De Neptune programmeertaal

K. van Urk (s1076884)

Kornwerd 48,
1274 AL Huizen

M. Boersma (s1113429)

Jollemanhof 72,
1019 GW Amsterdam

1 Inhoudsopgave

2 Inleiding

De programmeertaal Neptune is ontwikkeld door Koen van Urk en Marcel Boersma. Het doel van Neptune is om de ontwerpers van deze taal inzicht te geven in het specificeren van een programmeertaal en het bouwen van de bijbehorende compiler.

De taal Neptune is ontwikkeld met behulp van een compiler generator, ANTLR3 (http://www.antlr3.org/ versie 3.5.2). De compiler gegenereerd door ANTLR kan de instructies uit de Neptune programmeertaal omzetten in instructies voor de Triangle Abstract Machine (TAM).

In dit verslag wordt dieper ingegaan op hoe de programmeertaal Neptune is vormgegeven. Zowel de syntax, context beperkingen en semantiek komen aan bod. Op deze manier wordt een compleet beeld van Neptune gegeven. Verder wordt er besproken hoe de eigen ontwikkelde Java programmatuur samenwerkt met de gegenereerde programmatuur van ANTLR om een zo volledig mogelijk beeld te geven van de compiler.

3 Beknopte beschrijving

Neptune is een taal met een aantal basis functionaliteiten, hieronder worden de verschillende mogelijkheden van Neptune nader toegelicht.

Waarden en types: er zijn vier verschillende soorten typen. Elke letterlijke waarde heeft een bijbehorend type. Hieronder staan de vier typen met enkele voorbeelden.

- *int*: numerieke waarden (bijvoorbeeld 3, 11, 1337)
- *char*: character waarden (bijvoorbaald '@', '5', 's')
- *char array*: string (bijvoorbeeld "Hello World")
- *bool*: waarheden (true en false)

In Neptune zijn types niet vrij uitwisselbaar. Sommige operaties vereisen een bepaald type om correct uitgevoerd te kunnen worden. In sectie 5.2 wordt hier uitgebreider op ingegaan.

Variabelen: entiteit die een waarde kan bevatten. Hierbij moet het type van de variabele compatibel zijn aan de toegewezen waarden. Tevens mag een entiteit een lijst van waarden bevatten (de één dimensionale array). Daarnaast is er een speciale categorie variabele, de constanten. Hiervan mag de waarde niet meer veranderen na het definiëren.

Scoping en codeblock: In Neptune wordt gebruik gemaakt van scoping. Dit houdt in dat bepaalde variabelen alleen binnen een bepaald gebied (*scope*) beschikbaar zijn. In sectie 5.2 wordt hier uitgebreider over gesproken.

Operaties: In Neptune worden de onderstaande operatoren ondersteund. In sectie 5.2 wordt uitgebreider gekeken naar de vereiste van de operanden en het resultaat van deze operaties.

- +, - en ! op één operand
- *, /, %, +, -, <, <=, >=, >, ==, !=, &&, || op twee operanden

Statements: Neptune heeft ondersteuning voor een aantal statements.

- If statement: hierbij kan code worden uitgevoerd indien er is voldaan aan gestelde voorwaarden.
- While statement: hierbij wordt de code uitgevoerd zolang de voorwaarde voldoet.
- For each: een variatie op de while statement. Hiermee kan eenvoudig door een lijst heengelopen worden.
- Function: hierbij kan code gegroepeerd worden en worden aangeroepen.

In appendix C is een voorbeeld programma gegeven met de bijbehorende in- en uitvoer.

4 Problemen en oplossingen

In deze sectie worden kort enkele problemen beschreven waar de makers tegen aanliepen tijdens het ontwerpen van de compiler. De gevonden oplossingen worden daar ook bij besproken.

4.1 Array problemen

4.1.1 Geheugen allocatie

Tijdens het toevoegen van de eendimensionale array functionaliteit zijn wij tegen een aantal problemen aangelopen. De types die werden gebruikt in de Neptune taal waren allemaal 1 woord groot om op te slaan op de stack. Bij arrays was er sprake een lijst van hetzelfde type, hierdoor was het niet mogelijk om 1 woord te reserveren op de stack. De oplossing is om in de symbol table bij te houden hoeveel geheugenruimte een variabele nodig heeft. Zo weet het programma later hoeveel ruimte er vrijgemaakt moet worden op de stack.

4.1.2 Uitlees volgorde

Verder waren er moeilijkheden met de volgorde waarmee de items in de array op de stack werden geplaatst. De eerste oplossing werkte volgens het FIFO principe, waarbij het eerste item als eerste op de stack werd gezet. Dit heeft als resultaat dat het laatste item dat op de stack is gezet het eerst van de stack af wordt gehaald, bijvoorbeeld bij een print operatie.

Hierdoor was de volgorde van de items in omgekeerde volgorde, “naam” werd bijvoorbeeld dan als “maan” getoond. Dit hebben we opgelost door in de Node klassen de inhoud van arrays in omgekeerde volgorde op de stack te zetten. Het laatste item staat dan dus als eerste op de stack. Hiermee zijn de problemen opgelost voor het uitlezen van de array items.

4.2 Variabelen

In de meeste programma’s wordt veel gebruik gemaakt van variabelen. Omdat de variabele een waarde representeert is het noodzakelijk dat de waarde behorend bij de variabele geladen wordt. Hiervoor is het noodzakelijk om te weten op welke geheugenlocatie de bijbehorende waarde gevonden kan worden. Om dit op te lossen is er een *SymbolTable* bedacht, hierin wordt bijgehouden welke adres de variabele heeft. Ook heeft de Neptune taal ondersteuning voor scoping via de *SymbolTable* (zie secties 5.2 en 7.2). Na het sluiten van een *scope* worden variabelen die niet meer beschikbaar zijn verwijderd. Het geheugenadres kan dan ook opnieuw gebruikt worden.

4.3 Read en print statement met één argument

Read en *print* statement hebben afhankelijk van het aantal argumenten een andere functionaliteit. De taal specificeert dat wanneer één argument wordt meegegeven dit de waarde van dat argument de return waarde van de functie *print* en *read* wordt. Om deze functionaliteit te ondersteunen wordt in de *PrintNode* en *ReadNode* bijgehouden hoeveel argumenten er mee worden gegeven. Als dit één argument betreft wordt de expressie behorende bij het argument opnieuw uitgevoerd.

4.4 Functies

Bij het maken van functies liepen we tegen verschillende problemen aan. Hieronder staan er drie individueel besproken met gevonden oplossingen.

4.4.1 Geheugen adressering

Het adresseren van variabelen binnen een functie werkt anders omdat er een nieuwe *frame* op de stack wordt gezet. Voorheen werd het register van de *[SB]* (*stack base*) gebruikt als basis voor het adres van een variabele. Hierbij stond het de eerste variabele op de positie 0[SB]. Het register *[LB]* (*local base)* bood uitkomst. Echter, de eerste drie woorden op een frame zijn niet beschikbaar om variabelen in op te slaan. Dit betekent dat in de globale *scope* de eerste variabele op positie 0[LB] stond maar in een functie op 3[LB] moest komen te staan. Hiervoor is de *SymbolTable* aangepast om 3 op te tellen bij het adres wanneer een variabele binnen een functie gedefinieerd is.

4.4.2 Argumenten in verkeerde volgorde op de stack

De argumenten van een functie moeten op de stack worden geplaatst. Hierbij liepen we tegen hetzelfde probleem aan als bij de arrays. De waarden werden op de verkeerde volgorde op de stack gezet waardoor de argumenten van die functie verkeerde waarden kregen toegekend. Dit is opgelost door de argumenten in omgekeerde volgorde op de stack te zetten.

4.4.3 Functie identifiers

Functies hebben een identifier nodig zodat er later in het programma een aanroep gedaan kan worden. Deze identifiers zijn globaal gedefinieerd, maar mochten eerst niet dezelfde naam hebben als een variabele. Dit hebben wij opgelost door een functie prefix in de *SymbolTable* toe te voegen.

4.5 Schoonhouden van de stack

Bij veel operaties worden normaal gesproken waarden op de stack gezet die daarna niet meer worden gebruikt. Bijvoorbeeld bij het printen met een enkel argument en het toewijzen van een nieuwe waarde aan een variabele. Er waren twee opties om dit op te lossen. Achteraf verwijderen van deze waarden of deze waarden er nooit op zetten. Er is gekozen voor het tweede alternatief omdat het binnen onze structuur met de Node klassen beter is te implementeren en daarnaast ook efficiënter is om uit te voeren. Onze implementatie geeft door aan *child nodes* of het resultaat dat ze eventueel kunnen genereren gebruikt gaat worden. Als dit niet zo is zal een *child node* in de assembly generatie fase geen instructies aanmaken.

5 Syntax, context beperkingen en semantiek

In deze sectie is de formele grammaticale definitie van de taal Neptune weergegeven. Ook wordt er ingegaan op de context beperkingen en de semantiek van de taal Neptune.

5.1 Syntax

De grammatica is opgesteld in de **Extended Backus Naur Form**. Hier betekend '*?*' optioneel,'*' 0 of meer keer en '*+*' één of meer keer herhalen. Verder wordt er gebruikt gemaakt van de helpers als beschreven in de Lexer helpers sectie.

Eerst is de set met terminals weergegeven:

, ; { } [ ] ( ) =

Alle **vet** gedrukte woorden in de onderstaande grammatica behoren tot de set met terminals.

5.1.0.1 EBNF

program	::= lines **EOF**
lines	::= line+
line	::= expression **;** | declaration **;** | logic_statement | return_statement **;** | function_declaration
codeblock	::= **{** lines **}**
logic_statement	::= while_statement | foreach_statement | if_statement
while_statement	::= **WHILE (** expression **) {** lines **}**
foreach_statement	::= **FOREACH (** IDENTIFIER **IN** IDENTIFIER **) {** lines **}**
if_statement	::= **IF (** expression **) {** lines **}** (**ELSIF (** expression **) {** lines **}**)* (**ELSE {** lines **}**)?
print_statement	::= **PRINT (** expression (**,** expression)* **)**
read_statement	::= **READ (** expression (**,** expression)* **)**
declaration	::= type IDENTIFIER (**=** expression)? | **CONST** type IDENTIFIER **=** expression
function_declaration	::= **FUNCTION** type IDENTIFIER **(** (type IDENTIFIER (**,** type IDENTIFIER)*)? **) {** lines **}**
return_statement	::= **RETURN (** expression **)**
expression	::= assignment_expr
assignment_expr	::= or_expr (**=** assignment_expr)?
or_expr	::= and_expr (**OR** and_expr)*
and_expr	::= boolean_expr (**AND** boolean_expr)*
boolean_expr	::= plus_expr ((**LT** | **LT_EQ** | **GT** | **GT_EQ** | **EQ** | **NEQ**) plus_expr)*

```
plus_expr       ::= multi_expr ((PLUS | MINUS) multi_expr)*;
multi_expr      ::= unary_expr ((TIMES | DIVIDE | MOD) unary_expr)*
unary_expr      ::= operand
                  | MINUS operand
                  | PLUS operand
                  | NEGATE operand
operand         ::=  IDENTIFIER ( | [ expression ] | ( ( expression (, expression)*)? ))
                  | NUMBER
                  | ( assignment_expr )
                  | [ expression (, expression)* ]
                  | print_statement
                  | read_statement
                  | SIZEOF ( IDENTIFIER )
                  | (TRUE | FALSE)
                  | CHAR_LITERAL
                  | STRING_LITERAL
                  | codeblock
type            ::= INTEGER array_def?
                  | CHAR array_def?
                  | BOOLEAN array_def?
array_def       ::=  [ NUMBER ]
```

5.1.0.2 Lexer helpers

In de EBNF grammatica is verwezen naar helpers, bijvoorbeeld de IDENTIFIER. Hieronder is een formele definitie gegeven van de helpers in EBNF.

```
IDENTIFIER      ::=  LETTER (LETTER | DIGIT | '_')*
NUMBER          ::=  DIGIT
COMMENT         ::=  '//' .* '\n'
CHAR_LITERAL    ::=  '\'' ~'\'' '\''
STRING_LITERAL  ::=  '"' (~'"')+ '"'
WS              ::=  (' ' | '\t' | '\f' | '\r' | '\n')+
DIGIT           ::=  ('0'..'9') ;
LOWER           ::=  ('a'..'z') ;
UPPER           ::=  ('A'..'Z') ;
LETTER          ::=  LOWER | UPPER ;
```

5.2 context beperking

- Bij de *binary *, /, %, +, -* operatoren mogen de linker en rechter kant van het type *int* of *char* zijn. Indien het type char wordt gegeven wordt de numerieke waarde uit de ASCII

tabel gebruikt om de berekening uit te voeren. De operaties leveren altijd een *int* waarde terug.

- De typen int en char zijn uitwisselbaar (*compatibel)* bij gebruik als operanden voor berekeningen en functies.
- Bij de *unary +, -* operatoren moet de rechter waarde van het type *int* zijn. Het resultaat zal ook weer van het type *int* zijn.
- De *unary ! (binaire inverse)* operator vereist een rechter waarde van het type *bool* en zal ook een *bool* opleveren.
- De *<, <=, >=, >* operatoren vereisen aan de linker en rechter kant een waarde van het type *int* of *char*. De operatie zal een waarde van het type *bool* teruggeven.
- De *== (equal), != (not equal)* operatoren vereisen dat het linker type compatibel is met het rechter type. Het type moet een *int, bool, char of array* type zijn. De operatie zal een *bool* teruggeven.
- De *&&,* || operatoren vereisen een linker en rechter kant van het type *bool*. De operatie levert een type *bool*.
- Variabelen worden gedefinieerd met een type (*int, bool, char)*.
- Een variabele mag eenmaal gedefinieerd worden per scope level.
- Een variabele moet gedefinieerd zijn voordat deze gebruikt kan worden.
- Een variabele uit een hogere scope kan gebruikt worden in de scopes daaronder.
- Een constante variabele mag per scope level eenmaal gedefinieerd worden.
- Een constante variabele kan geen assigment krijgen.
- Assignment: een variabele is gedefinieerd met een type (*int, bool, char)* en vereist dat de waarde die wordt verbonden aan de variabele compatibel is.
- Een *if statement* heeft een expressie van het type *bool*. Dit geldt ook voor de elsif clausules indien deze aanwezig zijn. Verder is het vereist dat binnen elke clausule (if, elsif en else) minimaal één regel code staat. Een *if statement* geeft geen waarde terug.
- Een *while statement* vereist een expressie van het type *bool*. Verder moet de while clausule minimaal één regel code bevatten. De *while statement* geeft geen waarde terug.
- Een *foreach* vereist dat de *identifier* na de **IN** terminal een array is. Dan wordt automatisch de *identifier* vóór de terminal **IN** van hetzelfde type als de elementen in de array. Deze variabele heeft hetzelfde scope level als andere elementen in de *foreach* routine.
- Functies moeten een return type *bool, int* of *char* hebben.
- Een functie mag meerdere return statements bevatten. Echter, het is vereist dat de laatste regel van een functie ook een return statement is.
- De type van een functie moet compatibel zijn met het type van de return.
- Een return statement mag alleen in een functie voorkomen.
- Een functie heeft nul of meer argumenten van het type *int, bool* of *char*. Dit mogen ook arrays zijn van deze typen.
- Functies mogen alleen in de globale scope gedefinieerd worden.
- Een functie moet een unieke naam hebben ten opzichte van andere functies. Een functie mag wel de naam hebben van een variabele.
- Argumenten van een functie zijn alleen binnen de functie scope beschikbaar.

- Variabelen uit de globale scope zijn niet beschikbaar binnen een functie.
- Een array bevat het aantal elementen binnen de vierkante blokhaken.
- Een array is van het type *int, bool* of *char*.
- Array assignment: het type van de elementen moet compatibel zijn met het type van de array.
- Een *char* array mag een string als waarde hebben (bijvoorbeeld char[5] s = “lepel”).
- Het aantal elementen van een array definitie moet overeenkomen met het aantal waarden dat wordt toegekend aan de array variabele.
- De laatste regel van een *codeblock* wordt terug gegeven als waarde, of void indien de laatste regel geen waarde teruggeeft.
- Een *codeblock* bevat minimaal één regel code.
- Het *print statement* vereist een argument van het type *bool, int, char*. Het argument mag een array zijn van deze types. Indien het print statement maar een argument heeft dan geeft de operatie de waarde terug van dit argument. Indien er meerdere argumenten worden meegegeven dan geeft *print* geen waarde terug.
- Het *read statement* vereist een variabele van het type *int, char*. Het argument mag een array zijn van deze types. Indien het read statement maar een argument heeft dan geeft de operatie de waarde terug van dit argument. Indien er meerdere argumenten worden meegegeven dan geeft *read* geen waarde terug.
- Een string wordt gezien als een array van het type *char*. Het aantal elementen van de array moet overeen komen met het aantal characters in de string.
- Er kan verwezen worden naar een een specifiek element binnen een array door de variabele naam te volgen met blokhaken als volgt: variable[*expression*]. Expression dient van het type *int* te zijn. Let op: in dit geval is *char* niet compatibel met *int*.
- De taal Neptune is hoofdlettergevoelig.

De operatoren hebben een bepaalde prioriteit mee gekregen en deze is schematisch weergeven in de volgende tabel:

Prioriteit	Operator	Operand type	Result type
1	+, - (unary)	int	int
1	!	bool	bool
2	*, /, %	int, char	int
3	+,-	int, char	int
4	<, <=, >=, >	int	bool
4	==, !=	int, bool, char, array	bool
5	&&	bool	bool
6	||	bool	bool

5.3 Semantiek

- **while statement**: een while statement (WHILE expr c) heeft een expressie en bevat code. Indien de de expr tot true evalueert word de code c in het while block uitgevoerd. Na het uitvoeren van de code wordt de expressie wederom gecheckt. Dit proces herhaald zich tot dat de waarde false is.
- **foreach statement**: een (FOREACH id1 IN id2 c) wordt als volgt uitgevoerd. Het eerste element van id2 wordt opgeslagen in de locale variabele id1. Vervolgens wordt de code c uitgevoerd. Daarna wordt het volgende element van id2 opgeslagen in id1. En c weer uitgevoerd. Dit herhaald zich tot alle elementen uit id2 zijn geweest.
- **if statement:** een (IF expr1 c1 ELSIF expr2 c2 ELSE c3) voert de code uit indien de expressie behorende bij het code block tot *true* evalueert. Dus indien expr1 waar is wordt c1 uitgevoerd. Indien expr2 waar is wordt c2 uitgevoerd. Indien niks waar is wordt c3 uitgevoerd. De if statement voert het code block uit bij de eerste expressie die waar is. Dus indien expr1 en expr2 waar zijn wordt alleen c1 uitgevoerd.
- **Assignment:** (id = expr) waarbij id een variabele identifier is en expr een expressie. De waarde van de expressie wordt opgeslagen in de variabele id. In het verdere verloop van het programma word de waarde expr geladen indien id wordt gebruikt. In geval van multiple assignments geeft de huidige assignment de waarde terug op de stack.
- Een **declaration** (t id) hier wordt een variabele id gedefinieerd binnen het huidige scope level als type t.

- Een **assignment** (T[count] id = [E]) wordt uitgevoerd door de waarden in de volgorde van de lijst [E] in de elementen op te slaan van id. Dus id[0] is het eerste element van de waarden in de lijst [E].
- Een **declaration** (const T id = E): hier wordt de waarde E in de variabele id opgeslagen. Deze variabele kan in het verdere verloop van het programma niet meer worden aangepast.
- **Print assignment** *id=print(E)* wordt uitgevoerd door de waarde van E te laden de waarde weer te geven op het scherm. Hierna wordt deze waarde opgeslagen in *id.*
- **Print statement** *print(E1, E2, En...)*. Hierbij worden de expressies een voor een op het scherm weergegeven.
- **Read assignment** id1 = read(id2) wordt uitgevoerd door de input waarde van de gebruiker toe te kennen aan de variabele id2. Vervolgens wordt dezelfde waarde ook toegekend aan id1.
- **Read statement** read(id1, id2, id_n...) wordt uitgevoerd door de input waarde van de gebruiker toe te kennen aan de variabele zoals ze opgegeven zijn. Er wordt hierbij n maal een waarde uitgelezen.
- **Function call**: func_name(args); Hier wordt de functie *func_name* die eerder moet zijn gedefiniëerd aangeroepen met de argumenten *args* die worden meegegeven. Vervolgens wordt de code geëvalueerd en aan het einde wordt een waarde terug gegeven.
- **integer literal** wordt uitgevoerd door de letterlijke waarde terug te geven.
- **character literal** wordt uitgevoerd door de integer waarde van de letter uit de ASCII tabel terug te geven met als type *char*.
- **string literal** wordt uitgevoerd door een lijst met character waarden terug te geven.
- **boolean literal** wordt uitgevoerd door een 0 (false) of 1 (true) op terug te geven.

6 Vertaalregels

De volgende code templates zijn gemaakt om de Neptune language om te zetten in TAM Assembly code.

```
run[[P]] :=
        PUSH s                  // s is de benodigde ruimte voor variabelen op de stack
        evaluate[[P]]
        POP s
        HALT

elaborate[[var_ref]] :=
        LOAD(1) var_addr[LB]

elaborate[[literal]] :=
        LOADL value

evaluate [[while expression code]] :=
        label X:
        evaluate[[expression]]
        JUMPIF(0) Y[CB]
        evaluate[[code]]
        JUMP X[CB]
        label Y:

evaluate [[ [expr1, expr2, expr3, ..] ]] :=
        elaborate[[..]]
        elaborate[[expr3]]
        elaborate[[expr2]]
        elaborate[[expr1]]

address[[id]] :=
        LOADA id

store[[id]] :=
        STORE(1) id[LB]

evaluate[[ arr[index] ]] :=
        evaluate[[index]]
        LOADL arr.elemCount()
        CALL valid0:            // out of bounds check
        address[[arr]]
```

```
        evaluate[[index]]
        CALL add
        LOADI(1)

evaluate[[ arr[index] = expr ]] :=
        evaluate[[expr]]
        evaluate[[index]]
        LOADL arr.elemCount()
        CALL valid0:                    // out of bounds check
        address[[arr]]
        evaluate[[index]]
        CALL add
        STOREI(1)


evaluate [[foreach id in arr code]] :=
        LOADL 0
        label X:
        LOAD(1) -[ST]
        address[[arr]]
        CALL add
        store[[id]]
        evaluate[[code]]
        CALL inc
        LOAD(1) -[ST]
        JUMPIF(arr.elemCount()) Y[CB]
        JUMP X[CB]
        label Y:

evaluate[[IF expr1 c1 ELSIF expr2 c2 ELSE c3]] :=
        evaluate[[expr1]]
        JUMPIF(0) next1[CB]
        evaluate[[c1]]
        JUMP OUT[CB]
        label next1:
        evaluate[[expr2]]
        JUMPIF(0) next2[CB]
        evaluate[[c2]]
        JUMP OUT[CB]
        label next2:
        evaluate[[c3]]
        label OUT:
```

```
evaluate[[ def FUNCTION  code]] :=
        label function:
        PUSH argsize + local variables size
        evaluate[[code]]
        RETURN(result_size) argsize

evaluate[[ call FUNCTION args ]] :=
        elaborate[[args]]
        CALL(LB) functionlabel[CB]

evaluate[[ id = expr]] :=
        evaluate[[expr]]
        store[[id]]

evaluate[[ e + e2 ]] :=
        evaluate[[e]]
        evaluate[[e2]]
        CALL add

evaluate[[ e - e2]] :=
        evaluate[[e]]
        evaluate[[e2]]
        CALL sub

evaluate[[ e * e2]] :=
        evaluate[[e]]
        evaluate[[e2]]
        CALL mult

evaluate[[ e / e2]] :=
        evaluate[[e]]
        evaluate[[e2]]
        CALL div

evaluate[[ e % e2]] :=
        evaluate[[e]]
        evaluate[[e2]]
        CALL mod

evaluate[[!e]] :=
        evaluate[[e]]
        CALL not
```

```
evaluate[[ e || e2]] =
        evaluate[[e]]
        evaluate[[e2]]
        CALL or

evaluate[[ e && e2]] =
        evaluate[[e]]
        evaluate[[e2]]
        CALL and

evaluate[[ e < e2]] :=
        evaluate[[e]]
        evaluate[[e2]]
        CALL lt

evaluate[[ e <= e2]]:=
        evaluate[[e]]
        evaluate[[e2]]
        CALL le

evaluate[[ e > e2]]:=
        evaluate[[e]]
        evaluate[[e2]]
        CALL gt

evaluate[[ e >= e2]]:=
        evaluate[[e]]
        evaluate[[e2]]
        CALL ge

evaluate[[e == e2]]:=
        evaluate[[e]]
        evaluate[[e2]]
        CALL eq

evaluate[[e != e2]]:=
        evaluate[[e]]
        evaluate[[e2]]
        CALL ne

evaluate[[-e]] :=
        evaluate[[e]]
        CALL neg
```

```
evaluate[[print c]]:=
        evaluate[[c]]
        CALL put

evaluate[[print i]]:=
        evaluate[[i]]
        CALL putint

evaluate[[id = print c]]:=
        evaluate[[c]]
        CALL put
        evaluate[[c]]
        store[[id]]

evaluate[[id = print i]]:=
        evaluate[[i]]
        CALL putint
        evaluate[[i]]
        store[[id]]

evaluate[[read ch]] =
        LOADA ch
        CALL get

evaluate[[read int]] =
        LOADA int
        CALL getint

evaluate[[sizeof(arr)]] :=
        LOADL arr.elemCount()
```

7 Beschrijving programmatuur

In de volgende secties wordt beschreven hoe de programmatuur van de Neptune compiler is opgeleverd. In de algemene sectie wordt de directory structuur nader toegelicht en overige algemeenheden. De overige secties gaan specifieker in op de klassen die zijn toegevoegd aan de gegenereerde compiler.

7.1 Algemeen

De Neptune compiler heeft meerdere sub-directories. In de onderstaande lijst is een overzicht gegeven van de folders op hoofd niveau en waarvoor deze gebruikt worden.
De opgeleverde programmatuur heeft de volgende directory structuur:

- Neptune
    - antlr: hier staan alle grammatica bestanden van ANTLR.
    - bin: hierin staat bytecode van voorgedefinieerde klassen
        - antlr.jar (versie 3.5.2)
        - hamcrest.jar
        - junit.jar (versie 4)
    - docs: hierin zijn alle javadocs te vinden.
    - samples: hierin staan een tweetal voorbeeld programma's:
        - helloworld.npt
        - sort.npt
    - src: hier staat de java source code voor de Neptune compiler en de testfiles voor de test suite.

Verder is er een README.txt bijgevoegd met daarin de instructies om van start te gaan met de Neptune compiler. De Neptune.jar is de Neptune compiler zelf, en de TestSuite.jar is om de JUnit test suite uit te voeren.

7.2 SymbolTable en IdEntry

De SymbolTable slaat gegevens over variabelen en functies op in de vorm van IdEntry objecten. Deze IdEntry objecten bevatten de scope, het geheugenadres en andere nuttige informatie over een variabele. Om objecten toe te voegen wordt de methode ‘enter’ gebruikt. Voor het ophalen van een variabele wordt de functie ‘retrieve’ gebruikt.

Variabelen worden op een scope level gedefinieerd, dit wordt ondersteund door het ‘openScope’ en ‘closeScope’ commando in SymbolTable. Door deze informatie op te slaan in een SymbolTable is het in de checker mogelijk om bijvoorbeeld te controleren of een variabele gedefinieerd is voordat deze wordt gebruikt.

Verder houdt de SymbolTable ook de locale variabelen bij in functies en de functie definities zelf. Door middel van ‘openFunctionScope’ en ‘closeFunctionScope’ wordt bijgehouden of nieuwe toevoegingen en aanvragen gebeuren vanuit een functie of vanuit de globale scope.

7.3 Node klasse

Een Node klasse is een abstracte klasse gedefinieerd om voor elke mogelijke grammatica regel een ‘validate’ en ‘generate’ functie te schrijven. Deze functies voeren voor de betreffende node de controles voor context beperking uitgevoerd (*validate* functie) en genereert vervolgens assemnly code zoals beschreven in de gedefinieerde code templates (*generate* functie). Op deze manier is het heel overzichtelijk en staan zowel de controles als de code generatie voor een functionaliteit zoals een while statement in één bestand. De taal is hierdoor modulair opgebouwd.

Nodes hebben *child nodes*, waardoor een *parent node* niet de specifieke instructies hoeft te weten om een goed resultaat te bereiken. Een operatie zoals een toewijzing (*a = 5 * 3;*) hoeft bijvoorbeeld niet te weten hoe een vermenigvuldiging werkt, hij moet er alleen op kunnen vertrouwen dat het goede resultaat op de stack komt te staan om mee verder te kunnen.

Enkele zulkek Node klassen zijn voor de beeldvorming hieronder weergegeven. De complete lijst van Node klassen kan in de broncode gevonden worden in de neptune.node package.

- VarDeclarationNode
- BinaryPrimitiveOperatorNode
- WhileNode
- ForeachNode
- PrintNode
- FunctionDeclarationNode
- LiteralNode

7.4 Instruction en Program

De instruction klasse bevat de standaard TAM instructies, hierbij moet bijvoorbeeld gedacht worden aan LOADL, STORE, JUMP, CALL en LABEL. Via functies kunnen argumenten worden meegegeven zodat de juiste TAM instructie gegenereerd wordt.

De Program klasse slaat instructies op en zorgt dat deze in de juiste volgorde als assembly worden teruggegeven. Verder is Program verantwoordelijk voor het toevoegen van de *out of bounds* runtime validation functie voor arrays, en het bewaren van de *SymbolTable*.

7.5 Test omgeving

De package neptune.test bevat alle code die nodig is om de neptune compiler te testen. In sectie 8 wordt hier dieper op ingegaan. Binnen deze package is een folder genaamd sample met daarin alle testen die geschreven zijn voor de compiler.

Via de bijgeleverde TestSuite.jar kan de complete test suite worden uitgevoerd. Hierin worden 25 programma's uitgevoerd samengevat in 11 tests. De testprogramma's dienen in dezelfde folder als TestSuite.jar te staan, zoals standaard het geval is. De JAR file is als volgt te gebruiken:

```
java -jar TestSuite.jar
```

7.6 Neptune Compiler

Bijgeleverd is een executable JAR file met alle benodigdheden om een programma te valideren, compileren en uit te voeren. De klasse neptune.Neptune bevat code die de Lexer/Parser en TreeParser uitvoert en via de Nodes het programma controleert op context beperkingen. Verder bevat de JAR file een versie van de TAM Assembler en Interpreter zodat programma's kunnen worden ge-assembled en uitgevoerd.

De JAR file is als volgt te gebruiken (dit is ook te lezen in de bijgeleverde README.txt):

```
java -jar Neptune.jar validate program.npt
```
Voorbeeld commando voor het valideren van een Neptune programma uit program.npt

```
java -jar Neptune.jar compile program.npt
```
Voorbeeld commando voor het valideren en compileren van een Neptune programma uit program.npt

```
java -jar Neptune.jar run program.npt
```
Voorbeeld commando voor het valideren, compileren en uitvoeren van een Neptune programma uit program.npt

Als een van de twee benodigde parameters wordt weggelaten komt een hulp bericht op het scherm.

8 Testplan en test resultaten

8.1 Algemeen

Om ervoor te zorgen dat de Neptune compiler zo correct mogelijk is zijn er testen ontworpen. Let wel dat de testen geen volledige garantie geven op een foutloze compiler maar alleen bevestigen dat de test voorbeelden die zijn gemaakt functioneren naar behoren. Om het testplan zo volledig mogelijk te maken is er gekozen om deze op te splitsen in 3 categoriëen: *constructs*, *context* en *semantics*. In de volgende secties wordt een korte toelichting gegeven op elke soort test.

Voor alle functionaliteiten beschreven in de syntax, context beperkingen en semantiek worden testen ontworpen. Bij elke bijbehorende test wordt een verwachte output bestand aangemaakt. Indien de uitvoer van het programma overeenkomt met de inhoud van het output bestand dan is de test geslaagd.

8.1.1 Constructs

Dit zijn testen die zijn ontworpen om de constructie (syntax) van de taal te testen. Binnen Neptune zijn een aantal verplichtingen van de syntax en deze worden getest. Verder worden hier fouten zoals spellingfouten getest. Zie sectie 5.1 voor een overzicht van de grammaticale syntax.

8.1.2 Context beperkingen

Dit zijn testen ontworpen om de context beperkingen te testen. Hierbij moet gedacht worden aan bijvoorbeeld type errors maar ook scope levels en declaraties. Zie sectie 5.2 voor een overzicht van context beperkingen.

8.1.3 Semantics

Deze testen bevatten bestanden met correcte voorbeelden. Alle test bestanden moeten een correcte uitvoer leveren die geïnterpreteerd kan worden door de TAM machine.

Het testplan is geschreven in JUnit wat als voordeel heeft dat deze een snel overzicht geeft van de resultaten van de tests.

8.2 Ontwikkelde testen

Voor het testen is er een aparte package aangemaakt, de neptune.test package. Hierin is een folder aangemaakt genaamd sample met daarin drie folder (semantics, context, constructs). In elke van deze directories zijn *.npt bestanden en *.txt bestanden te vinden. De output bestanden (txt) geven de verwachte uitkomst van het bijbehorende npt bestand. Het soort test komt overeen met de foldernaam als beschreven in de algemene sectie van het testplan.

De volgende testen zijn ontwikkeld:

- **arraySyntax(constructs):** testen of de parser de incorrecte constructie van de een array afvangt.
- **commands(constructs):** testen of de parser de incorrecte opvolging van meerdere regels code afvangt.
- **spelling(constructs):** testen of de parser het verkeerd spellen van keywords afvangt.

- **array(context):** testen of het aantal elementen aan een array toekennen te groot is in vergelijking met het aantal element van een array.
- **basicexpression_scope(context):** testen of een incorrect scope gebruik wordt afgevangen door de checker.
- **basicexpression(context):** testen of een incorrect type assignment afgevangen wordt.
- **codeblock(context):** testen of een type assignment mismatch met een codeblock afgevangen wordt.
- **const_reassignment(context):** testen of een constant niet nogmaals een assignment krijgt.
- **foreach(context):** testen of het wordt afgevangen dat een foreach alleen werkt op een array.
- **function_global_scope(context):** testen of er wordt afgevangen dat een function alleen in de global scope gedefinieerd mag worden.
- **function_type(context):** testen of het return type compatibel is met het type van de functie.
- **function_scope(context):** testen of de argumenten van een functie alleen beschikbaar zijn in de functie scope.
- **ifstatement(context):** testen of een if statement expressie van het type bool is.
- **illegal_print_statement_assingment(context):** testen of een print met meerdere argumenten geen assignment doet.
- **incorrect_variable_assingment(context):** testen of een verkeerd type assignment van twee variabelen wordt afgevangen.

- **array(semantics):** testen van arrays. Bevat is: *index-into-array*, multiple assignments met arrays en het printen van arrays.
- **basicexpression(semantics):** een combinatie van mathematische expressies om de juistheid te testen van alle operaties.
- **codeblock(semantics):** testen of codeblocks de juiste waarden terug leveren en goede *scoping* hebben.
- **foreach(semantics):** testen of een foreach loop in goede volgorde door een array heen loopt.
- **function(semantics):** testen of (recursieve) functies goed werken. Bevat onder andere de recursieve definitie van Fibonacci.

- **ifstatement(semantics):** testen of een if statement in verschillende varianten goed wordt uitgevoerd.
- **insertionsort(semantics):** testen of een geheel programma met arrays, functies, geneste while lussen en een foreach statement goed uitgevoerd worden. In dit programma wordt insertion sort voor een array van 10 elementen gedefiniëerd en uitgevoerd.
- **whilestatement(semantics):** testen of de conditie van een while lus correct uitgevoerd wordt en of de lus meerdere malen herhaald wordt.

8.3 Test resultaten

Het test programma geeft aan dat testen correct zijn verlopen, dit houdt in dat de output van het gecompileerde programma overeenkomt met de verwachte output. Indien dit niet het geval is zal het test programma de huidige uitkomst laten zien en de verwachte uitkomst. Hieronder is een abstract voorbeeld van een verwachte output gemaakt.

```
Testing file: file.npt test type:directory
>>> OUTPUT DOES NOT MATCH EXPECTED OUTPUT! <<<
==========================================
Actual Output
==========================================
!dlroW olleH
==========================================
Expected
==========================================
Hello World!
==========================================
```

De test resultaten van de tests zoals beschreven in sectie 8.2 zijn te vinden in appendix B.

9 Conclusies

Neptune is een kleine programmeertaal voor de TAM machine. De taal ondersteund basis functionaliteiten zoals basic expressions maar ook if statements, while statements en functies. Dit zijn functionaliteiten die in de meeste programmeertalen voorkomen. Wel is het zo dat de ontwikkeling van veel programmeertalen niet heeft stilgestaan en daardoor hebben deze talen nog meer mogelijkheden om zichzelf uit te drukken in vergelijking met Neptune.

Op dit moment heeft Neptune nog niet de mogelijkheid voor dynamic array allocation en classes bijvoorbeeld. Dit zijn elementen van een taal die veel worden gebruikt in complexe computer systemen. Maar voor kleine basis programma's is Neptune een leuke taal om in te beginnen.

Het ontwikkelen van Neptune heeft veel inzicht gegeven in het bouwen van een compiler. De compiler is de brug van een programmeertaal naar machine instructies die door hardware uitgevoerd kan worden. De ontwikkeling geeft inzicht in het feit dat hardware maar beperkte mogelijk instructieset heeft maar dat de programmeur in het bedenken van complexe systemen hier niet door gelimiteerd hoeft te worden. Het ontwikkelen van een hogere taal geeft de programmeur de mogelijkheid om zich beter uit te drukken terwijl het uiteindelijke systeem wel uitvoerbaar is op een computer.

De inzichten verkregen tijdens dit project zijn zeer waardevol voor ons als programmeurs.

- Marcel Boersma en Koen van Urk

10 Appendix A

10.1 Lexer en Parser

```
grammar Neptune;

options {
   k=1;                          // LL(1) - do not use LL(*)
   language=Java;                // target language is Java (= default)
   output=AST;                   // build an AST
}

tokens {
            COLON      =  ':'    ;
            SEMICOLON  =  ';'    ;
            LPAREN     =  '('    ;
            RPAREN     =  ')'    ;
            COMMA      =  ','    ;
            LBRACKET           =  '['                ;
            RBRACKET           =        ']'                ;
            LCURLY             =        '{'                ;
            RCURLY             =        '}'                ;

   // operators
            BECOMES            =  '='    ;
            PLUS               =  '+'    ;
            MINUS              =  '-'    ;
            TIMES              =  '*'    ;
            DIVIDE             =  '/'    ;

            UNARY_MINUS        =        'u-'       ;
            UNARY_PLUS         =        'u+'       ;
            NEGATE             =        '!'        ;
            MOD                =        '%'        ;

            // strings
            DQUOTE             =        '"'        ;
            QUOTE              =        '\''       ;
            FALSE              =        'false'    ;
            TRUE               =        'true'     ;

   // keywords
            PROGRAM            =  'program'        ;
            BLOCK              =        'block'    ;
            VAR                =  'var'            ;
            CONST              =        'const'    ;
            PRINT              =  'print'          ;
            READ               =        'read'     ;
            FUNCTION           =        'function'          ;
            RETURN             =        'return'   ;
```

```
        SIZEOF          =       'sizeof'    ;

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

  // control
        IF              =  'if'      ;
        THEN            =  'then'    ;
        ELSE            =  'else'    ;
        ELSIF           =  'elsif'   ;
        DO              =  'do'      ;
        WHILE           =  'while'   ;
        FOREACH                 =  'foreach' ;
        IN              =  'in'      ;

  // binary-operators
        LT              =  '<'       ;
        LT_EQ           =  '<='      ;
        GT              =  '>'       ;
        GT_EQ           =  '>='      ;
        EQ              =  '=='      ;
        NEQ                     =  '!='      ;
        AND             =       '&&'            ;
        OR              =       '||'            ;
        ARRAY_SET       =       'array_set'     ;
        ARRAY_DEF       =       'array_def'     ;
        ATOMIC_VAR      =       'atomic'   ;
        ARRAY_VAR       =       'array_var'     ;
}


@lexer::header {
package neptune;
}

@header {
package neptune;
}


// Parser rules

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

lines
```

```
        :       line+
        ;

line
        :       expression SEMICOLON!
        |       declaration SEMICOLON!
        |       logic_statement
        |       return_statement SEMICOLON!
        |       function_declaration
        ;

codeblock
        :       LCURLY lines RCURLY
                ->      ^(BLOCK lines)
        ;

logic_statement
        :       while_statement
        |       foreach_statement
        |       if_statement
        ;



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

foreach_statement
        :       FOREACH^ LPAREN! IDENTIFIER IN! IDENTIFIER RPAREN! LCURLY! lines RCURLY!
        ;

if_statement
        :       IF^ LPAREN! expression RPAREN! LCURLY! lines RCURLY!
                (ELSIF LPAREN! expression RPAREN! LCURLY! lines RCURLY!)*
                (ELSE LCURLY! lines RCURLY!)?
        ;

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

read_statement
        :       READ^ LPAREN! expression (COMMA! expression)* RPAREN!
        ;

declaration
        :       type IDENTIFIER (BECOMES expression)?
                -> ^(VAR type IDENTIFIER (BECOMES expression)?)
        |       CONST^ type IDENTIFIER BECOMES expression
```

```
        ;

function_declaration
        : FUNCTION type IDENTIFIER LPAREN (type IDENTIFIER (COMMA type IDENTIFIER)*)? RPAREN LCURLY line+
RCURLY
                -> ^(FUNCTION type IDENTIFIER (type IDENTIFIER)* lines)
        ;

return_statement
        :       RETURN^ LPAREN! expression RPAREN!
        ;

expression
        : assignment_expr
        ;

assignment_expr
        :       or_expr (BECOMES^ assignment_expr)?
        ;

or_expr
        :       and_expr (OR^ and_expr)*
        ;

and_expr
        :       boolean_expr (AND^ boolean_expr)*
        ;

boolean_expr
        :       plus_expr ((LT^ | LT_EQ^ | GT^ | GT_EQ^ | EQ^ | NEQ^) plus_expr)*
        ;

plus_expr
        :       multi_expr ((PLUS^ | MINUS^) multi_expr)*
        ;

multi_expr
        :       unary_expr ((TIMES^ | DIVIDE^ | MOD^) unary_expr)*
        ;

unary_expr
        :       operand
        |       MINUS operand -> ^(UNARY_MINUS operand)
        |       PLUS operand -> ^(UNARY_PLUS operand)
        |       NEGATE^ operand
        ;

operand
   :  IDENTIFIER ( -> ^(ATOMIC_VAR IDENTIFIER) | LBRACKET expression RBRACKET -> ^(ARRAY_VAR IDENTIFIER expression)
| (LPAREN (expression (COMMA expression)*)? RPAREN -> ^(FUNCTION IDENTIFIER ^(ARRAY_SET expression*))))
```

```
        |   NUMBER
        |   LPAREN! assignment_expr RPAREN!
        |           LBRACKET expression (COMMA expression)* RBRACKET
                    ->          ^(ARRAY_SET expression+)
        |           print_statement
        |           read_statement
        |           SIZEOF^ LPAREN! IDENTIFIER RPAREN!
        |           (TRUE | FALSE)
        |           CHAR_LITERAL
        |           STRING_LITERAL
        |           codeblock
        ;

type
        :           INTEGER  array_def?
        |           CHAR array_def?
        |           BOOLEAN array_def?
        ;

array_def
        :            LBRACKET NUMBER RBRACKET
                    -> ^(ARRAY_DEF NUMBER)
        ;

// Lexer rules

IDENTIFIER
  :  LETTER (LETTER | DIGIT | '_')*
  ;

NUMBER
  :  DIGIT+
  ;


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

CHAR_LITERAL
        :           '\'' ~'\'' '\''
        ;

STRING_LITERAL
        :           '"' (~'"')+ '"'
        ;

WS
  :  (' ' | '\t' | '\f' | '\r' | '\n')+
```

```
        { $channel=HIDDEN; }
    ;

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

// EOF
```

10.2 Checker en generator

```
tree grammar NeptuneTree;

options{
        tokenVocab=Neptune;
        ASTLabelType=CommonTree;
}

@header{
        package neptune;

        import neptune.node.*;
        import neptune.assembly.Program;
}

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

@members {
        public ProgramNode rootNode;
}

program
        : ^(PROGRAM n=lines)
{ rootNode = new ProgramNode(n); }
        ;

lines returns [ List<Node> nodes = new ArrayList<Node>() ]
        : (n=line { nodes.add(n); })+
        ;
```

```
line returns [ Node node ]
        : n=expression
        { node = new BasicNode("expression", n); }
        | n=declaration
        { node = new BasicNode("declaration", n);  }
        | n=logic_statement
        { node = new BasicNode("logic", n); }
        | n=return_statement
{ node = new BasicNode("return", n); }
        ;

codeblock returns [ Node node ]
        : ^(BLOCK n=lines)
{ node = new CodeblockNode(n); }
        ;

logic_statement returns [ Node node ]
        : ^(WHILE ex=expression l=lines)                                                { node = new
WhileNode(ex, l); }
        | ^(FOREACH x=IDENTIFIER y=IDENTIFIER l=lines)                                  { node = new
ForeachNode($x.text, $y.text, l); }
// If statement
        | { List<Node> ifBlocks = new ArrayList<Node>(); Node elseNode = null; }
         ^(IF ex=expression l=lines                                                     { ifBlocks.add(new IfBlockNode(ex, l)); }
                (
                        ELSIF ex=expression l=lines                                     { ifBlocks.add(new IfBlockNode(ex,
l)); }
                )*
                (
                        ELSE l=lines                                                    { elseNode = new
ElseNode(l); }
                )?
         )
                        { node = new IfNode(ifBlocks, elseNode); }
        ;

print_statement returns [ Node node ]
        : { List<Node> expressions = new ArrayList<Node>(); }
                ^(PRINT
                         n1=expression { expressions.add(n1); }
                        (n2=expression { expressions.add(n2); })*
                )
                        { node = new PrintNode(expressions); }
        ;

read_statement returns [ Node node ]
        : { List<Node> expressions = new ArrayList<Node>(); }
                ^(READ
                         n1=variable_expression { expressions.add(n1); }
                        (n2=variable_expression { expressions.add(n2); })*
```

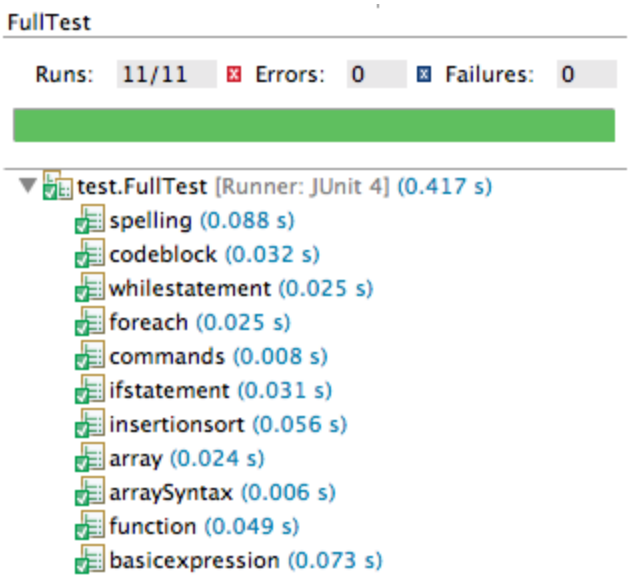
```
                )
                        { node = new ReadNode(expressions); }
        ;

declaration returns [ Node node ]
        : ^(VAR t=type id=IDENTIFIER (BECOMES ex=expression)?)                  { node = new
VarDeclarationNode($id.text, t, ex); }
        | ^(CONST t=type id=IDENTIFIER BECOMES ex=expression)                   { node = new
ConstDeclarationNode($id.text, t, ex); }
        | { List<Node> args = new ArrayList<Node>(); }
          ^(FUNCTION
                t1=type
                func=IDENTIFIER
                (
                        t=type id=IDENTIFIER                            { args.add(new VarDeclarationNode($id.text,
t, null)); }
                )*
                l=lines
        )
                        { node = new FunctionDeclarationNode($func.text, t1, args, l); }
        ;

return_statement returns [ Node node ]
        :       ^(RETURN ex=expression)
{ node = new ReturnNode(ex); }
        ;

expression returns [ Node node ]
        : n=operand
                { node = n; }
        | ^(BECOMES x=variable_use e1=expression)                                { node = new
BinaryAnyOperatorNode(Operator.BECOMES, x, e1); }
        | ^(AND e1=expression e2=expression)                                    { node = new
BinaryPrimitiveOperatorNode(Operator.AND, e1, e2); }
        | ^(OR e1=expression e2=expression)                                     { node =
new BinaryPrimitiveOperatorNode(Operator.OR, e1, e2); }
        | ^(LT e1=expression e2=expression)                                     { node = new
BinaryPrimitiveOperatorNode(Operator.LESS, e1, e2); }
        | ^(LT_EQ e1=expression e2=expression)                                  { node = new
BinaryPrimitiveOperatorNode(Operator.LESS_EQUAL, e1, e2); }
        | ^(GT e1=expression e2=expression)                                     { node = new
BinaryPrimitiveOperatorNode(Operator.GREATER, e1, e2); }
        | ^(GT_EQ e1=expression e2=expression)                                  { node = new
BinaryPrimitiveOperatorNode(Operator.GREATER_EQUAL, e1, e2); }
        | ^(EQ e1=expression e2=expression)                                     { node = new
BinaryAnyOperatorNode(Operator.EQUAL, e1, e2); }
        | ^(NEQ e1=expression e2=expression)                                    { node = new
BinaryAnyOperatorNode(Operator.NOT_EQUAL, e1, e2); }
        | ^(PLUS e1=expression e2=expression)                                   { node = new
BinaryPrimitiveOperatorNode(Operator.PLUS, e1, e2); }
```

```
        | ^(MINUS e1=expression e2=expression)                                    { node = new
BinaryPrimitiveOperatorNode(Operator.MINUS, e1, e2); }
        | ^(TIMES e1=expression e2=expression)                                  { node = new
BinaryPrimitiveOperatorNode(Operator.TIMES, e1, e2); }
        | ^(DIVIDE e1=expression e2=expression)                                 { node = new
BinaryPrimitiveOperatorNode(Operator.DIVIDE, e1, e2); }
        | ^(MOD e1=expression e2=expression)                                    { node = new
BinaryPrimitiveOperatorNode(Operator.MOD, e1, e2); }
        | ^(UNARY_MINUS e1=expression)                                                  { node =
new UnaryPrimitiveOperatorNode(Operator.UNARY_MINUS, e1); }
        | ^(UNARY_PLUS e1=expression)                                                   { node =
new UnaryPrimitiveOperatorNode(Operator.UNARY_PLUS, e1); }
        | ^(NEGATE e1=expression)                                                       { node =
new UnaryPrimitiveOperatorNode(Operator.UNARY_NEGATE, e1); }
        ;

operand returns [ Node node ]
        : ^(FUNCTION id=IDENTIFIER n=array_set)                                 { node = new
FunctionCallNode($id.text, n); } // fix
        | v=variable_expression
{ node = v; }
        | l=NUMBER
                { node = new LiteralNode($l.text, Node.type.INTEGER); }
        | n=array_set
        { node = n; }
        | l=TRUE
        { node = new LiteralNode($l.text, Node.type.BOOL); }
        | l=FALSE
        { node = new LiteralNode($l.text, Node.type.BOOL); }
        | l=CHAR_LITERAL
{ node = new LiteralNode($l.text, Node.type.CHAR); }
        | l=STRING_LITERAL
        { node = new LiteralNode($l.text, Node.type.CHAR); }
        | n=codeblock
        { node = n; }
        | n=print_statement
        { node = n; }
        | n=read_statement
        { node = n; }
        | ^(SIZEOF id=IDENTIFIER)                                                       { node =
new SizeOfNode($id.text); }
        ;

array_set returns [ Node node ]
        : { List<Node> elements = new ArrayList<Node>(); }
          ^(ARRAY_SET
                (
                        ex=expression                                   { elements.add(ex); }
                )*
```

```
            )
                        { node = new ArraySetNode(elements); }
        ;

variable_expression returns [ Node node ]
        :       ^(ATOMIC_VAR id=IDENTIFIER)
{ node = new VarNode($id.text); }
        |       ^(ARRAY_VAR id=IDENTIFIER ex=expression)                    { node = new
VarIndexedNode($id.text, ex); }
        ;

variable_use returns [ Node node ]
        :       ^(ATOMIC_VAR id=IDENTIFIER)
{ node = new VarNode($id.text); }
        |       ^(ARRAY_VAR id=IDENTIFIER ex=expression)                    { node = new
VarIndexedNode($id.text, ex); }
        ;

type returns [ Node node ]
        : INTEGER count=array_def?      { node = new TypeNode(Node.type.INTEGER, count); }
        | CHAR count=array_def?             { node = new TypeNode(Node.type.CHAR, count); }
        | BOOLEAN count=array_def?          { node = new TypeNode(Node.type.BOOL, count); }
        ;

array_def returns [int count = 0]
        : ^(ARRAY_DEF x=NUMBER) { count = Integer.parseInt($x.text); }
        ;
```

11 Appendix B

In de onderstaande afbeelding is een screenshot gemaakt van de JUnit test suite:



Verder is er in Appendix C een compleet programma met alle functionaliteit van Neptune gegeven met TAM Code en in- en uitvoer.

12 Appendix C

Hieronder is een voorbeeld programma geschreven. Dit programma sorteert een standaard array of een door de gebruiker ingevoerde array met behulp van insertion sort.

Code:

```
function int[10] sort(int[10] arr) {
	int a;
	int b;

	int i = 0;
	int j = 0;

	char newLine = 10;

	while(i < sizeof(arr)) {
		int x = arr[i];
		int j = i;
		while(j > 0 && arr[j-1] > x) {
			arr[j] = arr[j-1];
			j = j - 1;
		}
		arr[j] = x;

		i = i + 1;
	}

	return(arr);
}

function int print_array(int[10] arr) {
	int i = 0;
	char newLine = 10;
	print(newLine, '[');
	foreach(number in arr) {
		print(number);
```

```
                i = i + 1;

                if(i < sizeof(arr)) {
                        print(", ");
                }
        }
        print(']', newLine);
        return(1);
}

int[10] numbers = [2, 9, 17, 4, 8, 5, 2, 10, 11, 3];

print("Use default numbers? Type f to enter 10 numbers yourself. ");
char userInput;
read(userInput);

if(userInput == 'f') {
        int i = 0;
        while(i < 10) {
                read(numbers[i]);
                i = i + 1;
        }
}

print_array(numbers);
print("Starting sort");
numbers = sort(numbers);
print_array(numbers);
```

TAM Code

Bijbehorende TAM code:

```
PUSH 12
LOADL 3
LOADL 11
LOADL 10
LOADL 2
LOADL 5
LOADL 8
LOADL 4
LOADL 17
```

```
LOADL 9
LOADL 2
STORE(1) 0[LB]
STORE(1) 1[LB]
STORE(1) 2[LB]
STORE(1) 3[LB]
STORE(1) 4[LB]
STORE(1) 5[LB]
STORE(1) 6[LB]
STORE(1) 7[LB]
STORE(1) 8[LB]
STORE(1) 9[LB]
LOADL 32
LOADL 46
LOADL 102
LOADL 108
LOADL 101
LOADL 115
LOADL 114
LOADL 117
LOADL 111
LOADL 121
LOADL 32
LOADL 115
LOADL 114
LOADL 101
LOADL 98
LOADL 109
LOADL 117
LOADL 110
LOADL 32
LOADL 48
LOADL 49
LOADL 32
LOADL 114
LOADL 101
LOADL 116
LOADL 110
LOADL 101
LOADL 32
LOADL 111
LOADL 116
LOADL 32
```

```
LOADL 102
LOADL 32
LOADL 101
LOADL 112
LOADL 121
LOADL 84
LOADL 32
LOADL 63
LOADL 115
LOADL 114
LOADL 101
LOADL 98
LOADL 109
LOADL 117
LOADL 110
LOADL 32
LOADL 116
LOADL 108
LOADL 117
LOADL 97
LOADL 102
LOADL 101
LOADL 100
LOADL 32
LOADL 101
LOADL 115
LOADL 85
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
```

```
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
LOADA 10[LB]
```

```
CALL get
L10:
LOAD(1) 10[LB]
LOADL 102
LOADL 1
CALL eq
JUMPIF(0) L11[CB]
LOADL 0
STORE(1) 11[LB]
L12:
LOAD(1) 11[LB]
LOADL 10
CALL lt
JUMPIF(0) L13[CB]
LOAD(1) 11[LB]
LOADL 10
CALL(LB) valid0[CB]
LOADA 0[LB]
LOAD(1) 11[LB]
CALL add
CALL getint
LOAD(1) 11[LB]
LOADL 1
CALL add
STORE(1) 11[LB]
JUMP L12[CB]
L13:
JUMP L9[CB]
L11:
L9:
LOAD(1) 9[LB]
LOAD(1) 8[LB]
LOAD(1) 7[LB]
LOAD(1) 6[LB]
LOAD(1) 5[LB]
LOAD(1) 4[LB]
LOAD(1) 3[LB]
LOAD(1) 2[LB]
LOAD(1) 1[LB]
LOAD(1) 0[LB]
CALL(LB) 11211410511011609509711411409712 1[CB]
POP(0) 1
LOADL 116
```

```
LOADL 114
LOADL 111
LOADL 115
LOADL 32
LOADL 103
LOADL 110
LOADL 105
LOADL 116
LOADL 114
LOADL 97
LOADL 116
LOADL 83
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
LOAD(1) 9[LB]
LOAD(1) 8[LB]
LOAD(1) 7[LB]
LOAD(1) 6[LB]
LOAD(1) 5[LB]
LOAD(1) 4[LB]
LOAD(1) 3[LB]
LOAD(1) 2[LB]
LOAD(1) 1[LB]
LOAD(1) 0[LB]
CALL(LB) 115111114116[CB]
STORE(1) 0[LB]
STORE(1) 1[LB]
STORE(1) 2[LB]
STORE(1) 3[LB]
STORE(1) 4[LB]
STORE(1) 5[LB]
STORE(1) 6[LB]
```

```
STORE(1) 7[LB]
STORE(1) 8[LB]
STORE(1) 9[LB]
LOAD(1) 9[LB]
LOAD(1) 8[LB]
LOAD(1) 7[LB]
LOAD(1) 6[LB]
LOAD(1) 5[LB]
LOAD(1) 4[LB]
LOAD(1) 3[LB]
LOAD(1) 2[LB]
LOAD(1) 1[LB]
LOAD(1) 0[LB]
CALL(LB) 112114105110116095097114114097121[CB]
POP(0) 1
POP(0) 12
HALT
115111114116:
PUSH 27
LOAD(10) -10[LB]
STORE(1) 3[LB]
STORE(1) 4[LB]
STORE(1) 5[LB]
STORE(1) 6[LB]
STORE(1) 7[LB]
STORE(1) 8[LB]
STORE(1) 9[LB]
STORE(1) 10[LB]
STORE(1) 11[LB]
STORE(1) 12[LB]
LOADL 0
STORE(1) 15[LB]
LOADL 0
STORE(1) 16[LB]
LOADL 10
STORE(1) 17[LB]
L0:
LOAD(1) 15[LB]
LOADL 10
CALL lt
JUMPIF(0) L1[CB]
LOAD(1) 15[LB]
LOADL 10
```

```
CALL(LB) valid0[CB]
LOADA 3[LB]
LOAD(1) 15[LB]
CALL add
LOADI(1)
STORE(1) 18[LB]
LOAD(1) 15[LB]
STORE(1) 19[LB]
L2:
LOAD(1) 19[LB]
LOADL 0
CALL gt
LOAD(1) 19[LB]
LOADL 1
CALL sub
LOADL 10
CALL(LB) valid0[CB]
LOADA 3[LB]
LOAD(1) 19[LB]
LOADL 1
CALL sub
CALL add
LOADI(1)
LOAD(1) 18[LB]
CALL gt
CALL and
JUMPIF(0) L3[CB]
LOAD(1) 19[LB]
LOADL 1
CALL sub
LOADL 10
CALL(LB) valid0[CB]
LOADA 3[LB]
LOAD(1) 19[LB]
LOADL 1
CALL sub
CALL add
LOADI(1)
LOAD(1) 19[LB]
LOADL 10
CALL(LB) valid0[CB]
LOADA 3[LB]
LOAD(1) 19[LB]
```

```
CALL add
STOREI(1)
LOAD(1) 19[LB]
LOADL 1
CALL sub
STORE(1) 19[LB]
JUMP L2[CB]
L3:
LOAD(1) 18[LB]
LOAD(1) 19[LB]
LOADL 10
CALL(LB) valid0[CB]
LOADA 3[LB]
LOAD(1) 19[LB]
CALL add
STOREI(1)
LOAD(1) 15[LB]
LOADL 1
CALL add
STORE(1) 15[LB]
JUMP L0[CB]
L1:
LOAD(1) 12[LB]
LOAD(1) 11[LB]
LOAD(1) 10[LB]
LOAD(1) 9[LB]
LOAD(1) 8[LB]
LOAD(1) 7[LB]
LOAD(1) 6[LB]
LOAD(1) 5[LB]
LOAD(1) 4[LB]
LOAD(1) 3[LB]
RETURN(10) 1
112114105110116095097114114097121:
PUSH 23
LOAD(10) -10[LB]
STORE(1) 3[LB]
STORE(1) 4[LB]
STORE(1) 5[LB]
STORE(1) 6[LB]
STORE(1) 7[LB]
STORE(1) 8[LB]
STORE(1) 9[LB]
```

```
STORE(1) 10[LB]
STORE(1) 11[LB]
STORE(1) 12[LB]
LOADL 0
STORE(1) 13[LB]
LOADL 10
STORE(1) 14[LB]
LOAD(1) 14[LB]
CALL put
LOADL 91
CALL put
LOADL 0
L4:
LOAD(1) -1[ST]
LOADA 3[LB]
CALL add
LOADI(1)
STORE(1) 15[LB]
LOAD(1) 15[LB]
CALL putint
LOAD(1) 13[LB]
LOADL 1
CALL add
STORE(1) 13[LB]
L7:
LOAD(1) 13[LB]
LOADL 10
CALL lt
JUMPIF(0) L8[CB]
LOADL 32
LOADL 44
CALL put
CALL put
JUMP L6[CB]
L8:
L6:
CALL succ
LOAD(1) -1[ST]
JUMPIF(10) L5[CB]
JUMP L4[CB]
L5:
POP(0) 1
LOADL 93
```

```
CALL put
LOAD(1) 14[LB]
CALL put
LOADL 1
RETURN(1) 1
valid0: LOAD(1) -2[LB]
LOAD(1) -1[LB]
CALL ge
JUMPIF(0) valid1[CB]
LOADL 83
LOADL 68
LOADL 78
LOADL 85
LOADL 79
LOADL 66
LOADL 32
LOADL 70
LOADL 79
LOADL 32
LOADL 84
LOADL 85
LOADL 79
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
CALL put
HALT
valid1: RETURN(0) 2
```

Output

En de bijbehorende output van een uitvoer van het programma. Input van de gebruiker is schuingedrukt.

Use default numbers? Type f to enter 10 numbers yourself. *f*
1
4
9
100
3
17
1337
4
2
11

[1, 4, 9, 100, 3, 17, 1337, 4, 2, 11]
Starting sort
[1, 2, 3, 4, 4, 9, 11, 17, 100, 1337]

Use default numbers? Type f to enter 10 numbers yourself. *t*

[2, 9, 17, 4, 8, 5, 2, 10, 11, 3]
Starting sort
[2, 2, 3, 4, 5, 8, 9, 10, 11, 17]

Use default numbers? Type f to enter 10 numbers yourself. *f*
1
2
3
4
5
6
7
8

9
10

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Starting sort
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
