

KIDEB
Onze eigen programmeertaal

H.M. Bastiaan
s1204254
Doctor van Damstraat 198, Enschede

V.J. Smit
s1206257
Kremersmaten 168, Enschede

9 juli 2014

Inhoudsopgave

1	Inleiding	2
2	Taalbeschrijving	3
3	Problemen en oplossingen	4
3.1	Scoping	4
3.2	Opbouwen AST	4
3.3	Type inferentie	4
3.4	Import statement	4
3.5	Arrays als argument	5
4	Syntax, context en semantiek	6
4.1	Syntax	6
4.1.1	Terminale symbolen	6
4.1.2	Non-terminale symbolen	7
4.1.3	Productieregels	8
4.2	Context	11
4.2.1	Scope regels	11
4.2.2	Type regels	11
4.3	Semantiek	11
5	Vertaalregels	14
6	Java programmatuur	17
6.1	Symbol table	17
6.2	Type checking	17
6.3	AST klassen	17
6.4	Foutafhandeling	18
7	Testplan en resultaten	19
7.1	Syntax- en contexttests	19
7.2	Semantiektests	19
8	Conclusies	20
9	Appendix	21
9.1	Lexer specificatie	21
9.2	Parser specificatie	25
9.3	Treeparser specificaties	29
9.4	Testverslag	55

Hoofdstuk 1

Inleiding

Dit verslag beschrijft de zelf ontwikkelde programmeertaal KIDEB. Deze taal is gedefinieerd met de hulp van het ANTLR-framework versie 3.

In het verslag wordt allereerst de taal kort ingeleid. Hierop volgt een korte probleem-analyse. Daarna wordt de taal gedefinieerd in een grammatica en worden de contextregels en de semantiek uitgelegd. Vervolgens volgt uitleg over de vertaling van KIDEB-broncode naar TAM-code. Uiteindelijk worden de extra geschreven java-klassen en het testplan uitgelegd. In de appendix zijn specificaties van de lexer, de parser en de treeparsers toegevoegd, evenals een uitgebreid testprogramma met output.

De grammatica is opgebouwd door ons beiden. De checker is voor het grootste deel het werk van Martijn en de code generator volledig. Het verslag is voor het grootste gedeelte opgebouwd door Vincent.

Hoofdstuk 2

Taalbeschrijving

KIDEB is een kleine, imperatieve programmeertaal met beperkte mogelijkheden, ontwikkeld als project voor het vak vertalerbouw als onderdeel van de Technische Informatica bachelor van de Universiteit Twente. Ondanks de beperkte mogelijkheden bevat de taal toch enkele leuke onderdelen.

Uiteraard is de basis van een programmeertaal ook aanwezig in KIDEB. Variabelen kunnen worden gedeclareerd met primitieve typen: integer, boolean en character. Door toegevoegde type inferentie hoeft het primitieve type zelfs niet expliciet te worden vermeld.

Op deze variabelen zijn een aantal bewerkingen mogelijk. Zo zijn de standaard boolean operaties beschikbaar, om verschil of gelijkheid te bepalen. Daarnaast bevat de taal de basis rekenkundige operaties en zelfs bestaat de mogelijkheid tot machtsverheffen. Belangrijke statements zijn ook geïmplementeerd. If-else is onderdeel van de taal, evenals een while-statement.

De eerste belangrijke uitbreiding is de mogelijkheid gebruik te maken van subroutines. Naast het hoofdprogramma zijn namelijk ook functies te definiëren en uit te voeren. Functies geven altijd een waarde terug. De tweede belangrijke uitbreiding was al even genoemd. Dit is namelijk het gebruik van arrays. Door het gebruik van arrays wordt de taal een stuk complexer, maar ook een stuk sterker. Daarnaast kan reeds eerder geschreven broncode geïmporteerd worden. Grote code reduplicatie is dus overbodig. Als laatste bestaat ook de mogelijkheid met geheugen allocatie bezig te zijn. Geheugen kan gealloceerd en vrijgemaakt worden en er kan naar geheugenlocaties verwezen worden met behulp van pointers.

De volledige grammatica van de taal is te vinden in het hoofdstuk 4.

Hoofdstuk 3

Problemen en oplossingen

In dit hoofdstuk worden enkele belangrijke problemen besproken die bij de ontwikkeling van de taal naar voren kwamen. Allereerst komt scoping aan bod. Daarna wordt kort de AST besproken, gevolgd door type inferentie.

3.1 Scoping

Een belangrijk probleem bij het programmeren is het definiëren en gebruik van variabelen in verschillende scopes. Variabelen die bijvoorbeeld binnen een lus worden gedefinieerd, mogen daarbuiten niet gebruikt worden. Ook moet de variabele worden gebruikt die gedeclareerd is onder of in die scope.

Voor het definiëren van scopes is de volgende oplossing gekozen. Een scope binnen KIDEB bestaat tussen twee accolades. De scope wordt geopend door een ‘{’ en gesloten door een ‘}’. Verdere uitleg over scoping is te vinden in subsectie 4.2.1.

Om bij te houden waar een variabele gedeclareerd is en gebruikt wordt, wordt een symbol table bijgehouden.

3.2 Opbouwen AST

Voor het opbouwen van onze AST voldeed de standaard node niet. Hiertoe is een eigen node-hiërarchie gemaakt. De specificatie van deze nodes is te vinden in hoofdstuk 6.

Een ander probleem met de AST is het moeten gebruiken van `CommonNode` voor de belangrijke node duplicatie. Om dit te voorkomen heeft de klasse `AbstractNode`, als subklasse van `CommonNode`, een functie met generiek type. Hier kunnen subclasses van `CommonNode` toch de methode *getDuplicate()* gebruiken.

3.3 Type inferentie

3.4 Import statement

Een belangrijke functie in onze taal is het importeren van code. Dit is een lastig probleem, met een vrij simpele oplossing. Op de plek van het import-statement wordt in de AST van het hoofdprogramma de AST van de geïmporteerde code geplaatst. Hierdoor is de code ook in het hoofdprogramma te gebruiken.

3.5 Arrays als argument

Het is niet mogelijk om in een functiedeclaratie een array mee te geven zonder een expressie binnen de blokhaken. Het is echter absoluut niet wenselijk te eisen dat arrays als argument voor een functie een vaste lengte hebben. Om dit probleem te verhelpen moeten functies die een array nodig heeft een pointer naar deze array mee krijgen, in plaats van de array zelf.

Hoofdstuk 4

Syntax, context en semantiek

Dit hoofdstuk bespreekt de specificatie van de taal aan de hand van de syntax, de context regels en de semantiek.

4.1 Syntax

Deze sectie beschrijft de symbolen en productieregels van KIDEB. Samen vormen deze de totale grammatica van de taal.

4.1.1 Terminale symbolen

De terminale symbolen:

:	;	()	[
]	{	}	,	\
!=	+	-	/	<
^	=	<	>	>=
<=	==		&&	*
&	%	print	import	call
swap	if	else	then	do
while	from	break	continue	return
returns	func	true	false	bool
char	var	of	int	

4.1.2 Non-terminale symbolen

De non-terminale symbolen:

program (startsymbol)

command

declaration IDENTIFIER

- var_declaration
- scope_declaration
- func_declaration
- assign_statement

assignment

- var_assignment

argument

- arguments

statement

- while_statement
- if_statement
- if_part
- else_part
- for_statement
- return_statement
- assign_statement
- print_statement
- import_statement

expression

- expressionAO
- expressionLO
- expressionPM
- expressionMD
- expressionPW
- expression_list
- call_expression
- raw_expression
- get_expression
- operand
- bool
- array_literal
- array_value_list

type

- primitive_type
- compositie_type

identifier

number

4.1.3 Productieregels

program :=
 command+;

command :=
 assign_statement SEMICOLON |
 declaration |
 statement |
 expression |
 SEMICOLON;

commands :=
 command commands?;

declaration :=
 var_declaration |
 scope_declaration;

var_declaration :=
 type IDENTIFIER (var_assignment) SEMICOLON;

scope_declaration :=
 func_declaration;

func_declaration :=
 FUNC IDENTIFIER LPAREN arguments? RPAREN RETURNS type LCURLY
 commands? RCURLY;

assignment :=
 ASSIGN expression;

var_assignment :=
 ASSIGN expression;

argument :=
 type IDENTIFIER;

arguments :=
 argument (COMMA arguments)?;

statement :=
 if_statement |
 while_statement |
 return_statement |
 import_statement |
 BREAK SEMICOLON |
 CONTINUE SEMICOLON;

if_statement :=
 if_part else_part?;

if_part :=
 IF LPAREN expression RPAREN LCURLY command* RCURLY;

else_part :=
 ELSE LCURLY command* RCURLY;

while_statement :=
 WHILE LPAREN expression RPAREN LCURLY commands? RCURLY;

for_statement :=
 FOR LPAREN expression RPAREN LCURLY commands? RCURLY;

return_statement :=
 RETURN expression SEMICOLON;

print_statement :=
 PRINT LPAREN expression RPAREN;

import_statement :=
 IMPORT STRING_VALUE;

expression :=
 raw_expression |
 expressionAO |
 array_literal;

expressionAO :=
 expressionLO (AND expressionLO |OR expressionLO)*;

expressionLO :=
 expressionPM ((LT |GT |LTE |GTE |EQ |NEQ) expressionPM)*;

expressionPM :=
 expressionMD ((PLUS |MINUS) expressionMD)*;

expressionMD :=
 expressionPW ((MULTIPLE |DIVIDE) expressionPW);

expressionPW :=
 operand (POWER operand)*;

expression_list :=
 expression (COMMA expression_list)?;

call_expression :=
 IDENTIFIER LPAREN expression_list? RPAREN;

get_expression :=
 IDENTIFIER LBLOCK expression RBLOCK;

operand :=
 get_expression |
 call_expression |
 ASTERIX operand |
 AMPERSAND IDENTIFIER |
 ASTERIX operand |
 LPAREN expression RPAREN |
 IDENTIFIER |

```

    NUMBER |
    STRING_VALUE |
    bool;

bool :=
    TRUE |
    FALSE;

array_literal :=
    LBLOCK array_value_list RBLOCK;

array_value_list :=
    expression (COMMA array_value_list)?;

type
    primitive_type
    composite_type

primitive_type :=
    INTEGER |
    BOOLEAN |
    CHARACTER |
    AUTO |
    VAR;

composite_type :=
    primitive_type LBLOCK expression RBLOCK

IDENTIFIER :=
    (LETTER | UNDERSCORE) (LETTER | DIGIT | UNDERSCORE);

NUMBER :=
    DIGIT+;

STRING_VALUE :=
    '(\\\'? |~(\\|\' ) *';

COMMENT :=
    // .* \n;

WS :=
    _ | \t | \f | \r | \n

DIGIT :=
    0..9;

LETTER :=
    LOWER | UPPER;

LOWER :=
    a..z;

UPPER :=
    A..Z;

UNDERSCORE :=
    -;

```

4.2 Context

De context van de taal wordt opgedeeld in twee delen, namelijk scope regels en type regels. De eerste bespreekt declaratie en het gebruik van variabelen. De tweede bespreekt de typering van de taal.

4.2.1 Scope regels

Om de scoperegels uit te leggen, gebruiken we de volgende voorbeeld code.

```
1 int x;  
2 x = 5;  
3  
4 func som(int x) returns int {  
5     int y = 7;  
6     return x + y;  
7 }  
8  
9 print(som(x));
```

Op regel 1 wordt variabele x gedeclareerd, dit is de *binding occurrence* voor x . De eerste *applied occurrence* komt meteen op regel 2, waar x de waarde 5 krijgt.

De functie *som* wordt op regel 4 gedefinieerd en telt de waarde van variabele y hierbij op. De variabele y wordt gedefinieerd binnen de functie en is dus ook alleen binnen de functie te gebruiken.

4.2.2 Type regels

Voor de rekenkundige operatoren gelden de volgende type regels.

prioriteit	operatoren	operand types	resultaat type
1	\wedge	int	int
2	$*, /$	int	int
3	$+, -$	int	int
4	$<, <=, >=, >$	int	bool
	$==, !=$	int, char, bool	bool
5	$\&\&, $	bool	bool

Voor de statements gelden de volgende regels.

```
if Expression then Command else Command  
Expression must be of type boolean.  
while Expression do Command  
Expression must be of type boolean.  
Identifier = Expression  
Identifier and Expression must be of the same type.  
import String.waarde  
String.waarde moet een bestaande bestandsnaam zijn.
```

4.3 Semantiek

Deze sectie bespreekt de semantiek, ofwel de betekenis van de geschreven code.

Een *statement* S wordt uitgevoerd om de variabelen te updaten. Dit is inclusief input en output.

- Assign-statement($I = E$): Expressie E wordt gevalueerd en levert de waarde v op. Deze waarde wordt gekoppeld aan I .
- Import-statement(import SW): Van stringwaarde SW wordt uitgezocht of het bestaat als .kib broncode bestand. Zo ja, wordt de AST van het programma uitgebreid met de AST van dit bestand en is de code uit dit bestand beschikbaar voor gebruik in de eigen broncode.
- If-statement(if E then C_1 else C_2): Expressie E wordt geëvalueerd en levert een booleanwaarde op. Als de boolean waarde **true** is, worden commando's C_1 uitgevoerd, anders commando's C_2 .
- While-statement(while E do C): Expressie E wordt gevalueerd en levert een booleanwaarde op. Als de booleanwaarde **true** is, wordt commando C uitgevoerd. Daarna wordt E opnieuw gevalueerd. Is de booleanwaarde **false**, dan eindigt de loop.
- Print-statement(print(E)): De expressie E wordt gevalueerd en levert waarde v op. Deze waarde v wordt op de standaard output getoond.

Een *expressie* E levert een waarde op na evaluatie.

- expressionAO(E_1 operator E_2): Evalueert expressie E_1 en E_2 , welke beiden een boolean opleveren. De waarde die de expressie AO oplevert is ook een boolean, met als operator de binaire AND(&&) of OR(||).
- expressionLO(E_1 operator E_2): Evalueert expressie E_1 en E_2 , welke beiden een integer opleveren. De waarde die de expressie LO oplevert is een boolean, met als operator een waarde vergelijker.
- expressionPM(E_1 operator E_2): Evalueert expressie E_1 en E_2 , welke beiden een integer opleveren. De waarde die de expressie AO oplevert is de opgetelde of afgetrokken waarde van beide expressies.
- expressionMD(E_1 operator E_2): Evalueert expressie E_1 en E_2 , welke beiden een integer opleveren. De waarde die de expressie AO oplevert is de vermenigvuldigde of gedeelde waarde van beide expressies.
- expressionPW(E_1 operator E_2): Evalueert expressie E_1 en E_2 , welke beiden een integer opleveren. De waarde die de expressie AO oplevert is E_1 tot de macht E_2 .
- raw-expressie(_tam_ (T, SW)): De stringwaarde SW is TAM-code. Met deze expressie is dus direct TAM-code uit te voeren in de taal. Het type T is het return-type van de expressie.
- call-expressie(I(E_1, \dots, E_x)): Evalueert expressie E_1 tot en met E_x en leveren waarden v_1 tot en met v_x op. Functie I wordt vervolgens aangeroepen met argumenten v_1 tot en met v_x .
- get-expressie(I[E]): Expressie E wordt geëvalueerd en levert een integer met waarde v op. Uit de array met identifier I wordt vervolgens de waarde op positie v terug gegeven.

Een *delclaratie* D wordt uitgevoerd om bindingen te maken.

- Variabele declaratie(T I): Identifier I wordt gebonden aan een waarde, die op dit moment nog onbekend is. De waarde moet gelijk zijn aan type T . De variabele wordt buiten de scope waarin deze wordt gebruikt, gedealloceerd.
- Functie declaratie(func $I(ARGS)$ returns $T\{C\}$): Functie met identifier I wordt aangemaakt. De argumenten $ARGS$ stellen de waarden voor die de functie in de aanroep ervan mee moet krijgen. Type T is het type van de return-waarde van functie I . Commando C vormt de body van de functie en dit zijn de commando's die uitgevoerd worden na aanroep van deze functie.

Hoofdstuk 5

Vertaalregels

Om de vertaalregels van de broncode naar TAM-code duidelijk te maken, worden code templates gebruikt. KIDEB kent de volgende acties.

Klasse	Code functie	Effect van gegenereerde code
Program	<i>run P</i>	Draai programma P en daarna stoppen. Beginnen en eindigen met een lege stack.
Commando	<i>do C</i>	Is de volledige lijst met instructies, bestaande uit statements, expressies en declaraties.
Statement	<i>execute S</i>	Voer het statement S uit met mogelijk aanpassen van variabelen, maar zonder effect op de stack.
Expressie	<i>evaluate E</i>	Evalueer de expressie E en push het resultaat naar de stack. Geen verder effect op de stack.
Identifier	<i>fetch I</i>	Push de waarde van identifier I naar de stack.
Identifier	<i>assign I</i>	Pop een waarde van de stack en sla deze op in variabele I.
Declaratie	<i>declare D</i>	Verwerk declaratie D, breidt de stack uit om ruimte te maken voor variabelen die hierin gedeclareerd worden.

Een programma in KIDEB is een serie commands. Elk los command kan een statement, een expressie of een declaratie zijn. Een programma kan er dus als volgt uitzien.

```
run [[C ]] =  
  do C  
  HALT
```

Dit leidt tot de volgende commando's.

```
do [[S ]] =  
  execute S
```

```
do [[E ]] =  
  evaluate E
```

do **[[D]]** =
 declare D

De statements gaan als volgt.

execute **[[I = E;]]**
 evaluate E
 assign I

execute **[[if E then C₁ else C₂]]** =
 evaluate E
 JUMPIF(0) *g*
 execute C₁
 JUMP *h*
 g: gexecute C₂
 h:

execute **[[while E do C]]** =
 g: evaluate E
 JUMPIF(0) *h*
 execute C
 JUMP *g*
 h:

execute **[[return E]]** =
 evaluate E
 RETURNS(1)

execute **[[import SW]]** =
 Is wel een statement, maar levert geen code op. Importeert nl een bestand met naam SW, en voegt de AST van die code toe aan de AST van het hoofdbestand.

execute **[[print E]]** =
 evaluate E
 CALL put
 LOADL 10 newline
 CALL putint

De expressie worden als volgt verwerkt.

evaluate **[[E₁ O E₂]]** =
 evaluate E₁
 evaluate E₂
 CALL *p* *p* is het adres van de primitieve routine die hoort bij O

evaluate **[[I]]** =
 fetch I

evaluate **[[I(E₁,...,E_x)]]** =
 evaluate E₁
 ...
 evaluate E_x
 CALL(*r*) *e* //*e* is de geheugenlocatie van de commando's van I. (zie functiedeclaratie)

Laden en opslaan van waarden in variabelen

```
fetch [[I ]] =  
    LOAD (1) d[r]
```

```
assign [[I ]] =  
    STORE (1) d[r]
```

Declaraties worden hiermee afgehandeld. Variabelen worden pas gedeclareerd wanneer hun bovenliggende functie gedeclareerd wordt. Als de generator een declaratie van een variabele tegenkomt, slaat hij deze dus over tot de functie declareerd wordt. Dan pas wordt er ruimte gemaakt voor de variabele.

```
delcare [[func I(ARGS) returns T C ]] =  
    JUMP g  
e: execute C  
    RETURN(1) d    // d met groote van het aantal argumenten ARGS  
g:
```

```
declare [[T I ]] =  
    PUSH (1)
```

Hoofdstuk 6

Java programmatuur

Voor het correct laten werken van de KIDEB-compiler zijn een aantal extra Java-klassen gedefinieerd. Deze betreffen het opbouwen van de *symbol table*, code voor het checken van types en de extra nodes voor de AST. Ook valt hieronder een stukje foutafhandeling.

6.1 Symbol table

De zogenaamde *symbol table* zoekt uit welke variabele waar gedefinieerd is, waar deze gebruikt wordt en koppelt dit aan elkaar. De reeds gemaakte *symbol table* uit het practicum van vertalerbouw is hiervoor gebruikt.

6.2 Type checking

Voor het verwerken van types is het de klasse *Type* aangemaakt. Deze klasse bevat een enumerator van alle types die onze taal kent. Deze klasse wordt gebruikt voor het zetten van de types van de identifiers.

6.3 AST klassen

De AST is uitgebreid met extra nodes, om extra benodigde informatie bij te houden.

De abstracte klasse *AbstractNode* is een subklasse van *CommonTree*, de normale AST-klasse. Deze vormt de superklasse voor de klasse *CommonNode*. Deze laatste is de superklasse voor alle zelf-gedefinieerde AST-nodes.

De klasse *ControlNode* representeert nodes die de uitvoervolgorde van het programma veranderen. Hieronder vallen onder andere het *return*-, *continue*- en *break*-statement. Deze klasse houdt de scope bij waar het statement bij hoort. Deze klasse is een subklasse van *CommonNode*.

Een andere subklasse van *CommonNode* is de klasse *TypedNode*. Deze klasse vormt de superklasse van alle nodes in de AST die van een bepaald type zijn. Daarom heeft deze klasse als eigenschap een type. Tevens is het geheugenadres van deze (VAN DEZE WAT??) een eigenschap.

Onder de *TypedNode* komt de *IdentifierNode*. Deze node wordt gebruikt om de scope van een identifier te bepalen, evenals zijn type. Als eigenschap heeft deze klasse een zogenaamde *realNode*. Deze *realNode* is de node waar de declaratie van deze identifier plaatsvindt en er wordt bij gebruik van de identifier naar verwezen.

Het laagst in de hiërarchie zit de `FunctionNode`. Deze node wordt vanzelfsprekend gebruikt voor functies en houdt een lijst van identifiers en bijbehorende types bij. Deze lijst is dus de lijst met argument. Tevens heeft deze node een naam en `returnType` als eigenschap.

6.4 Foutafhandeling

Om compilatie af te breken bij een typefout, is de `InvalidTypeException`-klasse geïmplementeerd. Deze exceptie wordt gegooid bij een declaratie, als het gedeclareerde type niet bestaat. Tevens wordt deze exceptie gegooid als er verkeerde types in expressies gebruikt worden.

Hoofdstuk 7

Testplan en resultaten

Om de juistheid van de compiler te garanderen, is testen essentieel. Hiertoe zijn dan ook twee typen tests geschreven. De eerste is een in *python* geschreven programma, welke de syntax en context test. Het tweede type bestaat uit grotere voorbeeldprogramma's, welke syntax, context en semantiek testen.

De testprogramma's geschreven in *python* zijn te vinden in het bijgeleverde zip-bestand in de map *tests*. De testprogramma's uit onze eigen taal in de map *examples*.

7.1 Syntax- en contexttests

De python programma's testen zeer snel en efficiënt de volledige syntax en context. Hieronder vallen zowel correcte als foute code. Als de code bewust fout gaat, wordt dit afgevangen op de correcte foutmelding.

Als al deze tests slagen is er geen noemenswaardige output. Mocht er een test wel falen, dan wordt dit getoond op de standaard output.

7.2 Semantiektests

De voorbeeldprogramma's in de map *tests* testen de semantiek van de taal. Deze programma's leveren TAM-code en ... op als output. Het grootste testbestand is bijgevoegd in de appendix in sectie 9.4.

array.kib Het declareren van, toewijzen van waarden aan, en uitlezen van een array.

power.kib Het declareren en gebruik van een functie.

fibonacci_recursive.kib Het gebruik van recursie.

heap.kib Het alloceren en vrijgeven van geheugen wordt hier getest.

pointers.kib Test de declaratie en het gebruik van pointers.

raw.kib Het direct uitvoeren van TAM-code in de KIDEB broncode.

all.kib Test alle functionaliteit van de taal. Programma en output is te vinden in sectie 9.4.

Hoofdstuk 8

Conclusies

Ondanks de beperkte mogelijkheden is er een leuke expressietaal gebouwd. Er zijn de nodige functies geïmplementeerd bovenop de basisfunctionaliteit, welke de taal een stuk krachtiger maken. Hier zaten een aantal uitdagende problemen in, die allen opgelost zijn.

De taal is gedefinieerd als een LL(1) grammatica, op een enkele, lokale uitzondering na. Door stellen van context-eisen en de uitleg van de semantiek is de grammatica bruikbaar als programmeertaal. Op basis van de vertaalregels is broncode geschreven in KIDEB ook compileerbaar naar TAM-instructies, waardoor deze uitgevoerd kan worden op een TAM-machine. De controle van de syntax en context en de codegeneratie zijn gedefinieerd in het ANTLR-framework.

KIDEB is ook vrij uitgebreid getest door in *python* geschreven unittests. Het overgrote deel van de bugs die in de taal zaten is hiermee gevonden en verholpen.

Kortom, KIDEB is een kleine maar aardige taal, voortgekomen uit een leuk project van de Technische Informatica bachelor.

Hoofdstuk 9

Appendix

9.1 Lexer specificatie

Voor de lexer zijn de verschillende tokens van belang. Deze tokens staan hieronder allen gedefinieerd.

Tekens	
Token	teken
COLON	:
SEMICOLON	;
LPAREN	(
RPAREN)
LBLOCK	[
RBLOCK]
LCURLY	{
RCURLY	}
COMMA	,
DOUBLE_QUOTE	"
SINGLE_QUOTE	\
BODY	body
EXPR	assignment_expression
GET	get_expression

Operators

Token	token
PLUS	+
MINUS	-
DIVIDES	/
MULTIPL	*
POWER	^
LT	<
GT	>
GTE	>=
LTE	<=
EQ	=
NEQ	!
ASSIGN	==
OR	
AND	&&

Pointers

Token	token
AMPERSAND	&
ASTERIX	%

Keywords van KIDEB

Token	keyword
PROGRAM	program
SWAP	swap
IF	if
THEN	then
ELSE	else
DO	do
WHILE	while
FROM	from
IMPORT	import
BREAK	break
CONTINUE	continue
RETURN	return
FOR	for
IN	in
RETURNS	returns
FUNC	func
ARRAY	array
ARGS	args
CALL	call
VAR	var
OF	of
PRINT	print
TAM	__tam__

Standaard types

Token	keyword
INTEGER	int
CHARACTER	char
BOOLEAN	bool
AUTO	auto

9.2 Parser specificatie

25

```
1 // Parser rules
2 program: command+ -> ^(PROGRAM command+);
3
4 command:
5     (IDENTIFIER ASSIGN) => assign_statement SEMICOLON! |
6     (IDENTIFIER MULT* ASSIGN) => assign_statement SEMICOLON! |
7     (IDENTIFIER LBLOCK expression RBLOCK ASSIGN) => assign_statement SEMICOLON! |
8     statement |
9     declaration |
10    expression SEMICOLON! |
11    SEMICOLON!;
12
13 commands: command commands?;
14
15 // Declarations
16 declaration:
17     var_declaration |
18     scope_declaration;
19
20 var_declaration: type IDENTIFIER (a=var_assignment)? SEMICOLON
21                 -> {a == null}? ^(VAR type IDENTIFIER<IdentifierNode>)
22                 -> ^(VAR type IDENTIFIER<IdentifierNode>) ^(ASSIGN IDENTIFIER<IdentifierNode> ^(EXPR $a));
23
24 var_assignment: ASSIGN! expression;
25
26 scope_declaration:
27     func_declaration;
28
29 func_declaration: FUNC IDENTIFIER LPAREN arguments? RPAREN (RETURNS t=type)? LCURLY commands? RCURLY
30                 -> {t == null}? ^(FUNC IDENTIFIER<FunctionNode> IDENTIFIER<IdentifierNode> ["auto"] ^(ARGS arguments
31                 ?) ^(BODY commands?))
32                 -> ^(FUNC IDENTIFIER<FunctionNode> type ^(ARGS arguments?) ^(BODY commands?));
33
34 // Parses arguments of function declaration
35 argument: type IDENTIFIER;
```

```

36 arguments: argument (COMMA! arguments)?;
37
38 // Statements
39 statement:
40     if_statement |
41     while_statement |
42     return_statement |
43     for_statement |
44     print_statement |
45     import_statement |
46     BREAK SEMICOLON! |
47     CONTINUE SEMICOLON!;
48
49
50 if_part: IF LPAREN expression RPAREN LCURLY commands? RCURLY
51         -> expression ^(THEN commands?);
52
53 else_part: ELSE LCURLY commands? RCURLY
54           -> ^(ELSE commands?);
55
56 if_statement: if_part ep=else_part?
57             -> ^(IF if_part else_part?);
58
59 while_statement: WHILE LPAREN expression RPAREN LCURLY command* RCURLY
60                 -> ^(WHILE expression command*);
61 for_statement: FOR IDENTIFIER IN expression LCURLY commands? RCURLY
62               -> ^(FOR IDENTIFIER<IdentifierNode> expression commands?);
63
64 return_statement: RETURN expression SEMICOLON -> ^(RETURN expression);
65
66 import_statement
67 @init { CommonNode includetree = null; }:
68     IMPORT s=STRING.VALUE {
69         try {
70             String filename = $s.text.substring(1, $s.text.length() - 1);
71             GrammarLexer lexer = new GrammarLexer(new ANTLRFileStream(filename + ".kib"));
72             GrammarParser parser = new GrammarParser(new CommonTokenStream(lexer));
73             parser.setTreeAdaptor(new CommonNodeAdaptor());

```

```

74     includetree = (CommonNode)(parser.program().getTree());
75 } catch (Exception fnf) {
76     // TODO: Error handling?
77 };
78 }
79 }
80 -> ^({includetree})
81 ;
82
83 assign:
84     MULT assign -> ^(DEREFERENCE assign) |
85     ASSIGN expression -> ^(EXPR expression) |
86     LBLOCK expression RBLOCK a=assign -> ^ (GET assign expression);
87
88 assign_statement:
89     IDENTIFIER assign -> ^ (ASSIGN IDENTIFIER assign);
90
91 print_statement: PRINT LPAREN expression RPAREN -> ^ (PRINT expression);
92
93 // Expressions, order of operands:
94 // ()
95 // ^
96 // *, /
97 // +, -
98 // <=, >=, <, >, ==, !=, ||, &&
99 expression:
100     expressionAO |
101     raw_expression |
102     array_literal;
103
104 expressionAO: expressionLO (AND<TypedNode>^ expressionLO | OR<TypedNode>^ expressionLO)*;
105 expressionLO: expressionPM ((LT<TypedNode>^ | GT<TypedNode>^ | LTE<TypedNode>^ | GTE<TypedNode>^ | EQ<TypedNode>^ | NEQ<
    TypedNode>^) expressionPM)*;
106 expressionPM: expressionMD ((PLUS<TypedNode>^ | MINUS<TypedNode>^) expressionMD)*;
107 expressionMD: expressionPW ((MULT<TypedNode>^ | DIVIDES<TypedNode>^ | MOD<TypedNode>^) expressionPW)*;
108 expressionPW: operand (POWER<TypedNode>^ operand)*;
109
110 expression_list: expression (COMMA! expression_list)?;

```

```

111 raw_expression: TAM^ LPAREN! type COMMA! STRING_VALUE RPAREN!;
112 call_expression: IDENTIFIER LPAREN expression_list? RPAREN
113                 -> ^(CALL IDENTIFIER expression_list?);
114
115 get_expression: IDENTIFIER LBLOCK expression RBLOCK
116                -> ^(GET IDENTIFIER expression);
117
118 operand:
119   (IDENTIFIER LBLOCK) => get_expression |
120   (IDENTIFIER LPAREN) => call_expression |
121   (MULT IDENTIFIER) => DEREERENCE^ IDENTIFIER<IdentifierNode> |
122   NOT^ operand |
123   AMPERSAND^ IDENTIFIER<IdentifierNode> |
124   MULT operand -> ^(DEREERENCE operand) |
125   LPAREN! expression RPAREN! |
126   IDENTIFIER<IdentifierNode> |
127   NUMBER<TypedNode> |
128   STRING_VALUE<TypedNode>|
129   bool;
130
131 bool: TRUE<TypedNode> | FALSE<TypedNode>;
132
133 array_literal: LBLOCK array_value_list? RBLOCK -> ^(ARRAY array_value_list?);
134 array_value_list: expression (COMMA! array_value_list)?;
135
136 // Types
137 type:
138   (primitive_type LBLOCK) => composite_type |
139   MULT type -> ^(DEREERENCE type) |
140   primitive_type;
141
142 primitive_type:
143   INTEGER<TypedNode> |
144   BOOLEAN<TypedNode> |
145   CHARACTER<TypedNode> |
146   AUTO<TypedNode> |
147   VAR<TypedNode>;
148

```

```

149 composite_type:
150     primitive_type LBLOCK expression RBLOCK
151     -> ^(ARRAY primitive_type expression+);
152
153 // Lexer rules
154 IDENTIFIER: (LETTER | UNDERSCORE) (LETTER | DIGIT | UNDERSCORE)*;
155 STRING_VALUE: '\\' ( '\\' '\\' '?' | ~( '\\' | '\\' ) ) * '\\';
156
157 NUMBER: DIGIT+;
158 COMMENT: '//' .* '\\n' { $channel=HIDDEN; };
159 WS: ( ' ' | '\\t' | '\\f' | '\\r' | '\\n' )+ { $channel=HIDDEN; };
160
161 fragment DIGIT : ('0'..'9');
162 fragment LOWER : ('a'..'z');
163 fragment UPPER : ('A'..'Z');
164 fragment UNDERSCORE : '_';
165 fragment LETTER : LOWER | UPPER;

```

29

9.3 Treeparser specificaties

De Checker-klasse.

```

1 tree grammar GrammarChecker;
2
3 options {
4     tokenVocab=Grammar;
5     ASTLabelType=CommonNode;
6     output=AST;
7 }
8
9 @rulecatch {
10     catch(RecognitionException e){
11         throw e;
12     }
13 }

```

```

14
15 @header {
16     package checker;
17     import java.util.Stack;
18     import java.util.EmptyStackException;
19     import symtab.SymbolTable;
20     import symtab.SymbolTableException;
21     import symtab.IdEntry;
22     import ast.*;
23     import reporter.Reporter;
24     import org.javatuples.Pair;
25 }
26
27 @members {
28     protected SymbolTable<IdEntry> symtab = new SymbolTable<>();
29
30     private IdentifierNode getID(CommonNode node, String id) throws InvalidTypeException{
31         if (symtab.retrieve(id) == null){
32             reporter.error(node, "Could not find symbol.");
33         }
34         return symtab.retrieve(id).getNode();
35     }
36
37     private Checkers checkers = new Checkers(this);
38
39     private Type assignType;
40
41     private int argumentCount;
42
43     private FunctionNode calling;
44
45     private Stack<Pair<FunctionNode, Stack<CommonNode>>> loops = new Stack<Pair<FunctionNode, Stack<CommonNode>>>();
46
47     private Stack<TypedNode> arrays = new Stack<>();
48
49     public Reporter reporter;
50     public void setReporter(Reporter r){ this.reporter = r; }
51     public void log(String msg){ this.reporter.log(msg); }

```

```

52 |
53 |
54 | }
55 |
56 | program
57 | @init {
58 |     symlab.openScope();
59 | }
60 | @after {
61 |     symlab.closeScope();
62 |     loops.pop();
63 | }
64 | : ^ (p=PROGRAM<FunctionNode>{
65 |     loops.push(Pair.with((FunctionNode)$p.tree, new Stack<CommonNode>()));
66 |     loops.peek().getValue0().setName("--root--");
67 |     loops.peek().getValue0().setMemAddr(Pair.with(loops.size() - 1, -1));
68 | } command+);
69 |
70 | commands: command commands?;
71 | command: declaration | expression | statement | ^(PROGRAM command+);
72 |
73 | declaration: var_declaration | scope_declaration;
74 |
75 | var_declaration:
76 |     ^ (VAR t=type id=IDENTIFIER<IdentifierNode>){
77 |         IdentifierNode var = (IdentifierNode)$id.tree;
78 |
79 |         try {
80 |             symlab.enter($id.text, new IdEntry(var));
81 |         } catch (SymbolTableException e) {
82 |             reporter.error($id.tree, String.format(
83 |                 "but variable %s was already declared %s",
84 |                 $id.text, reporter.pointer(symlab.retrieve($id.text).getNode())
85 |             ));
86 |         }
87 |
88 |         var.setExprType(((TypedNode)$t.tree).getExprType());
89 |

```



```

90     if (var.getExprType().containsVariableType()){
91         reporter.error($id.tree, "Variable cannot have variable type.");
92     }
93
94     // Register variable with function
95     FunctionNode func = loops.peek().getValue0();
96     func.getVars().add(var);
97     var.setMemAddr(Pair.with(loops.size() - 1, func.getVars().size() - 1));
98
99     log(String.format(
100         "Set relative memory address of %s to (%s, %s)",
101         $id.text, var.getMemAddr().getValue0(), var.getMemAddr().getValue1()
102     ));
103
104     var.setScope(func);
105     log(String.format("Setting scope of %s to %s().", $id.text, func.getName()));
106 };
107
108 scope_declaration: func_declaration;
109
110 func_declaration:
111     ^(FUNC id=IDENTIFIER<FunctionNode> t=type{
112         FunctionNode func = (FunctionNode)$id.tree;
113         func.setName($id.text);
114         func.setScope(loops.peek().getValue0());
115         func.setExprType(Type.Primitive.FUNCTION);
116         log(String.format("Setting %s.parent = %s", $id.text, func.getScope().getName()));
117
118         loops.push(Pair.with(func, new Stack<CommonNode>()));
119
120         try {
121             symtab.enter($id.text, new IdEntry((IdentifierNode)$id.tree));
122         } catch (SymbolTableException e) {
123             reporter.error($id.tree, String.format(
124                 "but variable %s was already declared %s",
125                 $id.text, reporter.pointer(symtab.retrieve($id.text).getNode())
126             ));
127         }

```

```

128
129     func.setReturnType(((TypedNode)$t.tree).getExprType());
130     symtab.openScope();
131     func.setMemAddr(Pair.with(loops.size() - 1, -1));
132
133     if (loops.size() > 6 + 2){ // +2 for root node, and current function declaration
134         reporter.error(func, "You can only nest functions 6 levels deep, it's morally wrong to nest deeper ;-).");
135     }
136
137     argumentCount = 0;
138 } ^(ARGS arguments?) {
139     IdentifierNode arg;
140     int inverse_count;
141     for (int i=0; i < func.getArgs().size(); i++){
142         arg = func.getArgs().get(i);
143         inverse_count = -1 * (func.getArgs().size() - i);
144         arg.setMemAddr(Pair.with(loops.size() - 1, inverse_count));
145
146         log(String.format(
147             "Setting relative address of %s to (%s, %s).",
148             arg.getText(), loops.size() - 1, inverse_count
149         ));
150     }
151 } ^(BODY commands?) {
152     symtab.closeScope();
153     loops.pop();
154 };
155
156 argument: t=type id=IDENTIFIER<IdentifierNode>{
157     IdentifierNode inode = (IdentifierNode)$id.tree;
158
159     try {
160         symtab.enter($id.text, new IdEntry(inode));
161         ((TypedNode)$id.tree).setExprType(((TypedNode)$t.tree).getExprType());
162     } catch (SymbolTableException e) {
163         reporter.error(inode, e.getMessage());
164     }
165

```

```

166     FunctionNode function = loops.peek().getValue0();
167     function.getArgs().add(inode);
168
169     log(String.format(
170         "Register argument \\\%s of \\\%s to \\\%s().",
171         $id.text, ((TypedNode)$id.tree).getExprType(), function.getName()
172     ));
173
174     inode.setMemAddr(Pair.with(loops.size() - 1, 0));
175 };
176
177 arguments: argument {
178     argumentCount += 1;
179 } arguments?;
180
181 statement:
182     ^(PRINT expression) |
183     ^(IF exp=expression {
184         symtab.openScope();
185
186         TypedNode ext = (TypedNode)$exp.tree;
187         if (!(ext.getExprType().equals(Type.Primitive.BOOLEAN))) {
188             reporter.error($exp.tree, "Expression must of be of type boolean. Found: " + ext.getExprType() + ".");
189         }
190     } ^ (THEN commands?) {
191         symtab.closeScope();
192         symtab.openScope();
193     } (^ (ELSE commands?))?) {
194         symtab.closeScope();
195     } |
196     ^(w=WHILE{
197         loops.peek().getValue1().push($w.tree);
198     } ex=expression command*) {
199         checkers.type((TypedNode)$ex.tree, Type.Primitive.BOOLEAN);
200     } |
201     ^(r=RETURN<ControlNode> ex=expression){
202         ControlNode ret = (ControlNode)$r.tree;
203         ret.setScope(loops.peek().getValue0());

```

```

204
205     FunctionNode func = (FunctionNode)ret.getScope();
206     TypedNode expr = (TypedNode)$ex.tree;
207
208     if(loops.size() == 1){
209         reporter.error(r, "Return must be used in function.");
210     }
211
212     if (func.getReturnType().getPrimType() == Type.Primitive.AUTO){
213         func.setReturnType(expr.getExprType());
214         log(String.format("Setting '%s' to %s", func.getName(), func.getReturnType()));
215     }
216
217     checkers.typeEQ(expr, Type.Primitive.AUTO, "Return value must have type, not auto (maybe we did not discover it's
type yet?)");
218
219     // Test equivalence of types
220     if (!func.getReturnType().equals(expr.getExprType(), true)){
221         reporter.error(ret, String.format(
222             "Expected %s, but got %s.", func.getReturnType(), expr.getExprType())
223         );
224     }
225 }|
226 b=BREAK<ControlNode>{
227     try{
228         CommonNode loop = loops.peek().getValue1().peek();
229     } catch(EmptyStackException e){
230         reporter.error($b.tree, "'break' outside loop.");
231     }
232
233     ((ControlNode)$b.tree).setScope(loops.peek().getValue0());
234 }|
235 c=CONTINUE<ControlNode>{
236     try{
237         CommonNode loop = loops.peek().getValue1().peek();
238     } catch(EmptyStackException e){
239         reporter.error($c.tree, "'continue' outside loop.");
240     }

```

```

241      ((ControlNode)$c.tree).setScope(loops.peek().getValue0());
242  }|
243  assignment;
244
245
246  assign:
247    ^(a=DEREFERENCE<TypedNode> as=assign){
248      ((TypedNode)$a.tree).setExprType(new Type(
249        Type.Primitive.POINTER, ((TypedNode)$as.tree).getExprType()
250      ));
251    } |
252    ^(expr=EXPR<TypedNode> ex=expression){
253      ((TypedNode)$expr.tree).setExprType(((TypedNode)$ex.tree).getExprType());
254    }|
255    ^(g=GET<TypedNode> value=assign index=expression){
256      assignType = assignType.getInnerType();
257
258
259      ((TypedNode)$g.tree).setExprType(((TypedNode)$value.tree).getExprType());
260    };
261
262
263  assignment: ^(a=ASSIGN id=IDENTIFIER<IdentifierNode>{
264    IdentifierNode inode = (IdentifierNode)$id.tree;
265
266    inode.setRealNode(getID(inode, $id.text));
267
268    assignType = inode.getExprType();
269
270  } ex=assign{
271    // If 'id' is AUTO, infer type from expression
272    if(((inode.getExprType().getPrimType().equals(Type.Primitive.AUTO))) {
273      inode.setExprType((TypedNode)$ex.tree);
274      log(String.format("Setting '%s' to %s", $id.text, inode.getExprType()));
275    }
276
277    TypedNode ext = (TypedNode)$ex.tree;
278    if(!assignType.equals(ext.getExprType(), true)){

```

```

279         reporter.error($a.tree, String.format(
280             "Cannot assign value of %s to variable of %s.",
281             ext.getExprType(), assignType
282         ));
283     }
284 });
285
286 bool_op: AND | OR;
287 same_op: PLUS | MINUS | DIVIDES | MULT | MOD;
288 same_bool_op: EQ | NEQ;
289 same_bool_int_op: LT | GT | LTE | GTE;
290
291 expression_list: expr=expression {
292     TypedNode arg = calling.getArgs().get(argumentCount);
293     TypedNode exp = (TypedNode)$expr.tree;
294
295     if (!arg.getExprType().equals(exp.getExprType()), true){
296         reporter.error(exp, String.format(
297             "Argument %s of %s expected %s, but got %s.",
298             argumentCount + 1, calling.getName(),
299             arg.getExprType(), exp.getExprType()
300         ));
301     }
302
303     argumentCount += 1;
304 } expression_list?;
305
306 expression:
307     operand |
308     ^(c=CALL<TypedNode> id=IDENTIFIER<IdentifierNode>{
309         IdentifierNode idNode = (IdentifierNode)$id.tree;
310         idNode.setRealNode(getID($id.tree, $id.text));
311         FunctionNode func = calling = (FunctionNode)idNode.getRealNode();
312         ((TypedNode)$c.tree).setExprType(func.getReturnType());
313
314         argumentCount = 0;
315     } expression_list? {
316         if (argumentCount != func.getArgs().size()){

```

```

317         reporter.error(func, String.format(
318             "Expected %s arguments, %s given.", func.getArgs().size(), argumentCount
319         ));
320     }
321 }|
322 ^(op=bool_op ex1=expression ex2=expression) {
323     ((TypedNode)$op.tree).setExprType(new Type(Type.Primitive.BOOLEAN));
324     checkers.type((TypedNode)$ex1.tree, Type.Primitive.BOOLEAN);
325     checkers.type((TypedNode)$ex2.tree, Type.Primitive.BOOLEAN);
326 }|
327 ^(op=same_op ex1=expression ex2=expression){
328     TypedNode ext1 = (TypedNode)$ex1.tree;
329
330     if (ext1.getExprType().getPrimType() == Type.Primitive.POINTER){
331         log("Warning: pointer arithmetic is unchecked logic.");
332         ((TypedNode)$op.tree).setExprType(ext1.getExprType());
333     } else {
334         checkers.equal($op.tree, (TypedNode)$ex1.tree, (TypedNode)$ex2.tree);
335     }
336 }|
337 ^(op=same_bool_op ex1=expression ex2=expression){
338     checkers.equal($op.tree, (TypedNode)$ex1.tree, (TypedNode)$ex2.tree);
339     ((TypedNode)$op.tree).setExprType(Type.Primitive.BOOLEAN);
340 }|
341 ^(op=same_bool_int_op ex1=expression ex2=expression){
342     checkers.type((TypedNode)$ex1.tree, Type.Primitive.INTEGER);
343     checkers.type((TypedNode)$ex2.tree, Type.Primitive.INTEGER);
344     ((TypedNode)$op.tree).setExprType(Type.Primitive.BOOLEAN);
345 }|
346 ^(tam=TAM<TypedNode> t=type STRING.VALUE){
347     ((TypedNode)$tam.tree).setExprType(((TypedNode)$t.tree).getExprType());
348 }|
349 ^(p=DEREFERENCE<TypedNode> ex=expression){
350     checkers.type((TypedNode)$ex.tree, Type.Primitive.POINTER, "Cannot dereference non-pointer.");
351
352     // Dereference variable: take over inner type
353     ((TypedNode)$p.tree).setExprType(((TypedNode)$ex.tree).getExprType().getInnerType());
354 }|

```

```

355 ^ (p=AMPERSAND<TypedNode> id=IDENTIFIER<IdentifierNode>){
356     // Make pointer to variable
357     ((IdentifierNode)$id.tree).setRealNode(getID($id.tree, $id.text));
358
359     ((TypedNode)$p.tree).setExprType(new Type(
360         Type.Primitive.POINTER, ((IdentifierNode)$id.tree).getExprType()
361     ));
362 }|
363 ^ (get=GET<TypedNode> id=IDENTIFIER<IdentifierNode> ex=expression){
364     IdentifierNode inode = (IdentifierNode)$id.tree;
365     inode.setRealNode(getID($id.tree, $id.text));
366
367     checkers.symbol(get, "get_from_array", "builtins/math");
368     checkers.type(inode, Type.Primitive.ARRAY);
369     checkers.type((TypedNode)$ex.tree, Type.Primitive.INTEGER);
370
371     // Result returns inner type of array
372     ((TypedNode)$get.tree).setExprType(inode.getExprType().getInnerType());
373 }|
374 ^ (n=NOT<TypedNode> ex=expression){
375     checkers.type((TypedNode)$ex.tree, Type.Primitive.BOOLEAN);
376 }|
377 ^ (p=POWER<TypedNode> base=expression power=expression){
378     checkers.equal((TypedNode)$p.tree, (TypedNode)$base.tree, (TypedNode)$power.tree);
379     checkers.symbol((TypedNode)$p.tree, "power", "builtins/math");
380 };
381
382 type:
383     primitive_type |
384     composite_type ;
385
386 primitive_type:
387     i=INTEGER<TypedNode>      { ((TypedNode)$i.tree).setExprType(Type.Primitive.INTEGER); }|
388     b=BOOLEAN<TypedNode>      { ((TypedNode)$b.tree).setExprType(Type.Primitive.BOOLEAN); }|
389     c=CHARACTER<TypedNode>    { ((TypedNode)$c.tree).setExprType(Type.Primitive.CHARACTER); }|
390     a=AUTO<TypedNode>         { ((TypedNode)$a.tree).setExprType(Type.Primitive.AUTO); }|
391     v=VAR<TypedNode>          { ((TypedNode)$v.tree).setExprType(Type.Primitive.VARIABLE); };
392

```



```

393 composite_type:
394     ^(arr=ARRAY<TypedNode> t=primitive_type size=expression){
395         TypedNode size = (TypedNode)$size.tree;
396         if(!size.getExprType().equals(Type.Primitive.INTEGER)){
397             reporter.error($size.tree, "Expected Type<INTEGER> but found " + size.getExprType());
398         }
399
400         ((TypedNode)$arr.tree).setExprType(new Type(
401             Type.Primitive.ARRAY, ((TypedNode)$t.tree).getExprType()
402         ));
403
404         checkers.symbol($size.tree, "alloc", "builtins/heap");
405     }|
406     ^(a=DEREFERENCE<TypedNode> t=type){
407         ((TypedNode)$a.tree).setExprType(new Type(
408             Type.Primitive.POINTER, ((TypedNode)$t.tree).getExprType()
409         ));
410     };
411
412 array_expression:
413     ex=expression{
414         Type arrType = arrays.peek().getExprType();
415         Type expType = ((TypedNode)$ex.tree).getExprType();
416
417         if(arrType.getInnerType().getPrimType() == Type.Primitive.AUTO){
418             // We are the first one!
419             arrType.setInnerType(expType);
420         }
421
422         if(arrType.getInnerType().getPrimType() == Type.Primitive.AUTO){
423             // If this type is *still* AUTO, we do not know what to do.
424             reporter.error($ex.tree, String.format(
425                 "Cannot assign AUTO types to an array of AUTO."
426             ));
427         }
428
429         // Checking type against previous array element (essentially)
430         if (!arrType.getInnerType().equals(expType)){

```

```

431         reporter.error($ex.tree, String.format(
432             "Elements of array must be of same type. Found: \%, expected \%,.",
433             expType, arrType.getInnerType()
434         ));
435     }
436 };
437
438 operand:
439     id=IDENTIFIER<IdentifierNode> {
440         ((IdentifierNode)$id.tree).setRealNode(getID($id.tree, $id.text));
441     } |
442     n=NUMBER {
443         ((TypedNode)$n.tree).setExprType(Type.Primitive.INTEGER);
444     } |
445     s=STRING_VALUE {
446         ((TypedNode)$s.tree).setExprType(Type.Primitive.CHARACTER);
447     } |
448     b=(TRUE|FALSE) {
449         ((TypedNode)$b.tree).setExprType(Type.Primitive.BOOLEAN);
450     } |
451     ^(arr=ARRAY<TypedNode> {
452         TypedNode array = (TypedNode)$arr.tree;
453         array.setExprType(Type.Primitive.ARRAY);
454         array.getExprType().setInnerType(new Type(Type.Primitive.AUTO));
455         arrays.push(array);
456     } values=array_expression*) {
457         arrays.pop();
458     }
459 ;

```

De Codegenerator

```

1 tree grammar GrammarChecker;
2
3 options {
4     tokenVocab=Grammar;
5     ASTLabelType=CommonNode;

```

```

6      output=AST;
7  }
8
9  @rulecatch {
10     catch(RecognitionException e){
11         throw e;
12     }
13 }
14
15 @header {
16     package checker;
17     import java.util.Stack;
18     import java.util.EmptyStackException;
19     import symtab.SymbolTable;
20     import symtab.SymbolTableException;
21     import symtab.IdEntry;
22     import ast.*;
23     import reporter.Reporter;
24     import org.javatuples.Pair;
25 }
26
27 // Alter code generation so catch-clauses get replaced with this action.
28 // This disables ANTLR ERROR handling: CalcExceptions are propagated upwards.
29 @members {
30     protected SymbolTable<IdEntry> symtab = new SymbolTable<>();
31
32     private IdentifierNode getID(CommonNode node, String id) throws InvalidTypeException{
33         if (symtab.retrieve(id) == null){
34             reporter.error(node, "Could not find symbol.");
35         }
36         return symtab.retrieve(id).getNode();
37     }
38
39     //
40     private Checkers checkers = new Checkers(this);
41
42     // Upon evaluating pointer assignments (b\% = 3, for example) we need to keep track
43     // of the current type while descending the tree: (ASSIGN b (\% 3)).

```

```

44     private Type assignType;
45
46     // Used to keep track of arguments in a function definition call
47     private int argumentCount;
48
49     // Used to keep track of currently called function, which is used by 'expression_list'
50     // to verify the correctness of given types.
51     private FunctionNode calling;
52
53     // Keep a stack of all loops within functions.
54     private Stack<Pair<FunctionNode, Stack<CommonNode>>>> loops = new Stack<Pair<FunctionNode, Stack<CommonNode>>>>();
55
56     // Keep track of array literals
57     private Stack<TypedNode> arrays = new Stack<>();
58
59     public Reporter reporter;
60     public void setReporter(Reporter r){ this.reporter = r; }
61     public void log(String msg){ this.reporter.log(msg); }
62
63 }
64
65
66 program
67 @init {
68     symtab.openScope();
69 }
70 @after {
71     symtab.closeScope();
72     loops.pop();
73 }
74 : ^(p=PROGRAM<FunctionNode>{
75     loops.push(Pair.with((FunctionNode)$p.tree, new Stack<CommonNode>()));
76     loops.peek().getValue0().setName("--root--");
77     loops.peek().getValue0().setMemAddr(Pair.with(loops.size() - 1, -1));
78 } command+);
79
80 commands: command commands?;
81 command: declaration | expression | statement | ^(PROGRAM command+);

```

```

82
83 declaration: var_declaration | scope_declaration;
84
85 var_declaration:
86     ^(VAR t=type id=IDENTIFIER<IdentifierNode>){
87         IdentifierNode var = (IdentifierNode)$id.tree;
88
89         try {
90             symtab.enter($id.text, new IdEntry(var));
91         } catch (SymbolTableException e) {
92             reporter.error($id.tree, String.format(
93                 "but variable %s was already declared %s",
94                 $id.text, reporter.pointer(symtab.retrieve($id.text).getNode())
95             ));
96         }
97
98         // Copy expression type of 't'
99         var.setExprType(((TypedNode)$t.tree).getExprType());
100
101         // Disallow variable type
102         if (var.getExprType().containsVariableType()){
103             reporter.error($id.tree, "Variable cannot have variable type.");
104         }
105
106         // Register variable with function
107         FunctionNode func = loops.peek().getValue0();
108         func.getVars().add(var);
109         var.setMemAddr(Pair.with(loops.size() - 1, func.getVars().size() - 1));
110
111         log(String.format(
112             "Set relative memory address of %s to (%s, %s)",
113             $id.text, var.getMemAddr().getValue0(), var.getMemAddr().getValue1()
114         ));
115
116         // Set scope to textual function scope
117         var.setScope(func);
118         log(String.format("Setting scope of %s to %s().", $id.text, func.getName()));
119     };

```

```

120
121 scope_declaration: func_declaration;
122
123 func_declaration:
124   ^(FUNC id=IDENTIFIER<FunctionNode> t=type{
125     // Set name and parent of function
126     FunctionNode func = (FunctionNode)$id.tree;
127     func.setName($id.text);
128     func.setScope(loops.peek().getValue0());
129     func.setExprType(Type.Primitive.FUNCTION);
130     log(String.format("Setting %s.parent = %s", $id.text, func.getScope().getName()));
131
132     // Register new scope of looping
133     loops.push(Pair.with(func, new Stack<CommonNode>()));
134
135     try {
136       symtab.enter($id.text, new IdEntry((IdentifierNode)$id.tree));
137     } catch (SymbolTableException e) {
138       reporter.error($id.tree, String.format(
139         "but variable %s was already declared %s",
140         $id.text, reporter.pointer(symtab.retrieve($id.text).getNode())
141       ));
142     }
143
144     func.setReturnType(((TypedNode)$t.tree).getExprType());
145     symtab.openScope();
146     func.setMemAddr(Pair.with(loops.size() - 1, -1));
147
148     // Disallow nesting deeper than 6 levels (limitation in TAM, as the pseudoregisters
149     // L1, L2... only exist up to L6).
150     //
151     // TODO: Implement our own pseudoregisters (resolving static links dynamically)
152     if (loops.size() > 6 + 2){ // +2 for root node, and current function declaration
153       reporter.error(func, "You can only nest functions 6 levels deep, it's morally wrong to nest deeper ;-).");
154     }
155
156     argumentCount = 0;
157   } ^(ARGS arguments?) {

```

```

158 // Set memory addresses for arguments
159 IdentifierNode arg;
160 int inverse_count;
161 for (int i=0; i < func.getArgs().size(); i++){
162     arg = func.getArgs().get(i);
163     inverse_count = -1 * (func.getArgs().size() - i);
164     arg.setMemAddr(Pair.with(loops.size() - 1, inverse_count));
165
166     log(String.format(
167         "Setting relative address of %s to (%s, %s).",
168         arg.getText(), loops.size() - 1, inverse_count
169     ));
170 }
171 } ^(BODY commands?) {
172     symtab.closeScope();
173     loops.pop();
174 };
175
176 argument: t=type id=IDENTIFIER<IdentifierNode>{
177     IdentifierNode inode = (IdentifierNode)$id.tree;
178
179     // Code duplication! :(
180     try {
181         symtab.enter($id.text, new IdEntry(inode));
182         ((TypedNode)$id.tree).setExprType(((TypedNode)$t.tree).getExprType());
183     } catch (SymbolTableException e) {
184         reporter.error(inode, e.getMessage());
185     }
186
187     // Register argument with FunctionNode
188     FunctionNode function = loops.peek().getValue0();
189     function.getArgs().add(inode);
190
191     log(String.format(
192         "Register argument %s of %s to %s().",
193         $id.text, ((TypedNode)$id.tree).getExprType(), function.getName()
194     ));
195

```

```

196 // Set memory address of node, which counts backwards for stack-based models
197 inode.setMemAddr(Pair.with(loops.size() - 1, 0));
198 };
199
200 arguments: argument {
201     argumentCount += 1;
202 } arguments?;
203
204 statement:
205     ^(PRINT expression) |
206     ^(IF exp=expression {
207         symtab.openScope();
208
209         // Expression must be of type boolean.
210         TypedNode ext = (TypedNode)$exp.tree;
211         if (!(ext.getExprType().equals(Type.Primitive.BOOLEAN))) {
212             reporter.error($exp.tree, "Expression must of be of type boolean. Found: " + ext.getExprType() + ".");
213         }
214     } ^ (THEN commands?) {
215         symtab.closeScope();
216         symtab.openScope();
217     } ^ (ELSE commands?)?) {
218         symtab.closeScope();
219     } |
220     ^ (w=WHILE {
221         // Add this loop to the stack of current loops
222         loops.peek().getValue1().push($w.tree);
223     } ex=expression command*) {
224         checkers.type((TypedNode)$ex.tree, Type.Primitive.BOOLEAN);
225     } |
226     ^ (r=RETURN<ControlNode> ex=expression) {
227         // Set parent (function) of this control node (break, continue)
228         ControlNode ret = (ControlNode)$r.tree;
229         ret.setScope(loops.peek().getValue0());
230
231         FunctionNode func = (FunctionNode)ret.getScope();
232         TypedNode expr = (TypedNode)$ex.tree;
233

```



```

234 // PROGRAM is a special 'function node', but doesn't allow return statements.
235 if(loops.size() == 1){
236     reporter.error(r, "Return must be used in function.");
237 }
238
239 // Set expression type of function if type inference requested
240 if (func.getReturnType().getPrimType() == Type.Primitive.AUTO){
241     func.setReturnType(expr.getExprType());
242     log(String.format("Setting '%s' to %s", func.getName(), func.getReturnType()));
243 }
244
245 // If we don't know the type of 'expr', throw an error.
246 checkers.typeEQ(expr, Type.Primitive.AUTO, "Return value must have type, not auto (maybe we did not discover it's
type yet?)");
247
248 // Test equivalence of types
249 if (!func.getReturnType().equals(expr.getExprType(), true)){
250     reporter.error(ret, String.format(
251         "Expected %s, but got %s.", func.getReturnType(), expr.getExprType())
252     );
253 }
254 }|
255 b=BREAK<ControlNode>{
256     try{
257         CommonNode loop = loops.peek().getValue1().peek();
258     } catch(EmptyStackException e){
259         reporter.error($b.tree, "'break' outside loop.");
260     }
261
262     ((ControlNode)$b.tree).setScope(loops.peek().getValue0());
263 }|
264 c=CONTINUE<ControlNode>{
265     try{
266         CommonNode loop = loops.peek().getValue1().peek();
267     } catch(EmptyStackException e){
268         reporter.error($c.tree, "'continue' outside loop.");
269     }
270 }

```

```

271         ((ControlNode)$c.tree).setScope(loops.peek().getValue0());
272     }|
273     assignment;
274
275 assign:
276     ^(a=DEREFERENCE<TypedNode> as=assign){
277         ((TypedNode)$a.tree).setExprType(new Type(
278             Type.Primitive.POINTER, ((TypedNode)$as.tree).getExprType()
279         ));
280     } |
281     ^(expr=EXPR<TypedNode> ex=expression){
282         ((TypedNode)$expr.tree).setExprType(((TypedNode)$ex.tree).getExprType());
283     }|
284     ^(g=GET<TypedNode> value=assign index=expression){
285         assignType = assignType.getInnerType();
286
287
288         ((TypedNode)$g.tree).setExprType(((TypedNode)$value.tree).getExprType());
289     };
290
291
292 assignment: ^(a=ASSIGN id=IDENTIFIER<IdentifierNode>{
293     IdentifierNode inode = (IdentifierNode)$id.tree;
294
295     // Retrieve identifier from symtab.
296     inode.setRealNode(getID(inode, $id.text));
297
298     // Set assign type, so we can use it in 'assign'
299     assignType = inode.getExprType();
300
301 } ex=assign{
302     // If 'id' is AUTO, infer type from expression
303     if(((inode.getExprType().getPrimType().equals(Type.Primitive.AUTO))){
304         inode.setExprType((TypedNode)$ex.tree);
305         log(String.format("Setting '%s' to %s", $id.text, inode.getExprType()));
306     }
307
308     TypedNode ext = (TypedNode)$ex.tree;

```

```

309     if (!assignType.equals(ext.getExprType(), true)){
310         reporter.error($a.tree, String.format(
311             "Cannot assign value of %s to variable of %s.",
312             ext.getExprType(), assignType
313         ));
314     }
315 });
316
317 bool_op: AND | OR;
318 same_op: PLUS | MINUS | DIVIDES | MULT | MOD;
319 same_bool_op: EQ | NEQ;
320 same_bool_int_op: LT | GT | LTE | GTE;
321
322 expression_list: expr=expression {
323     TypedNode arg = calling.getArgs().get(argumentCount);
324     TypedNode exp = (TypedNode)$expr.tree;
325
326     if (!arg.getExprType().equals(exp.getExprType(), true)){
327         reporter.error(exp, String.format(
328             "Argument %s of %s expected %s, but got %s.",
329             argumentCount + 1, calling.getName(),
330             arg.getExprType(), exp.getExprType()
331         ));
332     }
333
334     argumentCount += 1;
335 } expression_list?;
336
337 expression:
338     operand |
339     ^(c=CALL<TypedNode> id=IDENTIFIER<IdentifierNode>{
340         IdentifierNode idNode = (IdentifierNode)$id.tree;
341         idNode.setRealNode(getID($id.tree, $id.text));
342         FunctionNode func = calling = (FunctionNode)idNode.getRealNode();
343         ((TypedNode)$c.tree).setExprType(func.getReturnType());
344
345         // Reset argumentCount to allow counting the numbers of arguments following
346         argumentCount = 0;

```

```

347 } expression_list? {
348     // Compare the number of arguments given with the number of needed arguments
349     if (argumentCount != func.getArgs().size()){
350         reporter.error(func, String.format(
351             "Expected %s arguments, %s given.", func.getArgs().size(), argumentCount
352         ));
353     }
354 }|
355 ^(op=bool_op ex1=expression ex2=expression) {
356     ((TypedNode)$op.tree).setExprType(new Type(Type.Primitive.BOOLEAN));
357     checkers.type((TypedNode)$ex1.tree, Type.Primitive.BOOLEAN);
358     checkers.type((TypedNode)$ex2.tree, Type.Primitive.BOOLEAN);
359 }|
360 ^(op=same_op ex1=expression ex2=expression){
361     TypedNode ext1 = (TypedNode)$ex1.tree;
362
363     if(ext1.getExprType().getPrimType() == Type.Primitive.POINTER){
364         log("Warning: pointer arithmetic is unchecked logic.");
365         ((TypedNode)$op.tree).setExprType(ext1.getExprType());
366     } else {
367         checkers.equal($op.tree, (TypedNode)$ex1.tree, (TypedNode)$ex2.tree);
368     }
369 }|
370 ^(op=same_bool_op ex1=expression ex2=expression){
371     checkers.equal($op.tree, (TypedNode)$ex1.tree, (TypedNode)$ex2.tree);
372     ((TypedNode)$op.tree).setExprType(Type.Primitive.BOOLEAN);
373 }|
374 ^(op=same_bool_int_op ex1=expression ex2=expression){
375     checkers.type((TypedNode)$ex1.tree, Type.Primitive.INTEGER);
376     checkers.type((TypedNode)$ex2.tree, Type.Primitive.INTEGER);
377     ((TypedNode)$op.tree).setExprType(Type.Primitive.BOOLEAN);
378 }|
379 ^(tam=TAM<TypedNode> t=type STRING.VALUE){
380     ((TypedNode)$tam.tree).setExprType(((TypedNode)$t.tree).getExprType());
381 }|
382 ^(p=DEREFERENCE<TypedNode> ex=expression){
383     checkers.type((TypedNode)$ex.tree, Type.Primitive.POINTER, "Cannot dereference non-pointer.");
384 }

```

```

385 // Dereference variable: take over inner type
386 ((TypedNode)$p.tree).setExprType(((TypedNode)$ex.tree).getExprType().getInnerType());
387 }|
388 ^ (p=AMPERSAND<TypedNode> id=IDENTIFIER<IdentifierNode>){
389 // Make pointer to variable
390 ((IdentifierNode)$id.tree).setRealNode(getID($id.tree, $id.text));
391
392 ((TypedNode)$p.tree).setExprType(new Type(
393     Type.Primitive.POINTER, ((IdentifierNode)$id.tree).getExprType()
394 ));
395 }|
396 ^ (get=GET<TypedNode> id=IDENTIFIER<IdentifierNode> ex=expression){
397     IdentifierNode inode = (IdentifierNode)$id.tree;
398     inode.setRealNode(getID($id.tree, $id.text));
399
400     checkers.symbol(get, "get_from_array", "builtins/math");
401     checkers.type(inode, Type.Primitive.ARRAY);
402     checkers.type((TypedNode)$ex.tree, Type.Primitive.INTEGER);
403
404 // Result returns inner type of array
405 ((TypedNode)$get.tree).setExprType(inode.getExprType().getInnerType());
406 }|
407 ^ (n=NOT<TypedNode> ex=expression){
408     checkers.type((TypedNode)$ex.tree, Type.Primitive.BOOLEAN);
409 }|
410 ^ (p=POWER<TypedNode> base=expression power=expression){
411     checkers.equal((TypedNode)$p.tree, (TypedNode)$base.tree, (TypedNode)$power.tree);
412     checkers.symbol((TypedNode)$p.tree, "power", "builtins/math");
413 };
414
415 type:
416     primitive_type |
417     composite_type ;
418
419 primitive_type:
420     i=INTEGER<TypedNode>      { ((TypedNode)$i.tree).setExprType(Type.Primitive.INTEGER); }|
421     b=BOOLEAN<TypedNode>      { ((TypedNode)$b.tree).setExprType(Type.Primitive.BOOLEAN); }|
422     c=CHARACTER<TypedNode>    { ((TypedNode)$c.tree).setExprType(Type.Primitive.CHARACTER); }|

```

```

423     a=AUTO<TypedNode>          { ((TypedNode)$a.tree).setExprType(Type.Primitive.AUTO); }|
424     v=VAR<TypedNode>           { ((TypedNode)$v.tree).setExprType(Type.Primitive.VARIABLE); };
425
426 composite_type:
427     ^(arr=ARRAY<TypedNode> t=primitive_type size=expression){
428         TypedNode sizen = (TypedNode)$size.tree;
429         if(!sizen.getExprType().equals(Type.Primitive.INTEGER)){
430             reporter.error($size.tree, "Expected Type<INTEGER> but found " + sizen.getExprType());
431         }
432
433         // Set type of ARRAY
434         ((TypedNode)$arr.tree).setExprType(new Type(
435             Type.Primitive.ARRAY, ((TypedNode)$t.tree).getExprType()
436         ));
437
438         // We need alloc/free for declaration
439         checkers.symbol($size.tree, "alloc", "builtins/heap");
440     }|
441     ^(a=DEREFERENCE<TypedNode> t=type){
442         ((TypedNode)$a.tree).setExprType(new Type(
443             Type.Primitive.POINTER, ((TypedNode)$t.tree).getExprType()
444         ));
445     };
446
447 array_expression:
448     ex=expression{
449         Type arrType = arrays.peek().getExprType();
450         Type expType = ((TypedNode)$ex.tree).getExprType();
451
452         if(arrType.getInnerType().getPrimType() == Type.Primitive.AUTO){
453             // We are the first one!
454             arrType.setInnerType(expType);
455         }
456
457         if(arrType.getInnerType().getPrimType() == Type.Primitive.AUTO){
458             // If this type is *still* AUTO, we do not know what to do.
459             reporter.error($ex.tree, String.format(
460                 "Cannot assign AUTO types to an array of AUTO."

```

```

461         ));
462     }
463
464     // Checking type against previous array element (essentially)
465     if (!arrType.getInnerType().equals(expType)){
466         reporter.error($ex.tree, String.format(
467             "Elements of array must be of same type. Found: \%, expected \%, ",
468             expType, arrType.getInnerType()
469         ));
470     }
471 };
472
473 operand:
474     id=IDENTIFIER<IdentifierNode> {
475         ((IdentifierNode)$id.tree).setRealNode(getID($id.tree, $id.text));
476     } |
477     n=NUMBER {
478         ((TypedNode)$n.tree).setExprType(Type.Primitive.INTEGER);
479     } |
480     s=STRING.VALUE {
481         ((TypedNode)$s.tree).setExprType(Type.Primitive.CHARACTER);
482     } |
483     b=(TRUE|FALSE) {
484         ((TypedNode)$b.tree).setExprType(Type.Primitive.BOOLEAN);
485     } |
486     ^(arr=ARRAY<TypedNode> {
487         TypedNode array = (TypedNode)$arr.tree;
488         array.setExprType(Type.Primitive.ARRAY);
489         array.getExprType().setInnerType(new Type(Type.Primitive.AUTO));
490         arrays.push(array);
491     } values=array_expression*){
492         arrays.pop();
493     }
494 ;

```

Listing 9.1: De codegenerator

9.4 Testverslag

Dit zijn de python bestanden die alle tests afhandelen. De output bij incorrect broncode wordt afgevangen in de tests zelf en wordt dus niet als output gegeven. Er is alleen noemenswaardige output als de test faalt, dus er niet opgeleverd wordt wat verwacht is. Met de huidige test komt dit niet voor.

De code die alles test.