ‘`==`’ | ‘`<>`’) ⟨*math_expr_pr3*⟩)*

⟨*math_expr_pr3*⟩ ::= ⟨*math_expr_pr2*⟩ ((‘`+`’ | ‘`-`’) ⟨*math_expr_pr2*⟩)*

⟨*math_expr_pr2*⟩ ::= ⟨*math_expr_pr1*⟩ ((‘`/`’ | ‘`*`’ | ‘`%`’) ⟨*math_expr_pr1*⟩)*

⟨*math_expr_pr1*⟩ ::= ‘`+`’ ⟨*operand*⟩
| ‘`-`’ ⟨*operand*⟩
| ‘`!`’ ⟨*operand*⟩
| ⟨*operand*⟩

⟨*operand*⟩ ::= ⟨*bool_literal*⟩
| ⟨*char_literal*⟩
| ⟨*int_literal*⟩
| ⟨*identifier*⟩
| ‘`(`’ ⟨*expression*⟩ ‘`)`’
| ⟨*print_statement*⟩
| ⟨*read_statement*⟩
| ⟨*closed_compound*⟩

⟨*assign_statement*⟩ ::= ⟨*identifier*⟩ ‘`:=`’ ⟨*expression*⟩

⟨*print_statement*⟩ ::= ‘`print`’ ‘`(`’ ⟨*expression*⟩ (‘`,`’ ⟨*expression*⟩)* ‘`)`’

⟨*read_statement*⟩ ::= ‘`read`’ ‘`(`’ ⟨*identifier*⟩ (‘`,`’ ⟨*identifier*⟩)* ‘`)`’

⟨*type*⟩ ::= ‘`bool`’
| ‘`char`’
| ‘`int`’

⟨*bool_literal*⟩ ::= ‘`true`’ | ‘`false`’

⟨*char_literal*⟩ ::= ‘’’ ⟨*LETTER*⟩ ‘’’

⟨*int_literal*⟩ ::= ⟨*DIGIT*⟩+

⟨*identifier*⟩ ::= ⟨*LETTER*⟩ (⟨*LETTER*⟩ | ⟨*DIGIT*⟩)*

⟨*DIGIT*⟩ ::= ‘`0`’..‘`9`’

⟨*LETTER*⟩ ::= ‘`a`’..‘`Z`’

## 2.2 Contextbeperkingen

### 2.2.1 Scoperegels

Stil vertoont een *nested block structure*. Dit betekent dat er meerdere *scope-levels* zijn:

- Declaraties in het uiterste block krijgen scope-level 0 toegewezen
- Een `if`, `else`, `while` en `compound-expression` openen een nieuw block. Declaraties in deze scopes krijgen een hoger scope-level. Omdat blocks genest kunnen worden – een `compound-expression` kan bijvoorbeeld ook een `if`-statement bevatten – wordt bij elk geopende block het scope-level opgehoogd en weer verlaagd als het block wordt afgesloten.

Voor Stil gelden de volgende scoperegels:

- Een identifier mag slechts een keer gedeclared zijn per scope-level. Dezelfde identifier mag wel vaker gedeclareerd worden op meerdere scope-levels, zelfs in nested blocks.
- Voor elke keer dat een identifier zich voordoet, moet er een overeenkomstige declaratie zijn. Dit kan in het huidige block, of in een van de overkoepelende blocks. De declaratie uit het block met een hoger scope-level heeft hierin voorrang.
- Voor elk gebruik van een identifier dat niet een declaratie, toewijzing of inlezing (`read`) is, moet de variabele geassigned zijn. Dit kan op het huidige scopelevel, of op een scopelevel daaronder. Een assignment in een block met een scope-level hoger dan het huidige is niet geldig voor het huidige block.

### 2.2.2 Typeregels

Stil is volledig *statically typed*. Dit impliceert dat elke expressie een type heeft, welke bepaald kan worden zonder de expressie uit te voeren. Hieronder staan de typeregels beschreven. Als kanttekening moet hierbij gezegd worden dat 'geen type' hier equivalent is met type `void`.

- De typeregel van `const T I := E` is:
  E moet van type T zijn
  I wordt type T
  Statement returnt geen type.
- De typeregel van `var T I` is:
  I wordt type T
  Statement returnt geen type.
- De typeregel van `if(E)` {`<instructions>`} `(else` {`<instructions>`}`)?` is:
  E moet van type `bool` zijn.
  Statement returnt geen type.
- De typeregel van `while(E)` {`<instructions>`} is:
  E moet van type `bool` zijn.
  Statement returnt geen type.
- De typeregel van `print(E*)` is:
  Statement returnt type van $E_1$ indien enkele expressie of `void` indien meerdere expressies.
- De typeregel van `read(E*)` is:
  Analoog aan print: statement returnt type van $E_1$ indien enkele expressie of `void` indien meerdere expressies.
- Compound Expression: De typeregel van { `<instructions>` } is:
  Statement returnt met type van de laatste instructie. En kan dus ook void zijn.
- De typeregel van `I := E` is:
  E moet hetzelfde type hebben als I
  Statement returnt met type I
- De typeregel van `E1 || E2` is:
  E1 en E2 moeten beide van type `bool` zijn.
  Statement returnt met type `bool`.
- De typeregel van `E1 && E2` is:
  E1 en E2 moeten beide van type `bool` zijn.
  Statement returnt met type `bool`.

- De typeregel van `E1 < E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `bool`.

- De typeregel van `E1 <= E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `bool`.

- De typeregel van `E1 > E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `bool`.

- De typeregel van `E1 >= E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `bool`.

- De typeregel van `E1 == E2` is:
  E1 en E2 moeten van (gelijkwaardig) type `int`, `bool` of `char` zijn.
  Statement returnt met type `bool`.

- De typeregel van `E1 <> E2` is:
  E1 en E2 moeten van (gelijkwaardig) type `int`, `bool` of `char` zijn.
  Statement returnt met type `bool`.

- De typeregel van `E1 + E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `int`.

- De typeregel van `E1 - E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `int`.

- De typeregel van `E1 / E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `int`.

- De typeregel van `E1 * E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `int`.

- De typeregel van `E1 % E2` is:
  E1 en E2 moeten beide van type `int` zijn.
  Statement returnt met type `int`.

- De typeregel van `+E` is:
  E moet van type `int` zijn.
  Statement returnt met type `int`.

- De typeregel van `-E` is:
  E moet van type `int` zijn.
  Statement returnt met type `int`.

- De typeregel van `!E` is:
  E moet van type `bool` zijn.
  Statement returnt met type `bool`.

- De typeregel van `bool_literal` is:
  Statement returnt met type `bool`.
- De typeregel van `int_literal` is:
  Statement returnt met type `int`.
- De typeregel van `char_literal` is:
  Statement returnt met type `char`.

## 2.3 Semantiek

De semantiek van de Stil syntax is gebaseerd op de semantiek van de *Basic Expression Language*. Het wijkt alleen op enkele punten af, maar voor de volledigheid zal de semantiek per instructie behandeld worden:

- Het constant-declaration `const T I := E` wordt als volgt uitgevoerd. I wordt gedefinieerd als een constante van type T met als waarde E.
- Het var-declaration `var T I` wordt als volgt uitgevoerd. T wordt gedefinieerd als variabele van type T (maar zonder initiële waarde).
- Het arithmetic-expression `E1 (<operator> E2)?` wordt als volgt uitgevoerd. E1 wordt geëvalueerd. Afhankelijk van de operator wordt er een logische of arithmatische operatie gedaan. Omdat E1 en E2 zelf ook weer `arithmetic_expression`s kunnen zijn, worden er meerdere niveaus van prioriteit gebruikt. Deze zijn beschreven in Tabel 2.3

Tabel 2.1: Prioriteiten

| **Prioriteit** | **Operatoren** | **Operandtypen** | **Resultaattype** |
|---|---|---|---|
| 1 | (unary) -, + | int | int |
| | ! | bool | bool |
| 2 | *, /, % | int | int |
| 3 | +, - | int | int |
| 4 | <, <=, >=, > | int | bool |
| | ==, <> | int, bool, char | bool |
| 5 | && | bool | bool |
| 6 | \|\| | bool | bool |

- Het unary-expression `(+|-|!)Operand` wordt als volgt uitgevoerd. De operator `plus`, `minus` of `not` worden uitgevoerd op de operand, de waarde wordt gereturnt. Een operand kan een `literal`, `(expression)` (expressie met haakjes omvat), `read-statement`, `print-statement` of `closed_compound_expression` zijn.
- Het assignment-statement `I := E` wordt als volgt uitgevoerd. E wordt geëvalueerd en de resulterende waarde wordt toegewezen aan I.
- Het print-statement `print(E*)` wordt als volgt uitgevoerd. Elke expressie E wordt geëvalueerd en geprint op de *standard output*. Indien er slechts een Expressie is, wordt de waarde van E teruggegeven.
- Het read-statement `read(I*)` wordt als volgt uitgevoerd. Voor elke variabele I wordt een waarde met type gelijk aan het type van I van de *standard input* ingelezen. Indien er slechts een Identifier is, wordt de ingelezen waarde teruggegeven met het type van I. Indien input niet omgezet kan worden naar het type van I wordt de waarde: 0 in het geval van `int`, 0 in het geval van `char` en `false` in het geval van `bool`.

- Het closed-compound-statement {`<instructions>`} wordt als volgt uitgevoerd. De instructies worden geëvalueerd. De waarde van de laatste `instruction` wordt teruggegeven. Dit kan ook een lege waarde zijn.

Naast de *Basic Expression Language* zijn ook `if-else` en `while` statements toegevoegd:

- Het if-statement `if(E)` {`<instructions_1>`} `(else` {`<instructions_2>`}`)?` wordt als volgt uitgevoerd. Eerst wordt E geëvalueerd. Indien E `true` opleverd worden `<instructions_1>` uitgevoerd. Indien E `false` opleverd worden `<instructions_2>` uitgevoerd, mits het `else` deel aanwezig is.
- Het while-statement `while(E)` {`<instructions>`} wordt als volgt uitgevoerd. Eerst wordt E geëvalueerd. Indien E `true` opleverd worden `<instructions>` uitgevoerd. Vervolgens herhaalt het bovenstaande zich, todat E `false` oplevert.

# Hoofdstuk 3

# Vertaalregels

De formele vertaalregels zullen in dit hoofdstuk gedefineerd worden. Voor de gerealiseerde vertaaltemplates voor Jasmin, verwijzen we de lezer naar appendix E.

Definities:
`T`: `type`
`E`: `expression`
`I`: `identifier`

*execute* [`INSTRUCTION`] =

    *evaluate* INSTRUCTION
    `pop`

*execute* [`I := E `] =

    *evaluate* E
    `dup`
    `istore` I

*execute* [`const T I := E`] =

    *evaluate* E
    `dup`
    `istore` I

*execute* [`var T I`] =

    no operation

*execute* [`if (E) {if-instructions} else {else-instructions}`] =

    *evaluate* E
    `ifeq L1`
    *evaluate* if-instructions
    `goto L2`
    `L1:`
    *evaluate* else-instructions
    `L2:`
    `iconst_0`

*execute* [`while (E) {instructions}`] =

```
L1:
evaluate E
ifeq L2
evaluate instructions
goto L1
L2:
iconst_0
```

*execute* [`print(expressions)`] =

*evaluate* print-single(expression) forall expressions

*execute* [`print-single(expression)`] =

```
evaluate E
dup
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(<type(expression)>)V
```

*execute* [`read(identifiers)`] =

*evaluate* readMultiple(identifier) forall identifiers

(vanwege de complexiteit van dit statement verwijzen we de lezer naar het StringTemplate bestand `jasmin.stg`, te vinden in appendix E).

*(closed compound expression)*
*execute* [{`instructions`} ] =

*evaluate* instructions

*(unary minus)*
*execute* [`-E`] =

```
evaluate E
ineg
```

*(unary plus)*
*execute* [`+E`] =

*evaluate* E

*(unary not)*
*execute* [`!E`] =

```
evaluate E
ifeq L1
bipush 0
goto L2
L1:
bipush 1
L2:
```

*(binary operator)*

*execute* [`E1` ***`operator`*** `E2`] =

*evaluate* E1
*evaluate* E2
*evaluate* operator

# Hoofdstuk 4

# Beschrijving van Java-programmatuur

In de verschillen fasen van het compileerproces worden verschillende Java-klassen gebruikt ter ondersteuning hiervan. In de onderstaande secties wordt per fase uitgelicht welke klassen wij hiervoor gedefinieerd hebben en hoe deze samenwerken.

Alle klassen en packages bevinden zich in de package `vb.stil`. De grammatica van de taal wordt beschreven met behulp van ANTLR [4]. De ANTLR-grammatica's en de gegenereerde Java-code bevinden zich in deze root. `Stil.java` bevat de `main`-methode en hier worden alle componenten geladen en op de juiste wijze aangeroepen. Bij het aanroepen van het programma zijn de volgende flags beschikbaar:

**-AST**
: Output de gegenereerde AST als tekstrepresentatie op de standaarduitvoer.

**-DOT**
: Output de gegenereerde AST als DOT-representatie op de standaarduitvoer.

**-NO_ASSEMBLE**
: Genereer geen Jasmin-bestand (`gen/program.j`).

**-NO_CHECKER**
: Sla de checkerfase over.

**-NO_CODE_GENERATOR**
: Sla de codegeneratiefase over. Wordt afgedwongen door `-NO_CHECKER`.

**-NO_JAR**
: Genereer geen Jar-bestand (`gen/program.jar`).

Standaard (zonder expliciet flags mee te geven) worden er drie fasen doorlopen: het lexen en parsen, checken en genereren van code.

## 4.1 Grammatica

Zoals gespecificeerd in de grammatica `Stil.g` wordt er gebruik gemaakt van eigen nodes: `DeclNode`, `ExprNode`, `IdNode`, `LiteralNode` en `LogicExprNode`. Deze zijn alle gedefinieerd in de `vb.stil.tree` package en zijn een subklasse van `StilNode`. Wij definiëren eigen StilNodes zodat we de AST kunnen decoreren met specifieke eigenschappen. In Sectie 4.2 wordt hier verder op ingegaan.

Tot slotte moeten we het gebruik van eigen nodes vastleggen door een `StilNodeAdaptor` te definiëren en te registreren. Opvallend hierbij is is het overriden van een aantal methoden zoals `dupNode()`. Dit is van essentieel belang voor een juiste werking van ANTLR.

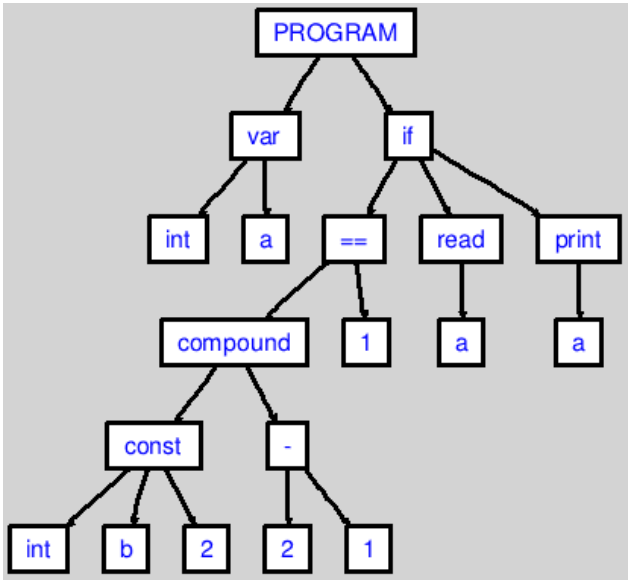
Aan het eind van deze fase is een boomstructuur van de verschillende nodes opgebouwd in de vorm van een Abstract Syntax Tree (AST). Deze wordt doorgegeven aan de checker. Ter illustratie laten wij de AST zien van het volgende test programma, zie figuur 4.1.

Listing 4.1: Voorbeeld code

```
var int a;

if({const int b := 2; 2-1;} == 1) {
    read(a);
    print(a);
}
```

Figuur 4.1: AST van testprogramma




## 4.2 Checker

In de checkerfase wordt over de AST gelopen. Voor alle declaraties, statements en expressies wordt een `processX()` methode aangeroepen in de `TypeChecker` of `DeclarationChecker` klasse. Deze klassen zijn ondergebracht in de `vb.stil.checker` package. Op deze wijze hebben we zoveel mogelijk logica ondergebracht in Java-klassen in plaats van de ANTLR-grammatica.

In deze fase wordt een zogeheten Symbol Table geïnitialiseerd voor het bijhouden van de declaraties van constanten en variabelen en hun bijbehorende scope. Dit stelt ons in staat om per scope (bij een `compound`, `if` en `while` statement wordt een nieuwe scope geopend) te bepalen of een identifier gedeclareerd is. De symbol table is te vinden in de package `vb.stil.symtab` en een entry wordt gerepresenteerd door middel van de `IdEntry` klasse. Per declaratie wordt ook een verwijzing naar de `DeclNode` in de AST geregistreerd om eigenschappen opgeslagen in deze node te kunnen gebruiken.

**Voorbeeld** Bij de declaratie van een constante wordt `processConstantDeclaration()` aangeroepen met als parameters de `DeclNode`, identifier, het type en de symbol table instantie. De mogelijke typen worden gedefinieerd in de enumeratie `EntityType`, te vinden in de package `vb.stil.tree`. Wat deze methode doet is het decoreren van de `DeclNode` met het type (bijvoorbeeld `char`) en de kind (bijvoorbeeld `const`). Ten slotte wordt de declaratie van de identifier samen met een verwijzing naar de `DeclNode` toegevoegd aan de symbol table. Als een identifier al gedeclareerd is in de betreffende scope wordt dat op dit punt opgemerkt door de symbol table en wordt er een exceptie gegooid. Op dit punt wordt nog niet gekeken of het type van de declaratie overeenkomt met het type van de expressie. Dit wordt wel gecontroleerd in de checker omdat ook de `processConstantAssignmentExpression()` methode aangeroepen wordt. Deze methode controleert of de identifier gedeclareerd is, of dit als constante gedeclareerd is en of het type van de expressie overeenkomt met het type van de declaratie. Zie Listing 4.2 voor een voorbeeldimplementatie.

Listing 4.2: Voorbeeld van het registreren van de declaratie van een constante.

```
public void processDeclaration(DeclNode node, StilNode id, EntityType type, DeclNode.Kind
    kind, SymbolTable<IdEntry> symtab) throws StilException {
  node.setEntityType(type);
  node.setKind(kind);

  try {
    IdEntry entry = new IdEntry();

    entry.setLevel(symtab.currentLevel());
    entry.setDeclNode(node);

    symtab.enter(id.getText(), entry);
  } catch (SymbolTableException e) {
    throw new StilException(node, e.getMessage());
  }
}
```

Listing 4.3: Voorbeeld van het checken van de declaratie van een constante.

```
public EntityType processConstantAssignmentExpression(DeclNode node, StilNode id, EntityType
    t1, SymbolTable<IdEntry> symtab) throws StilException {
  IdEntry symbol = symtab.retrieve(id.getText());

  if (symbol == null) {
    throw new StilException(node, "use of undeclared identifier");
  }

  DeclNode declNode = symbol.getDeclNode();

  if (declNode.getEntityType() != t1) {
    throw new StilException(node, "operand type does not match identifier type");
  }

  if (!declNode.isConstant()) {
    throw new StilException(node, "identifier must be declared as variable");
  }

  return t1;
}
```

Op soortgelijke wijze worden de andere nodes doorlopen waarbij opvallend is dat bij logische expressies ook de operator (vastgelegd in de enumeratie `vb.stil.tree.Operator` wordt opgeslagen in de `LogicExprNode`. Voor alle declaraties en expressies wordt het type van de node bepaald, afhankelijk van de operands, en opgeslagen in het node object. Zo zal een `print` statement bij een enkele parameter het type van de parameter aannemen en bij meerdere parameters het type `void`.

Excepties die gegooid worden door Stil worden getypeerd als `StilException`, te vinden in de package `vb.stil.exceptions`. Waar mogelijk wordt bij het gooien van een exceptie de AST-node meegegeven als parameter. Dit stelt ons in staat om lijn- en regelnummers te tonen in de foutmeldingen.

Na het succesvol (zonder optreden van excepties) doorlopen van de checkerfase zijn aan alle contextbeperkingen, scope en type regels, voldaan. De AST is gedecoreerd met referenties van identifiers naar de declarerende `DeclNode`, type- en kind-informatie en operatoren. Deze gedecoreerde AST wordt doorgegeven aan de codegenerator.

## 4.3 Codegeneratie

Op soortgelijke wijze als bij de checker wordt bij de codegeneratie over de AST gelopen. In dit geval betreft het de gedecoreerde AST. Codegeneratie is dus alleen mogelijk als daarvoor de checkerfase doorlopen is. Wederom worden `processX()` methoden aangeroepen, maar ditmaal in de `CodeGenerator` klasse, te vinden in de `vb.stil.codegenerator` package.

Codegeneratie geschiedt door middel van `StringTemplates`. Dit stelt ons in staat om verschillende templates te definiëren in in een zogeheten group file. Op deze wijze is het heel makkelijk de templates te wijzigen en andere uitvoer te genereren, bijvoorbeeld `TAM`-machinecode. In ons geval compileren wij naar Jasmin en zijn de templates gedefinieerd in `jasmin.stg` in de package `vb.stil.codegenerator`. Een voorbeeld van een template is te vinden in Listing 4.4.

Listing 4.4: StringTemplate voor vergelijkingsoperatoren.

```
logic_operators ::= [
  "lt"  : "if_icmplt",
  "lte" : "if_icmple",
  "gt"  : "if_icmpgt",
  "gte" : "if_icmpge",
  "eq"  : "if_icmpeq",
  "neq" : "if_icmpne"
]

comparison(expr1, expr2, label1, label2, operator) ::= <<
<expr1>
<expr2>
<logic_operators.(operator)> L<label1>
bipush 0
goto L<label2>
L<label1>:
bipush 1
L<label2>:
>>
```

De templates bevatten placeholders waarvan de waardes worden bepaald opgegeven in de `processX()` methoden van de `CodeGenerator`-klasse. Aangezien veel placeholders weer gevuld moeten worden door andere templates, een tweeledige expressie heeft bijvoorbeeld de gevulde template van de twee expressies nodig, wordt de gegeneerde template (een `ST` object) aan de `StilNodes` toegevoegd: de AST wordt verder gedecoreerd.

Voor constructies waarbij labels nodig zijn is een labelnummer variabele gedefinieerd welke als waarde altijd het volgende beschikbaar labelnummer heeft. Bij het opvragen van een beschikbaar labelnummer door middel van `getNewLabelNumber()` wordt dit labelnummer teruggeven en het labelnummer opgehoogd. Deze nummers worden vervolgens in de template gesubstitueerd.

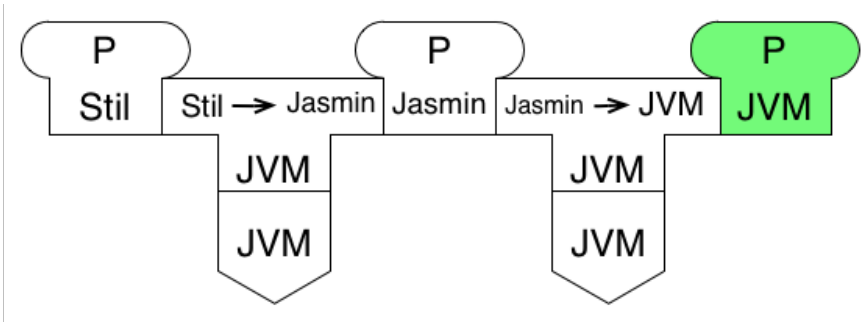
Aangezien wij zoveel mogelijke logica in de Java-klassen willen stoppen in plaats van in de ANTLR-specificaties maken wij in de `processX()`-methoden veel gebruik van de structuur van de AST. Bij bij-

voorbeeld het `print`-statement halen wij rechtstreeks de `ST`-templates op uit de *child nodes*: `((StilNode) node.getChild(index)).getST()`.

Bij het genereren van code wordt eerder opgeslagen informatie uit de nodes, bijvoorbeeld het type, gebruikt.

Uiteindelijk komen alle templates uit bij de root node van de AST: `program`. Hier worden de templates van alle instructies onder elkaar geplaatst. Tot slot wordt deze volledige template teruggeven en als valide Jasmin-file weggeschreven naar `gen/program.j`. Dit Jasmin file wordt vervolgens gecompileerd naar een .class file wat geschikt is voor de JVM. Dit wordt direct vanuit de code-generator gedaan door de Jasmin compiler aan te roepen. Figuur 4.3 illustreert deze twee-staps compilatie.

Figuur 4.2: Tombstone diagram voor Stil-compilatie:




Deze modulaire splitsing van ANTLR grammatica, Java code en StringTemplates stelt ons in staat om verantwoordelijkheden op de juiste plek te brengen en problemen te isoleren. Ook stelt het ons in staat om functionaliteit makkelijk toe te voegen of aan te passen.

# Hoofdstuk 5

# Testplan en -resultaten

Om de taal te testen zijn tests ontwikkeld voor verschillende fasen: voor de lexicale syntax, contextbeperkingen en uiteindelijke bytecode. Ter ondersteuning van het testen van de grammatica en de contextbeperkingen wordt het testingframework `gUnit` [2] gebruikt. gUnit stelt ons ook in staat om lexerregels, zoals `IDENTIFIER` te testen. Er wordt getest op zowel correcte als incorrecte programma's.

Daarnaast hebben we bij het ontwikkelen gebruik van test-driven development. Dat houdt in dat eerst tests zijn ontwikkeld en daarna pas begonnen is met implementeren.

**Lexer en parser.** De gUnit tests voor de lexer en parser zijn te vinden in `gunit/Stil.gunit`. Het bevat tests voor de verschillende statements en expressies die worden ondersteund door onze taal. Per test wordt aangegeven of verwacht wordt dat deze test gepasseerd of gefaald wordt. Zie Listing A.1 voor de tests. Er worden ook een aantal grotere tests aangeroepen zoals `statements.st` en per test is commentaar toegevoegd waarom een test zou moeten falen. Zie hiervoor de inhoud van de tests zelf in de map `gunit/`.

**Contextbeperkingen (checker).** De volgende stap is het testen van de contextbeperkingen. De gUnit tests voor de checker zijn te vinden in `gunit/StilChecker.gunit`. Het bevat tests voor de verschillende statements en expressies die worden ondersteund door onze taal waarbij gefocust is op de contextbeperkingen: scope en type rules. Per test wordt aangegeven of verwacht wordt dat deze test gepasseerd of gefaald wordt. Zie Listing A.2 voor de tests. Er worden ook een aantal grotere tests aangeroepen zoals `statements.st` (detail: deze zelfde test wordt gebruikt bij het testen van de lexer en parser) en per test is commentaar toegevoegd waarom een test zou moeten falen. Zie hiervoor de inhoud van de tests zelf in de map `gunit/`.

**Runtime errors.** De laatste vorm van fouten die kunnen optreden zijn runtime errors. Dit zijn fouten die eigenlijk te maken hebben met een fout door de gebruiker/programmeur. Een voorbeeld hiervan is het gebruiken van een variabele terwijl deze wel gedeclareerd is, maar nog geen waarde is toegekend. Voor deze specifieke runtime error is een controle ingebouwd in de checker. Bij het gebruik van een identifier wordt gecontroleerd of er een waarde toegekend is voor deze identifier op de huidige scope. Dit wordt al getest bij de tests van de checker, zie de laatste tests van het onderdeel *declarations* in Listing A.2.

Een ander voorbeeld is het optreden van een deling door nul (*Division by zero*). Dit gedrag resulteert, zoals verwacht, in een Java `ArithmeticException`. Dit wordt getest in `tests/runtime_error.st`. Dit gebeurt zonder gUnit, maar door standaardinvoer aan te leveren de standaarduitvoer te vergelijken met de verwachte uitvoer.

**Correct programma.** Tot slot is een programma geschreven waarin alle facetten van de taal voorkomen. Hieronder wordt kort toegelicht waarop gelet is bij het bepalen van het testprogramma.

**Declaraties.** Elke variable en constante moet gedeclareerd worden met het type `int`, `char` of `bool`.

**Operatoren en operands.** De operatoren kunnen voor een of meerdere operands en operand van verschillende typen gedefinieerd zijn. Daarnaast zijn bepaalde operatoren geprioriteerd boven andere.

**Assignments.** Er komen zowel enkele als meerdere assignments voor en een assignment heeft een type.

**Read en print.** Een read statement omvat een of meerdere identifiers en het type is afhankelijk van het aantal parameters. Voor een print statement geldt hetzelfde, maar deze kan ook expressies als parameter ontvangen welke eerst geëvalueerd moeten worden.

**Compound expressies.** Een opeenvolging van declaraties, statements en expressies. Moet eindigen met een expressie.

**If en while.** Betreft een statement dus heeft geen type en geeft geen waarde terug. De inhoud kan declaraties, statements en expressies in willekeurige volgorde betreffen.

De bovenstaande eigenschappen zijn allemaal in `tests/stil_language.st` te vinden. Het bestand `tests/stil_language.st.in` schrijft de invoer voor en `tests/stil_language.st.out` de verwachte uitvoer. Alle tests in `tests/` hanteren dit formaat en de tests kunnen geautomatiseerd uitgevoerd worden door `output_test.sh` uit te voeren.

# Hoofdstuk 6

# Conclusies

In dit verslag is de programmeertaal Stil en de bijbehorende vertaler beschreven. Na in te zijn gegaan op enkele problemen en bijhorende oplossingen is de syntax behandeld. Kort samengevat is de syntax bijna gelijk aan de *Basic Expression Language*, aangevuld met syntax voor `if-else` en `while` statements. De taal kent geen functies, maar wel nested blocks, wat resulteert in enkele scope rules. Naast (de standaard) declaratie-checking wordt er ook gecontroleerd of een variabele gegarandeerd geassigned is binnen een block, en is de taal volledig statically typed. De belangrijkste typeregel is dat elk statement een return value heeft (met uitzondering van `while` en `if-else` en `read`/`print` met meerdere parameters).

Vervolgens zijn de vertaalregels gedefinieerd. De implementatie van deze regels is te vinden in appendix E. Noemenswaardig is dat elk statement/expressie wordt gevolgd met een `pop` instructie. Elk statement hoort dus een waarde op de stack te laten staan.

De Java-programmatuur van de vertaler is behandeld in hoofdstuk 4. De beschikbare executie-flags en de implementatie van de grammatica, checker en codegeneratie zijn beschreven. De programmatuur parset en checkt de code en compileert deze door middel van StringTemplates via Jasmin naar de JVM.

Als laatste is het testframework beschreven. Deze bestaat uit twee onderdelen: gUnit tests die de grammatica en checker controleren, en input/output tests die de volledige programmatuur controleren door middel van testprogramma's met gegeven input en verwachte output.

Al met al is er een complete vertaler opgeleverd met een uitgebreid testplan. Hoewel niet alle doelstellingen op het gebied van codeoptimalisatie en syntax-schoonheid zijn behaald, is Stil ontwikkeld tot een simpele doch robuuste taal. De nette opbouw, het gebruik van StringTemplates en gUnit tests bieden een stabiel raamwerk voor verdere ontwikkeling van de taal.

# Bibliografie


[1] J. Meyer. Jasmin Instruction Guide. http://jasmin.sourceforge.net. Accessed: 2014-06-14.

[2] T. Parr. gUnit - Grammar Unit Testing. https://theantlrguy.atlassian.net/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing. Accessed: 2014-06-05.

[3] T. Parr. StringTemplate 4 Documentation. http://www.stringtemplate.org. Accessed: 2014-06-13.

[4] T. Parr. The Definitive ANTLR Reference (ISBN: 978739256), May 2007.

# Bijlage A

# gUnit tests

## A.1 Lexer en parser

Listing A.1: GUnit tests voor de lexer en parser.

```
gunit Stil;

@header{package vb.stil;}

program:
basic.st       OK
types.st       OK
statements.st OK

// SEMICOLONS
"var int x; x := y"   FAIL
"var int x x := y;"   FAIL
"var int x; x := y;" OK

// COMPOUND
"var int x; read(x); x := print({var int y; read(y); x + y;});" OK
"var int x; read(x); x := print({var int y; read(y); x + y});"   FAIL

// DECLARATIONS
"const int x;"       FAIL
"const int x := 5;" OK
"var int x;"         OK
"var int x := 5;"    FAIL

// PRINT
"print(1);"         OK
"print(1)"          FAIL
"print(1,true);"   OK
"print('b', 10);" OK
"print(a);"         OK
"print false;"      FAIL

// READ
"read(1);"    FAIL // Can only read identifiers
"read(a)"     FAIL // Must end with ;
"read(a,b);" OK
"read b;"     FAIL // Must surround operand with ( )

// UNKNOWN STATEMENT
"raed(b);"   FAIL
"while(c);" FAIL

// IF STATEMENT
```

```
"if(true) { }"                                          OK
"if(true) { };"                                         FAIL
"if(1 >= 2) { }"                                        OK
"if(true)"                                              FAIL
"if(false) { const int x := 4; print(x); }"             OK
"if(true) { } else { }"                                 OK
"if(false) { } else"                                    FAIL

// WHILE STATEMENT
"while(true) { }"                                       OK
"while(true) { };"                                      FAIL
"while(1 >= 2) { }"                                     OK
"while(true)"                                           FAIL
"while(false) { const int x := 4; print(x); }"          OK

IDENTIFIER:

"abc123"   OK
"a"        OK
"XYZ@999"  FAIL
"123abc"   FAIL

INT_LITERAL:

"00000"      OK
"123456789"  OK
"-1"         FAIL
```

## A.2 Checker

Listing A.2: GUnit tests voor de checker.

```
gunit StilChecker walks Stil;

options {
  TreeAdaptor = vb.stil.tree.StilNodeAdaptor;
}

@header{package vb.stil;}
program walks program:
simple.st          OK
types.st           OK
duplicatedecl.st   FAIL
statements.st      OK

// DECLARATIONS
"var int x; var int x;"                                 FAIL
"var int x; x := print({var int x; read(x); x+5;});"    OK
"var int x; { x := 4; }; print(x);"                     FAIL
"var int x; if(true) { x := 4; } print(x);"             FAIL

// ASSIGNMENTS
"var int x; print(x);"                  FAIL
"var int x; var int y; y := x;"         FAIL
"var int x; var int y; x := 5; y := x;" OK
"var int x; read(x);"                   OK

// PRINT
"var int x; print(y);"                  FAIL
"var int x; var int y; print(x,y,z);"   FAIL
"var int x; x := print(2);"             OK
"var int x; var int y; x := print(x,y);" FAIL
"var int x; x := print(3*3);"           OK
"var bool t; t := print(5>3);"          OK
"var int x; var bool t; x := print(t);" FAIL
```

```
"var int x; var bool t; t := print(x+5);" FAIL

// READ
"var int x; read(y);"                       FAIL
"var int x; var int y; read(x,y,z);"        FAIL
"var int x; var int y; y := read(x);"       OK
"var int x; var int y; x := read(x,y);"     FAIL
"var int x; x := read(3*3);"                FAIL
"var int x; var bool t; x := read(t);"      FAIL

// COMPOUND
"var int x; read(x); x := print({var int y; read(y); x + y;});" OK
"var int x; read(x); x := {var int y; read(y); x + y;};"        OK
"var int x; x := {var bool y; read(y); 1+1;};"                  OK
"var int x; {var int x; read(x);};"                             OK
"var int x; x := {var bool y; read(y); y;};"                    FAIL
"var int x; x := {x; var bool y;};"                             FAIL
"var int x; x := {var bool y; read(y); x;}; print(y);"          FAIL

// IF STATEMENT
"if(true) {}"   OK
"if(2) {}"      FAIL
"if(1 >= 2) {}" OK

// WHILE STATEMENT
"while(true) {}"   OK
"while(2) {}"      FAIL
"while(1 >= 2) {}" OK
```

# Bijlage B

# Lexer- en parserspecificatie

Listing B.1: ANTLR-specificatie voor de lexer en parser.

```
grammar Stil;

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

tokens {
    COMMA        =   ','     ;
    COLON        =   ':'     ;
    LPAREN       =   '('     ;
    RPAREN       =   ')'     ;
    LCURLY       =   '{'     ;
    RCURLY       =   '}'     ;
    SEMICOLON    =   ';'     ;
    APOS         =   '\''    ;

    // operators
    BECOMES      =   ':='    ;
    OR           =   '||'    ;
    AND          =   '&&'    ;
    LT           =   '<'     ;
    LTE          =   '<='    ;
    GT           =   '>'     ;
    GTE          =   '>='    ;
    EQ           =   '=='    ;
    NEQ          =   '<>'    ;
    PLUS         =   '+'     ;
    MINUS        =   '-'     ;
    DIVIDE       =   '/'     ;
    MULTIPLY     =   '*'     ;
    MODULO       =   '%'     ;
    NOT          =   '!'     ;

    // keywords
    BOOL          =   'bool'     ;
    COMPOUND_EXPR =   'compound' ;
    CONST         =   'const'    ;
    CHAR          =   'char'     ;
    ELSE          =   'else'     ;
    FALSE         =   'false'    ;
    IF            =   'if'       ;
    INT           =   'int'      ;
    PRINT         =   'print'    ;
    PROGRAM       =   'program'  ;
```

```
    READ            =   'read'      ;
    TRUE            =   'true'      ;
    UNARY_MINUS     =   'minus'     ;
    UNARY_NOT       =   'not'       ;
    UNARY_PLUS      =   'plus'      ;
    VAR             =   'var'       ;
    WHILE           =   'while'     ;
}

@lexer::header {
    package vb.stil;
}

@header {
    package vb.stil;
    import  vb.stil.tree.*;
}

// Parser rules

program
    :   instructions EOF
            -> ^(PROGRAM instructions)
    ;

instructions
    : (((declaration | expression) SEMICOLON!) | statement)*
    ;

declaration
    :   constant_declaration
    |   var_declaration
    ;

constant_declaration
    :   CONST<DeclNode>^ type IDENTIFIER<IdNode> BECOMES! expression
    ;

var_declaration
    :   VAR<DeclNode>^ type IDENTIFIER<IdNode>
    ;

statement
    :   (if_statement | while_statement)
    ;

if_statement
    :   IF^ LPAREN! expression RPAREN! LCURLY! instructions RCURLY! (ELSE LCURLY!
        instructions RCURLY!)?
    ;

while_statement
    :   WHILE^ LPAREN! expression RPAREN! LCURLY! instructions RCURLY!
    ;

expression
    :   (IDENTIFIER<IdNode> BECOMES) => assignment_statement
    |   arithmetic_expression
    ;

compound_expression
    :   (((declaration SEMICOLON!) | statement)* expression SEMICOLON!)+
    ;

closed_compound_expression
```

```
    :   LCURLY compound_expression RCURLY -> ^(COMPOUND_EXPR<ExprNode> compound_expression)
    ;

// priority 6
arithmetic_expression
    :   arithmetic_expression_pr5 (OR<LogicExprNode>^ arithmetic_expression_pr5)*
    ;

arithmetic_expression_pr5
    :   arithmetic_expression_pr4 (AND<LogicExprNode>^ arithmetic_expression_pr4)*
    ;

arithmetic_expression_pr4
    :   arithmetic_expression_pr3 ((LT<LogicExprNode>^ | LTE<LogicExprNode>^ | GT<
        LogicExprNode>^ | GTE<LogicExprNode>^ | EQ<LogicExprNode>^ | NEQ<LogicExprNode>^)
        arithmetic_expression_pr3)*
    ;

arithmetic_expression_pr3
    :   arithmetic_expression_pr2 ((PLUS<LogicExprNode>^ | MINUS<LogicExprNode>^)
        arithmetic_expression_pr2)*
    ;

arithmetic_expression_pr2
    :   arithmetic_expression_pr1 ((DIVIDE<LogicExprNode>^ | MULTIPLY<LogicExprNode>^ |
        MODULO<LogicExprNode>^) arithmetic_expression_pr1)*
    ;

arithmetic_expression_pr1
    :   PLUS operand    -> ^(UNARY_PLUS<LogicExprNode> operand)
    |   MINUS operand   -> ^(UNARY_MINUS<LogicExprNode> operand)
    |   NOT operand     -> ^(UNARY_NOT<LogicExprNode> operand)
    |   operand
    ;

operand
    :   bool_literal
    |   CHAR_LITERAL<LiteralNode>
    |   INT_LITERAL<LiteralNode>
    |   IDENTIFIER<IdNode>
    |   LPAREN! expression RPAREN!
    |   print_statement
    |   read_statement
    |   closed_compound_expression
    ;

assignment_statement
    :   IDENTIFIER<IdNode> BECOMES<ExprNode>^ expression
    ;

print_statement
    :   PRINT<ExprNode>^ LPAREN! expression (COMMA! expression)* RPAREN!
    ;

read_statement
    :   READ<ExprNode>^ LPAREN! IDENTIFIER<IdNode> (COMMA! IDENTIFIER<IdNode>)* RPAREN!
    ;

type
    :   BOOL | CHAR | INT
    ;

bool_literal
    :   TRUE<LiteralNode>
    |   FALSE<LiteralNode>
```

```
    ;

// Lexer rules

CHAR_LITERAL
    :   APOS l=CHARALL APOS { setText(l.getText()); }
    ;

INT_LITERAL
    :   DIGIT+
    ;

IDENTIFIER
    :   LETTER (LETTER | DIGIT)*
    ;

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

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

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

# Bijlage C

# Checkerspecificatie

Listing C.1: ANTLR-specificatie voor de checker.

```
tree grammar StilChecker;

options {
    tokenVocab=Stil;                    // import tokens from Stil.tokens
    ASTLabelType = StilNode;            // AST nodes are of type StilNode
}

@header {
    package vb.stil;
    import  vb.stil.checker.*;
    import  vb.stil.symtab.*;
    import  vb.stil.tree.*;
    import  vb.stil.exceptions.*;
}

@rulecatch {
    catch (RecognitionException e) {
        throw e;
    }
}

@members {
    protected SymbolTable<IdEntry> symtab = new SymbolTable<>();
    protected DeclarationChecker declarationChecker = new DeclarationChecker();
    protected TypeChecker typeChecker = new TypeChecker();
}

program
    :   ^(PROGRAM
            { symtab.openScope(); }
            instructions
            { symtab.closeScope(); })
    ;

instructions
    :   (declaration | statement | expression)*
    ;

declaration
    :   constant_declaration | var_declaration
    ;

constant_declaration
    :   ^(CONST t=type id=IDENTIFIER expr=expression) {
            declarationChecker.processConstantDeclaration((DeclNode)$CONST, $id, t, symtab);
```

```
            typeChecker.processConstantAssignmentExpression((DeclNode)$CONST, $id, expr,
                symtab);
        }
    ;

var_declaration
    :   ^(VAR t=type id=IDENTIFIER) {
            declarationChecker.processVariableDeclaration((DeclNode)$VAR, $id, t, symtab);
        }
    ;

statement
    :   (if_statement | while_statement)
    ;

if_statement
    :   ^(  IF              { symtab.openScope();                                    }
            expr=expression { typeChecker.processIfStatement((StilNode)$IF, expr); }
            instructions    { symtab.closeScope();                                   }
         (  ELSE            { symtab.openScope();                                    }
            instructions    { symtab.closeScope();                                   })?)
    ;

while_statement
    :   ^(  WHILE           { symtab.openScope();                                       }
            expr=expression { typeChecker.processWhileStatement((StilNode)$WHILE, expr); }
            instructions    { symtab.closeScope();                                      })
    ;

print_statement returns [EntityType entityType = null;]
    :   ^(  node=PRINT
            t=expression { entityType = typeChecker.processPrintStatement((ExprNode)$node, t
                ); }
         (  t=expression { entityType = typeChecker.processMultiplePrintStatement((ExprNode)
             $node, t); })*)
    ;


read_statement returns [EntityType entityType = null;]
    :   ^(  node=READ
            id=IDENTIFIER { entityType = declarationChecker.retrieveDeclaration((ExprNode)
                $node, $id, symtab, true); }
        (   id=IDENTIFIER { entityType = declarationChecker.retrieveMultipleDeclaration((
            ExprNode)$node, $id, symtab, true); })*)
    ;

compound_expression returns [EntityType entityType = null;]
    :   ((declaration | statement)* expr=expression { entityType = expr; })*
    ;

//setEntityType

closed_compound_expression returns [EntityType entityType = null;]
    :   ^(  node=COMPOUND_EXPR      {   symtab.openScope();     }
            c=compound_expression   {   entityType = c; ((ExprNode)$node).setEntityType(
                entityType);
                                        symtab.closeScope();    })
    ;

expression returns [EntityType entityType = null;]
    :   p=print_statement               { entityType = p;}
    |   r=read_statement                { entityType = r;}
    |   o=operand                       { entityType = o; }
    |   c=closed_compound_expression    { entityType = c; }
    |   ^(node=BECOMES id=IDENTIFIER t1=expression)  { entityType = typeChecker.
```

```
        processVariableAssignmentExpression((ExprNode)$node, $id, t1, symtab); }
    |   ^(node=OR t1=expression t2=expression)       { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.OR, t1, t2); }
    |   ^(node=AND t1=expression t2=expression)      { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.AND, t1, t2); }
    |   ^(node=LT t1=expression t2=expression)       { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.LT, t1, t2); }
    |   ^(node=LTE t1=expression t2=expression)      { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.LTE, t1, t2); }
    |   ^(node=GT t1=expression t2=expression)       { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.GT, t1, t2); }
    |   ^(node=GTE t1=expression t2=expression)      { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.GTE, t1, t2); }
    |   ^(node=EQ t1=expression t2=expression)       { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.EQ, t1, t2); }
    |   ^(node=NEQ t1=expression t2=expression)      { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.NEQ, t1, t2); }
    |   ^(node=PLUS t1=expression t2=expression)     { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.PLUS, t1, t2); }
    |   ^(node=MINUS t1=expression t2=expression)    { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.MINUS, t1, t2); }
    |   ^(node=DIVIDE t1=expression t2=expression)   { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.DIVIDE, t1, t2); }
    |   ^(node=MULTIPLY t1=expression t2=expression) { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.MULTIPLY, t1, t2); }
    |   ^(node=MODULO t1=expression t2=expression)   { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.MODULO, t1, t2); }
    |   ^(node=UNARY_PLUS t1=expression)             { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.UNARY_PLUS, t1); }
    |   ^(node=UNARY_MINUS t1=expression)            { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.UNARY_MINUS, t1); }
    |   ^(node=UNARY_NOT t1=expression)              { entityType = typeChecker.
        processLogicExpression((LogicExprNode)$node, Operator.NOT, t1); }
    ;


operand returns [EntityType entityType = null;]
    :   id=IDENTIFIER   { entityType = declarationChecker.retrieveDeclaration($id, $id,
        symtab, false); ((IdNode)$id).setEntityType(entityType);}
    |   node=(TRUE | FALSE)  { entityType = EntityType.BOOL; ((LiteralNode)$node).
        setEntityType(entityType); }
    |   node=CHAR_LITERAL    { entityType = EntityType.CHAR; ((LiteralNode)$node).
        setEntityType(entityType); }
    |   node=INT_LITERAL     { entityType = EntityType.INT; ((LiteralNode)$node).
        setEntityType(entityType); }
    ;

type returns [EntityType entityType = null;]
    :   BOOL    { entityType = EntityType.BOOL; }
    |   CHAR    { entityType = EntityType.CHAR; }
    |   INT     { entityType = EntityType.INT; }
    ;
```

# Bijlage D

# Codegeneratorspecificatie

Listing D.1: ANTLR-specificatie voor de codegenerator.

```
tree grammar StilGenerator;

options {
    tokenVocab=Stil;                    // import tokens from Stil.tokens
    ASTLabelType = StilNode;            // AST nodes are of type StilNode
}

@header {
    package vb.stil;
    import  vb.stil.codegenerator.*;
    import  vb.stil.symtab.*;
    import  vb.stil.tree.*;
    import  vb.stil.exceptions.*;
    import  org.stringtemplate.v4.*;
}

@rulecatch {
    catch (RecognitionException e) {
        throw e;
    }
}

@members {
    protected CodeGenerator codeGenerator = new CodeGenerator();
}

program[int numOps, int locals] returns [ST template = null]
    :   ^(PROGRAM
            {codeGenerator.openScope();}
            instruction*) { template = codeGenerator.processProgram((StilNode)$PROGRAM,
                numOps, locals); }
            {codeGenerator.closeScope();}
    ;

instruction returns [ST template = null]
    :   st=declaration { template = st; }
    |   st=statement   { template = st; }
    |   st=expression  { template = st; }
    ;

declaration returns [ST template = null]
    :   st=var_declaration      { template = st; }
    |   st=constant_declaration { template = st; }
    ;

constant_declaration returns [ST template = null]
```

```
    :   ^(CONST type id=IDENTIFIER expression) {
            template = codeGenerator.processConstDeclaration((DeclNode)$CONST, (IdNode)$id);
                ((DeclNode)$CONST).setST(template);
        }
    ;

var_declaration returns [ST template = null]
    :   ^(VAR type id=IDENTIFIER) {
            template = codeGenerator.processVarDeclaration((DeclNode)$VAR, (IdNode)$id); ((
                DeclNode)$VAR).setST(template);
        }

    ;

statement returns [ST template = null]
    :   st=if_statement    { template = st; }
    |   st=while_statement { template = st; }
    ;

if_statement returns [ST template = null]
    @init { List<ST> ifInstructions = new ArrayList<>(), elseInstructions = new ArrayList
        <>(); }
    :   ^(  IF             { codeGenerator.openScope();  }
            expr=expression
            (i=instruction  { ifInstructions.add(i);      } )*
                            { codeGenerator.closeScope(); }
         (  ELSE            { codeGenerator.openScope();  }
            (i=instruction  { elseInstructions.add(i);    } )*
                            { codeGenerator.closeScope(); } )?) { template = codeGenerator.
                                processIfStatement((StilNode)$IF, ifInstructions,
                                elseInstructions); }
    ;

while_statement returns [ST template = null]
    @init { List<ST> instructions = new ArrayList<>(); }
    :   ^(  WHILE          { codeGenerator.openScope();  }
            expr=expression
            (i=instruction  { instructions.add(i);      } )*
                            { codeGenerator.closeScope(); }) { template = codeGenerator.
                                processWhileStatement((StilNode)$WHILE, instructions); }
    ;

print_statement returns [ST template = null]
    :   ^(PRINT expression+) { template = codeGenerator.processPrintStatement((ExprNode)
        $PRINT); }
    ;

read_statement returns [ST template = null;]
    :   ^(READ IDENTIFIER+) { template = codeGenerator.processReadStatement((ExprNode)$READ)
        ; }
    ;

closed_compound_expression returns [ST template = null]
    :   ^(COMPOUND_EXPR { codeGenerator.openScope(); }
            ((declaration | statement)* expr=expression)*
            { template = codeGenerator.processCompoundExpression((StilNode)$COMPOUND_EXPR);
              codeGenerator.closeScope(); ((ExprNode)$COMPOUND_EXPR).setST(template);}
        )
    ;

expression returns [ST template = null]
    :   st=print_statement                { template = st; }
    |   st=read_statement                 { template = st; }
    |   st=operand                        { template = st; }
    |   st=closed_compound_expression     { template = st; }
```

```
    |   ^(BECOMES IDENTIFIER expression)  { template = codeGenerator.becomes((ExprNode)
        $BECOMES); ((ExprNode)$BECOMES).setST(template);  }
    |   ^(node=OR          expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node);  }
    |   ^(node=AND         expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=LT          expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=LTE         expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=GT          expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=GTE         expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=EQ          expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=NEQ         expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=PLUS        expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=MINUS       expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=DIVIDE      expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=MULTIPLY    expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=MODULO      expression expression) { template = codeGenerator.
        processBinaryLogicExpression((LogicExprNode)$node); }
    |   ^(node=UNARY_PLUS  expression)            { template = codeGenerator.
        processUnaryLogicExpression( (LogicExprNode)$node); }
    |   ^(node=UNARY_MINUS expression)            { template = codeGenerator.
        processUnaryLogicExpression( (LogicExprNode)$node); }
    |   ^(node=UNARY_NOT   expression)            { template = codeGenerator.
        processUnaryLogicExpression( (LogicExprNode)$node); }
    ;

operand returns [ST template = null]
    :   id=IDENTIFIER       { template = codeGenerator.processIdOperand((IdNode)$id); ((
        IdNode)$id).setST(template); }
    |   v=(TRUE | FALSE)    { template = codeGenerator.processBoolLiteral((LiteralNode)$v);
        ((LiteralNode)$v).setST(template); }
    |   v=CHAR_LITERAL      { template = codeGenerator.processCharLiteral((LiteralNode)$v);
        ((LiteralNode)$v).setST(template); }
    |   v=INT_LITERAL       { template = codeGenerator.processIntLiteral((LiteralNode)$v);
        ((LiteralNode)$v).setST(template); }
    ;

type
    :   BOOL
    |   CHAR
    |   INT
    ;
```

# Bijlage E

# Jasmin Template

Listing E.1: StringTemplate bestand voor Jasmin

```
group stil;

type_map ::= [
  "int"  : "I",
  "bool" : "Z",
  "char" : "C"
]

type_scanner_map ::= [
  "int"  : "Int",
  "bool" : "Boolean",
  "char" : ""
]

type_scan_templ ::= [
  "int"  : "scan_int",
  "bool" : "scan_bool",
  "char" : "scan_char"
]

bool_map ::= [
  "true"  : "1",
  "false" : "0"
]

logic_operators ::= [
  "lt"  : "if_icmplt",
  "lte" : "if_icmple",
  "gt"  : "if_icmpgt",
  "gte" : "if_icmpge",
  "eq"  : "if_icmpeq",
  "neq" : "if_icmpne"
]

binary_operators ::= [
  "divide":   "idiv",
  "minus":    "isub",
  "or":       "ior",
  "modulo":   "irem",
  "multiply": "imul",
  "plus":     "iadd",
  "and":      "iand"
]

program(maxStackDepth, maxLocals, instructions) ::= <<
```

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

  .method public static main([Ljava/lang/String;)V
    .limit stack <maxStackDepth>
    .limit locals <maxLocals>
    new java/util/Scanner
    dup
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/util/Scanner/\<init>(Ljava/io/InputStream;)V
    astore 0 ; scanner
    <instructions ; separator="\npop\n">
    pop
    aload 0 ; scanner
    invokevirtual java/util/Scanner/close()V
    return
  .end method
>>


compound_expression(instructions) ::= <<
<instructions ; separator="\npop\n">
>>


becomes(varnum, expression, id, type) ::= <<
<expression>
dup
istore <varnum> ; <type> <id>
>>


idOperand(id, varnum, type) ::= <<
iload <varnum> ; <type> <id>
>>


boolLiteral(value) ::= <<
bipush <bool_map.(value)>
>>

charLiteral(value) ::= <<
bipush <value>
>>


intLiteral(value) ::= <<
bipush <value>
>>


not(expr, label1, label2) ::= <<
<expr>
ifeq L<label1>
bipush 0
goto L<label2>
L<label1>:
bipush 1
L<label2>:
>>
```

```
unary_minus(expr) ::= <<
<expr>
ineg
>>


unary_plus(expr) ::= <<
<expr>
>>

comparison(expr1, expr2, label1, label2, operator) ::= <<
<expr1>
<expr2>
<logic_operators.(operator)> L<label1>
bipush 0
goto L<label2>
L<label1>:
bipush 1
L<label2>:
>>


binary_operator(expr1,expr2,operator) ::= <<
<expr1>
<expr2>
<binary_operators.(operator)>
>>


if_statement(expr, if_instructions, else_instructions, label1, label2) ::= <<
<expr>
ifeq L<label1>
<if_instructions:{inst| <inst><\n>pop};separator="\n">
goto L<label2>
L<label1>:
<else_instructions:{inst| <inst><\n>pop};separator="\n">
L<label2>:
iconst_0
>>


while_statement(expr, instructions, label1, label2) ::= <<
L<label1>:
<expr>
ifeq L<label2>
<instructions:{inst| <inst><\n>pop};separator="\n">
goto L<label1>
L<label2>:
iconst_0
>>


print(expression,type) ::= <<
<expression>
dup
getstatic java/lang/System/out Ljava/io/PrintStream;
swap
invokevirtual java/io/PrintStream/println(<type_map.(type)>)V
>>


printMultiple(statements) ::= <<
<statements ; separator="\npop\n">
```

```
>>


scan_int(label1, label2) ::= <<
aload 0
invokevirtual java/util/Scanner/hasNextInt()Z
ifeq L<label1>
aload 0 ; scanner
invokevirtual java/util/Scanner/nextInt()I
goto L<label2>
L<label1>:
iconst_0
L<label2>:
>>


scan_bool(label1, label2) ::= <<
aload 0
invokevirtual java/util/Scanner/hasNextBoolean()Z
ifeq L<label1>
aload 0 ; scanner
invokevirtual java/util/Scanner/nextBoolean()Z
goto L<label2>
L<label1>:
iconst_0
L<label2>:
>>


scan_char(label1, label2) ::= <<
aload 0
invokevirtual java/util/Scanner/hasNext()Z
ifeq L<label1>
aload 0 ; scanner
invokevirtual java/util/Scanner/next()Ljava/lang/String;
iconst_0
invokevirtual java/lang/String/charAt(I)C
goto L<label2>
L<label1>:
iconst_0
L<label2>:
>>


read(var) ::= <<
iconst_0
istore <var.varnum> ; <var.id>
aload 0 ; scanner
invokevirtual java/util/Scanner/hasNextLine()Z
ifeq L<var.label1>
<(type_scan_templ.(var.type))(var.label2, var.label3)>
aload 0 ; scanner
invokevirtual java/util/Scanner/nextLine()Ljava/lang/String;
pop
istore <var.varnum> ; <var.id>
L<var.label1>:
>>


readMultiple(variables) ::= <<
<variables:read(); separator="\n">
iload <first(variables).varnum> ; <first(variables).id>
>>
```

# Bijlage F

# Testprogramma

In deze bijlage wordt een groot testprogramma met gegenereerde Jasmin bytecode gepresenteerd.

## F.1 Brontaal

Listing F.1: Testprogramma in Stil.

```
var int ivar;

ivar := {
    var int ivar1;
    var int ivar2;

    read(ivar1, ivar2);
    print(ivar1, ivar2);

    const int iconst1 := 1;
    const int iconst2 := 2;

    ivar2 := ivar1 := +16 + 2 * -8;

    print(  ivar1 < ivar2 && iconst1 <= iconst2,
            iconst1 * iconst2 > ivar2 - ivar1);

    ivar1 < read(ivar2) && iconst1 <= iconst2;

    ivar2 := print(ivar2) + 1;
} + 1;

var bool bvar;

bvar := {
    var bool bvar;

    read(bvar);
    print(bvar);

    bvar := 12 / 5 * 5 + 12 % 5 == 12 && 6 >= 6;

    const bool bconst := true;

    print(!false && bvar == bconst || true <> false);
} && true;

var char cvar;

cvar := {
```

```
    var char cvar1;
    var char cvar2;

    read(cvar1);

    const char cconst := 'c';

    cvar2 := 'z';

    print('a', cvar1 == cconst && (cvar2 <> 'b' || !true));

    'b';
};

print(ivar, bvar, cvar);

if(bvar)  { print(true);  }
if(!bvar) { print(false); }

var int i;
i := 0;

const int j := 1; // loop till j and comment test

while(i <= j) {
    var int foo;

    if(i == j) {
        print(true);
    } else {
        print(false);
    }

    i := i + 1;
}

while((i-1) > j) {
    print(true);

    var int bar;
}
```

## F.2 Gegenereerde Jasmin bytecode

Listing F.2: Jasmin bytecode van het testprogramma.

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

  .method public static main([Ljava/lang/String;)V
    .limit stack 100
    .limit locals 100
    new java/util/Scanner
    dup
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
    astore 0 ; scanner
    iconst_0
    istore 2 ; ivar1
    aload 0 ; scanner
```

```
    invokevirtual java/util/Scanner/hasNextLine()Z
    ifeq L0
    aload 0
    invokevirtual java/util/Scanner/hasNextInt()Z
    ifeq L1
    aload 0 ; scanner
    invokevirtual java/util/Scanner/nextInt()I
    goto L2
    L1:
    iconst_0
    L2:
    aload 0 ; scanner
    invokevirtual java/util/Scanner/nextLine()Ljava/lang/String;
    pop
    istore 2 ; ivar1
    L0:
    iconst_0
    istore 3 ; ivar2
    aload 0 ; scanner
    invokevirtual java/util/Scanner/hasNextLine()Z
    ifeq L3
    aload 0
    invokevirtual java/util/Scanner/hasNextInt()Z
    ifeq L4
    aload 0 ; scanner
    invokevirtual java/util/Scanner/nextInt()I
    goto L5
    L4:
    iconst_0
    L5:
    aload 0 ; scanner
    invokevirtual java/util/Scanner/nextLine()Ljava/lang/String;
    pop
    istore 3 ; ivar2
    L3:
    iload 2 ; ivar1
    pop
    iload 2 ; INT ivar1
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V
    pop
    iload 3 ; INT ivar2
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V
    pop
    bipush 1
    dup
    istore 4 ; INT iconst1
    pop
    bipush 2
    dup
    istore 5 ; INT iconst2
    pop
    bipush 16
    bipush 2
    bipush 8
    ineg
    imul
    iadd
    dup
    istore 2 ; INT ivar1
```

```
    dup
    istore 3 ; INT ivar2
    pop
    iload 2 ; INT ivar1
    iload 3 ; INT ivar2
    if_icmplt L6
    bipush 0
    goto L7
    L6:
    bipush 1
    L7:
    iload 4 ; INT iconst1
    iload 5 ; INT iconst2
    if_icmple L8
    bipush 0
    goto L9
    L8:
    bipush 1
    L9:
    iand
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(Z)V
    pop
    iload 4 ; INT iconst1
    iload 5 ; INT iconst2
    imul
    iload 3 ; INT ivar2
    iload 2 ; INT ivar1
    isub
    if_icmpgt L10
    bipush 0
    goto L11
    L10:
    bipush 1
    L11:
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(Z)V
    pop
    iload 2 ; INT ivar1
    iconst_0
    istore 3 ; ivar2
    aload 0 ; scanner
    invokevirtual java/util/Scanner/hasNextLine()Z
    ifeq L12
    aload 0
    invokevirtual java/util/Scanner/hasNextInt()Z
    ifeq L13
    aload 0 ; scanner
    invokevirtual java/util/Scanner/nextInt()I
    goto L14
    L13:
    iconst_0
    L14:
    aload 0 ; scanner
    invokevirtual java/util/Scanner/nextLine()Ljava/lang/String;
    pop
    istore 3 ; ivar2
    L12:
    iload 3 ; ivar2
    if_icmplt L15
    bipush 0
```

```
150     goto L16
        L15:
        bipush 1
        L16:
        iload 4 ; INT iconst1
155     iload 5 ; INT iconst2
        if_icmple L17
        bipush 0
        goto L18
        L17:
160     bipush 1
        L18:
        iand
        pop
        iload 3 ; INT ivar2
165     dup
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(I)V
        bipush 1
170     iadd
        dup
        istore 3 ; INT ivar2
        bipush 1
        iadd
175     dup
        istore 1 ; INT ivar
        pop
        iconst_0
        istore 3 ; bvar
180     aload 0 ; scanner
        invokevirtual java/util/Scanner/hasNextLine()Z
        ifeq L19
        aload 0
        invokevirtual java/util/Scanner/hasNextBoolean()Z
185     ifeq L20
        aload 0 ; scanner
        invokevirtual java/util/Scanner/nextBoolean()Z
        goto L21
        L20:
190     iconst_0
        L21:
        aload 0 ; scanner
        invokevirtual java/util/Scanner/nextLine()Ljava/lang/String;
        pop
195     istore 3 ; bvar
        L19:
        iload 3 ; bvar
        pop
        iload 3 ; BOOL bvar
200     dup
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(Z)V
        pop
205     bipush 12
        bipush 5
        idiv
        bipush 5
        imul
210     bipush 12
        bipush 5
        irem
        iadd
        bipush 12
```

```
215     if_icmpeq L22
        bipush 0
        goto L23
        L22:
        bipush 1
220     L23:
        bipush 6
        bipush 6
        if_icmpge L24
        bipush 0
225     goto L25
        L24:
        bipush 1
        L25:
        iand
230     dup
        istore 3 ; BOOL bvar
        pop
        bipush 1
        dup
235     istore 4 ; BOOL bconst
        pop
        bipush 0
        ifeq L26
        bipush 0
240     goto L27
        L26:
        bipush 1
        L27:
        iload 3 ; BOOL bvar
245     iload 4 ; BOOL bconst
        if_icmpeq L28
        bipush 0
        goto L29
        L28:
250     bipush 1
        L29:
        iand
        bipush 1
        bipush 0
255     if_icmpne L30
        bipush 0
        goto L31
        L30:
        bipush 1
260     L31:
        ior
        dup
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
265     invokevirtual java/io/PrintStream/println(Z)V
        bipush 1
        iand
        dup
        istore 2 ; BOOL bvar
270     pop
        iconst_0
        istore 4 ; cvar1
        aload 0 ; scanner
        invokevirtual java/util/Scanner/hasNextLine()Z
275     ifeq L32
        aload 0
        invokevirtual java/util/Scanner/hasNext()Z
        ifeq L33
        aload 0 ; scanner
```

```
    invokevirtual java/util/Scanner/next()Ljava/lang/String;
    iconst_0
    invokevirtual java/lang/String/charAt(I)C
    goto L34
    L33:
    iconst_0
    L34:
    aload 0 ; scanner
    invokevirtual java/util/Scanner/nextLine()Ljava/lang/String;
    pop
    istore 4 ; cvar1
    L32:
    iload 4 ; cvar1
    pop
    bipush 99
    dup
    istore 6 ; CHAR cconst
    pop
    bipush 122
    dup
    istore 5 ; CHAR cvar2
    pop
    bipush 97
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
    pop
    iload 4 ; CHAR cvar1
    iload 6 ; CHAR cconst
    if_icmpeq L35
    bipush 0
    goto L36
    L35:
    bipush 1
    L36:
    iload 5 ; CHAR cvar2
    bipush 98
    if_icmpne L37
    bipush 0
    goto L38
    L37:
    bipush 1
    L38:
    bipush 1
    ifeq L39
    bipush 0
    goto L40
    L39:
    bipush 1
    L40:
    ior
    iand
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(Z)V
    pop
    bipush 98
    dup
    istore 3 ; CHAR cvar
    pop
    iload 1 ; INT ivar
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
```

```
345     swap
        invokevirtual java/io/PrintStream/println(I)V
        pop
        iload 2 ; BOOL bvar
        dup
350     getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(Z)V
        pop
        iload 3 ; CHAR cvar
355     dup
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(C)V
        pop
360     iload 2 ; BOOL bvar
        ifeq L41
        bipush 1
        dup
        getstatic java/lang/System/out Ljava/io/PrintStream;
365     swap
        invokevirtual java/io/PrintStream/println(Z)V
        pop
        goto L42
        L41:
370     L42:
        iconst_0
        pop
        iload 2 ; BOOL bvar
        ifeq L43
375     bipush 0
        goto L44
        L43:
        bipush 1
        L44:
380     ifeq L45
        bipush 0
        dup
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
385     invokevirtual java/io/PrintStream/println(Z)V
        pop
        goto L46
        L45:
        L46:
390     iconst_0
        pop
        bipush 0
        dup
        istore 4 ; INT i
395     pop
        bipush 1
        dup
        istore 5 ; INT j
        pop
400     L53:
        iload 4 ; INT i
        iload 5 ; INT j
        if_icmple L47
        bipush 0
405     goto L48
        L47:
        bipush 1
        L48:
        ifeq L54
```

```
    iload 4 ; INT i
    iload 5 ; INT j
    if_icmpeq L49
    bipush 0
    goto L50
    L49:
    bipush 1
    L50:
    ifeq L51
    bipush 1
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(Z)V
    pop
    goto L52
    L51:
    bipush 0
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(Z)V
    pop
    L52:
    iconst_0
    pop
    iload 4 ; INT i
    bipush 1
    iadd
    dup
    istore 4 ; INT i
    pop
    goto L53
    L54:
    iconst_0
    pop
    L57:
    iload 4 ; INT i
    bipush 1
    isub
    iload 5 ; INT j
    if_icmpgt L55
    bipush 0
    goto L56
    L55:
    bipush 1
    L56:
    ifeq L58
    bipush 1
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(Z)V
    pop
    goto L57
    L58:
    iconst_0
    pop
    aload 0 ; scanner
    invokevirtual java/util/Scanner/close()V
    return
.end method
```

## F.3 Executievoorbeelden

**Voorbeeld 1.** In Tabel F.3 wordt de invoer, verwachte uitvoer en uitvoer gegeven voor het programma geïntroduceerd in Bijlage F.1. Hieruit blijkt dat het testprogramma naar verwachting werkt.

Tabel F.1: Overzicht van de in- en uitvoer

| **Invoer** | **Verwachte uitvoer** | **Uitvoer** |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 1 | false | false |
| false | true | true |
| c | 1 | 1 |
| | false | false |
| | true | true |
| | a | a |
| | true | true |
| | 3 | 3 |
| | true | true |
| | b | b |
| | true | true |
| | false | false |
| | true | true |

**Voorbeeld 2.** In Tabel F.3 wordt de invoer, verwachte uitvoer en uitvoer gegeven voor het programma geïntroduceerd in Bijlage F.1. Hieruit blijkt dat het testprogramma naar verwachting werkt.

Tabel F.2: Overzicht van de in- en uitvoer

| **Invoer** | **Verwachte uitvoer** | **Uitvoer** |
|---|---|---|
| 5 | 5 | 5 |
| 4 | 4 | 4 |
| 3 | false | false |
| true | true | true |
| z | 3 | 3 |
| | true | true |
| | true | true |
| | a | a |
| | false | false |
| | 5 | 5 |
| | true | true |
| | b | b |
| | true | true |
| | false | false |
| | true | true |