SELMA

Vonk, J
s0132778
Matenweg 75-201

Florisson, M
s000000
Box Calslaan xx-30

June 30, 2011

Contents

1 Inleiding

Voor vertalerbouw dient als eindopdracht een eigen taal geschreven te worden. Deze taal dient een expression-language te zijn, dit is een taal die geen statements, maar enkel expressies kent. Alles wat je dus aanroept zal een waarde teruggeven.
Voor deze zelfbedachte taal dient een parser en lexer geschreven te worden, een checker en een compiler. Hierbij dient een verslag met een uitgebreide beschrijving van de taal en een goede kijk op hoe alles onder de motorkap werkt. Ook moet er een bewijs worden geleverd dat de taal werkt, dit kan door een testprogramma te schrijven dat tamelijk uitgebreid is en te kijken of dit werkt naar behoren. (exhaustive testing)
Hoe uitgebreid de te definieren taal wordt is aan de studenten zelf - dit is echter terug te zien in het te behalen cijfer.
Voor onze taal, SELMA, hebben wij gekozen voor het volgende: **Klopt?**

- Basic Expression Language
- If- & while-statements
- Ondersteunen van functies
- Compileren naar JVM-code in plaats van TAM-code

Onze taal heet SELMA. Een naam aan een taal geven is lastig, zo waren er een aantal andere opties zoals: SMEF of Taal voor Vertalerbouw (TV). SELMA staat voor Simpel Expression Language. Nu moest de afkorting wat meer zeggen dus kozen we voor de meisjesnaam SELMA, alleen maar omdat een afkorting vinden voor SELDERIE wel heel veel werk is.
Gelukkig heet onze taal dus geen SELDERIE, maar SELMA:
Waarbij de MA voor Minor Adjustments stond, we hebben inmiddels zoveel werk eraan gehad dat "Minor" dat geen eer meer aan doet.
Dus met gepaste trots presenteren wij u SELMA:
Simple Expression Language Met Augurk
Vanaf nu enkel nog naar te verwijzen als SELMA.

2 Beknopte beschrijving

Onze taal is gemaakt naar de gegeven instructies van de practicumhandleiding en alles is of een expressie of declaration in deze taal. Bij sommige expressies is het echter niet mogelijk een resultaat te geven, hier kunnen die expressies niet anders dan een void-resultaat retouneren, wat ze effectief een statement maakt. De structuur van de taal en de keywords lijken qua layout op een hybride tussen C en Pascal.
De volledige taal is LL(1) wij hebben hierdoor vooral tijdens het ontwerpen goed moeten nadenken hoe we de taal zo logisch mogelijk opbouwden zodat de parser er mee uit de voeten kon. Eventueel is er de mogelijkheid om lokaal 1 stap verder te kijken, wij hebben dit echter niet nodig gehad omdat wij voldoende keywords hebben gebruikt, zoals voor een functie een @ zetten - en we in de parser bewust rekening hebben gehouden met de LL(1) limitatie.

@?

SELMA compileerde in eerste instantie naar TAM, op de cd is een fragment van deze code te zien. We hebben echter besloten dat het mooier was om JVM te gebruiken, niet zo zeer uit praktisch oogpunt, maar meer omdat JVM-bytecode ook door "echte" talen wordt gebruikt en omdat het een pluspunt is in de eindbeoordeling.
Op het moment dat we besloten om te schakelen waren we blij dat we haddne gekozen voor het gebruik van stringTemplates bij de codegeneratie, dit heeft ons wat werk gescheeld. En technisch gezien zouden we zo een extra compiler naar TAM-code erbij kunnen doen, aangezien er geen andere reden is dan "omdat het kan" hebben we onszelf die moeite bespaard.

Lees verder - of probeer eens een testprogramma te compileren in SELMA - om te leren hoe de vork nou precies in de steel zit met deze taal.

- Mark & Jeroen

3 Problemen en oplossingen

uitleg over de wijze waarop je de problemen die je bent tegenge- komen bij het maken van de opdracht hebt opgelost (maximaal twee A4-tjes).

4 Syntax, context-beperkingen en semantiek

4.1 Lexer - terminals

Om de code te kunnen parsen zal deze eerst door de lexer moeten gaan. Hier definieren wij een aantal terminal symbolen. Dit is een eindige set van een aantal symbolen of woorden, de lexer zal deze herkennen. Mits ze in de juiste volgorde worden gebruikt krijg je taalconstructies die de parser vervolgens weer begrijpt. We hebben een aantal speciale terminals die zijn opgebouwd uit meerdere karakters bijvoorbeeld. Deze vormen de lexicon. En een drietal terminals zonder textuele vorm. Deze zijn enkel voor de interne boekhouding van de parser.

```
CHARV           APOSTROPHE LETTER APOSTROPHE;
BOOLEAN         (TRUE | FALSE);
ID              LETTER (LETTER | DIGIT)*;
NUMBER          DIGIT+;

DIGIT           ('0'..'9');
LOWER           ('a'..'z');
UPPER           ('A'..'Z');
LETTER          (LOWER | UPPER);
TRUE            'true';
FALSE           'false';

UMIN;
UPLUS;
COMPOUND;
```

Verder zijn er nog de 'gewone' terminals. Te verdelen in keywords, tokens en operators. Keywords geven aan dat er een bepaalde actie gedaan wordt, zoals een variabele declareren of een if statement. Tokens zijn er om de taal iets meer structuur te geven, denk aan comma's tussen de variabelen. En operators zijn bewerkingen die je kunt uitvoeren op 1 of meer expressies.

Tokens		Keywords	
COLON	':';	PRINT	'print';
SEMICOLON	';';	READ	'read';
LPAREN	'(';	VAR	'var';
RPAREN	')';	CONST	'const';
LCURLY	'{';	INT	'integer';
RCURLY	'}';	BOOL	'boolean';
COMMA	',';	CHAR	'character';
EQ	'=';	BEGIN	'begin';
APOSTROPHE	'''';	END	'end.';
		IF	'if';
		THEN	'then';
		ELSE	'else';
		FI	'fi';
		WHILE	'while';
Operators		DO	'do';
		OD	'od';
NOT	'!';	PROC	'procedure';
MULT	'*';	FUNC	'function';
DIV	'/';		
MOD	'%';		
PLUS	'+';		
MINUS	'-';		
RELS	'<';		
RELSE	'<=';		
RELGE	'>=';		
RELG	'>';		
RELE	'==';		
RELNE	'<>';		
AND	'&&';		
OR	'||';		
BECOMES	':=';		

4.2 De basis - Programma

De basis van het programma geeft een aantal restricties op aan de taal. Allereerst is er het programma, dit bestaat uit een (zeer grote) compoundexpression waarna het programma stopt (End Of File). Deze wordt hier herschreven. Een compoundexpression is uiteindelijk opgebouwd uit een serie declaraties en statements, gescheiden door een semicolon. Hier is te zien dat het programma uit minimaal 1 expressie bestaat, dat declaraties en expressies door elkaar gebruikt mogen worden en dat het laatste statement in een programma altijd een expressie is.

```
program
        : compoundexpression EOF
                -> ^(BEGIN compoundexpression END)
        ;

compoundexpression
        : cmp -> ^(COMPOUND cmp)
        ;

cmp
  : ((declaration SEMICOLON!)* expression_statement SEMICOLON! )+
  ;
```

4.3 Expression_statement

Dit is een speciale tussenstap voor de interne boekhouding. Na elke semicolumn zal de mogelijk resterende waarde van de stack worden gepopped. Dit maakt dat er niet aan het eind van ons programma een hoop troep op de stack staat. Voorwaarde is wel dat er wordt bijgehouden wanneer een expression van het type void is, dan hoeft er namelijk niet gepopped te worden.

```
expression_statement
        : expression -> ^(EXPRESSION_STATEMENT expression)
        ;
```

4.4 Declaraties en types

SELMA kent twee soorten waarden-declaraties, variabelen en constanten. SELMA staat toe om per declaratie meerdere identifiers te definieren. Bij de declaratie dien je het type van de te declareren waarde mee te geven. En bij een constante dien je uiteraard een waarde mee te geven.

4.4.1 Syntax

```
declaration
//      : VAR^ identifier (COMMA! identifier)* COLON! type
//      | CONST^ identifier (COMMA! identifier)* COLON! type EQ!
    unsignedConstant
```

```
        : VAR identifier (COMMA identifier)* COLON type
                -> ^(VAR type identifier)+
        | CONST identifier (COMMA identifier)* COLON type EQ
            unsignedConstant
                -> ^(CONST type unsignedConstant identifier)+
        | FUNCDEF^ identifier LPAREN! (identifier (COMMA!
            identifier)* COLUMN! type SEMICOLUMN!)* RPAREN!
            funcbody;
        ;

type
        : INT
        | BOOL
        | CHAR
        ;
```

4.4.2 Context

- Het gegeven type dient bij de constante overeen te komen met het type van de gegeven waarde.
- Identifiers mogen niet eerder gedeclareerd zijn, in de huidige of bovenliggende scope.

4.4.3 Semantiek

Er zal ruimte gereserveerd worden voor de variabele en het adres wordt onthouden. Voor een constante geldt hetzelfde behalve dat dan ook direct de desbetreffende waarde op dat adres wordt gezet. Op het moment dat elders in het programma een verwijzing is naar deze gedeclareerde dan zal deze variabele of constante geladen worden.

4.4.4 Voorbeeld

```
var i, x: integer;
const c: char = 'g';
const b,t: boolean = true;
```

4.5 Functiedeclaratie

SELMA kent ook nog een functie declaratie. Deze valt logischerwijs ook onder de declaraties. De declaratie van een functie dient altijd voor het gebruik te komen. Een functie kan als een soort procedure worden gebruikt door geen return-type op te geven. Het return-type wordt dan automatisch void. Dit hebben we express gedaan, we willen het namelijk altijd een functie noemen, aangezien procedures niet echt een plek hebben binnen een expressietaal.

4.5.1 Syntax

```
        | FUNCDEF^ identifier LPAREN! (identifier (COMMA!
            identifier)* COLUMN! type SEMICOLUMN!)* RPAREN!
            funcbody;
funcbody
        : COLUMN! type LCURLY! compoundexpression FUNCRETURN
            expression SEMICOLUMN! RCURLY!
        | LCURLY! compoundexpression RCURLY!
        ;
```

4.5.2 Context

- De naam van de functie moet uniek zijn als functienaam, er mag wel een variabele of constante bestaan met die naam.
- De opgegeven identifiers moeten allemaal een andere naam hebben, ze hoeven echter niet uniek te zijn binnen het programma aangezien ze in een aparte scope staan.
- Het type van de expressie na het returntype dient hetzelfde te zijn als type.

4.5.3 Semantiek

Het adres waar deze functie staat wordt opgeslagen. Daarna komt de code van de functie. Aan het einde van de functie zal eventueel een result op de stack worden gezet en wordt het adres dat aan het begin is gegeven aangeroepen om weer terug te komen op de plek waar de functie werdt aangeroepen.

4.5.4 Voorbeeld

```
function foo() {
        6*7;
}
function foo(awesome,less:boolean;bar:integer;): integer {
        var i: integer;

        if (awesome;) then
                i := 42;
        else
                i := 2;
        fi;

        return i;
}
```

4.6 Expressies - assignment

De expressies zijn ingedeeld in verschillende niveaus, dit om te zorgen dat ze in de juiste volgorde worden uitgevoerd. Zo willen we dat 6+3*12 niet 108 is maar 42, niet alleen om dat 42 een mooier getal is, maar voornamelijk omdat het fijn is als de taal voldoet aan de conventionele rekenregels.
Het hoogste niveau is de assignment.

4.6.1 Syntax

```
expression
        : expr_assignment
        ;

expr_assignment
        : expr_arithmetic (BECOMES^ expression)?
        ;
```

4.6.2 Context

- expr_arithmetic moet een identifier worden, in het eind, aangezien dat het enige is waaraan je een waarde kunt toekennen
- deze identiefier moet dan verwijzen naar een geldige variabele
- het type van expression en expression_arithmetic moet hetzelfde zijn
- expression is van het type van expr_assignment
- expr_assignment is van het type van expr_arithmetic

4.6.3 Semantiek

De waarde van expression zal worden toegekend aan het linker deel van de assignment. Tevens gaat de waarde van de hele expressie op de stack, zo is er een asignement met meerdere identifiers mogelijk.

4.6.4 Voorbeeld

```
7*6;
foo := 7*6;
foo := bar := 7*6;
```

4.7 Expressies - OR

De Of-operator is de laagste operator in het rijtje, vandaar dat deze bovenin de structuur zit.
NB: expr_al1 staat voor "expression arithmetic level 1"

4.7.1 Syntax

```
expr_arithmetic
        : expr_al1
        ;

        expr_al1                                                //
            expression arithmetic level 1
                : expr_al2 (OR^ expr_al2)*
                ;
```

4.7.2 Context

- Als expr_al1 enkel uit 1 expr_al2 bestaat dan zijn er geen restricties
- In de andere gevallen dienen alle expr_al2 van het type boolean te zijn.
- het type van expr_arithmetic is het type van expr_al1
- als exp_1 == expr_al2 dan is het type van expr_al1 het type van expr_al2
- als exp_1 != expr_al2 dan is het type van expr_al1 een boolean

4.7.3 Semantiek

De eerste expr_al2 zal op de stack worden gezet. Hierna wordt er telkens een expr_al2 erbij gezet. De OR-operatie zal worden aangeroepen en het resultaat blijft op de stack zijn. Als er nog een expr_al2 is dan zal deze ook op de stack worden gezet en wordt de OR-operatie opnieuw aangeroepen. Aldoende blijft er uiteindelijk 1 waarde op de stack staan.

4.7.4 Voorbeeld

```
7*6;
true OR false;
true OR false OR foo;
```

4.8 Expressies - AND

Hier wordt de AND-expressie beschreven. Net zoals bij de OR-expressie is het mogelijk nul tot veel AND-operatoren achter elkaar te plakken. De AND-expressie is een niveau hoger dan de OR-expressie en zal dus eerder worden uitgevoerd.
Het is eventueel mogelijk later in de compiler om een AND eerder af te breken aangezien als er een false in het rijtje zit het resultaat altijd false is. Wij hebben deze optimalisatie er nog niet inzitten, dit omdat sommige expressies ongeacht de eerdere expressies uitgevoerd dienen te worden, denk bijvoorbeeld aan een READ()-statement dat anders niet uitgevoerd zou worden.

4.8.1 Syntax

```
expr_al2
        : expr_al3 (AND^ expr_al3)*
        ;
```

4.8.2 Context

- Als expr_al2 enkel uit 1 expr_al3 bestaat dan zijn er geen restricties
- In de andere gevallen dienen alle expr_al3 van het type boolean te zijn.
- als exp_2 == expr_al3 dan is het type van expr_al2 het type van expr_al3
- als exp_2 != expr_al3 dan is het type van expr_al2 een boolean

4.8.3 Semantiek

Hetzelfde als bij het OR-statement. De waardes zullen op de stack geladen worden en er zal telkens een AND-operatie op 2 waardes worden uitgevoerd. De resulterende waarde is weer geschikt voor bijvoorbeeld nog een AND-operatie.

4.8.4 Voorbeeld

```
7*6;
foo AND bar;
foo AND false AND bar;
```

4.9 Expressies - Relaties

Hier worden bijna alle comperatoren afgehandeld. Het is belangrijk dat er in de checker goed wordt gekeken of de types van de linker en rechterzijde compatible zijn.

4.9.1 Syntax

```
expr_al3
        : expr_al4 ((RELS|RELSE|RELG|RELGE|RELE|RELNE)^
            expr_al4)*
        ;
```

4.9.2 Context

- alle expr_al4 dienen van hetzelfde type te zijn
- bij een operatie tussen twee expr_al4 anders dan RELE & RELNE dient expr_al4 een integer te zijn.
- als exp_3 == expr_al4 dan is het type van expr_al3 het type van expr_al4
- als exp_3 != expr_al4 dan is het type van expr_al3 een boolean

4.9.3 Semantiek

Vergelijkbaar met andere binaire operatoren zoals AND en OR, er zullen waardes op de stack worden gezet en de operatie zal 1 waarde achterlaten op de stack.

4.9.4 Voorbeeld

```
5 > 6;
true == false;
5 == 42;
```

4.10 Expressies - plus en minus

Hier zijn we aangeland bij de eerder genoemde 6+3*12, plus en minus zit 1 niveau lager dan de vermenigvuldigingen.

4.10.1 Syntax

```
        expr_al4
                : expr_al5 ((PLUS|MINUS)^ expr_al5)*
                ;
```

4.10.2 Context

- als er minimaal 1 operatie wordt uitgevoerd dan dient expr_al5 een integer te zijn
- als exp_4 == expr_al5 dan is het type van expr_al4 het type van expr_al5
- als exp_4 != expr_al5 dan is het type van expr_al4 een integer

4.10.3 Semantiek

Wederom een binaire operatie. Let op, de unaire plus en minus komen nog. Dus 5–6 zal de tweede minus niet hier worde opgevangen.

4.10.4 Voorbeeld

```
foo := 5;
foo := 5 + 37;
10 + 50 - 18;
```

4.11 Expressies - delen en vermenigvuldigen

Naast delen en vermenigvuldigen is het ook mogelijk een modulus te nemen. Wat wellicht is opgevallen bij het bovenstaande, is dat het mogelijk is om enkel een som in de code te zetten. Dit vinden wij prima, echter moet daarbij wel de resulterende waarde gepopped worden als die niet meer gebruikt wordt.

4.11.1 Syntax

```
        expr_al5
                : expr_al6 ((MULT|DIV|MOD)^ expr_al6)*
                ;
```

4.11.2 Context

- als er minimaal 1 operatie wordt uitgevoerd dan dient expr_al6 een integer te zijn
- als exp_5 == expr_al6 dan is het type van expr_al5 het type van expr_al6
- als exp_5 != expr_al6 dan is het type van expr_al5 een integer

4.11.3 Semantiek

Hetzelfde als bij optellen. Goed om te weten is dat de geretouneerde waarde een integer is, dus er zal worden afgerond.

4.11.4 Voorbeeld

```
foo := 6;
foo := 6*7;
foo := 21*6%84;
```

4.12 Expressies - unaries

Hier wordt gekeken of de expressie eventueel een NOT-, PLUS- of MIN-operator voor zich heeft staan. Om later verwarring te voorkomen zullen PLUS en MIN vervangen worden door speciale terminals, zijnde UMIN en UPLUS. UPLUS zou eventueel weg kunnen worden gelaten aangezien +x==x. Als er geen operator voor de expressie staat dan is expr_al6 gewoon een expr_al7

4.12.1 Syntax

```
        expr_al6
//              : (PLUS|MINUS|NOT)? expr_al7
                : PLUS expr_al7
                        -> UPLUS expr_al7
                | MINUS expr_al7
                        -> UMIN expr_al7
                | NOT expr_al7
                | expr_al7
                ;
```

4.12.2 Context

- expr_al7 dient bij PLUS expr_al7 een integer te zijn
- expr_al7 dient bij MIN expr_al7 een integer te zijn
- expr_al7 dient bij NOT expr_al7 een boolean te zijn
- het type van expr_al6 het type van expr_al7

4.12.3 Semantiek

Bij UMIN zal expr_al6 == - expr_al7
Bij UPLUS zal expr_al6 == expr_al7
Bij NOT zal expr_al6 == ! expr_al7

4.12.4 Voorbeeld

```
one := +1;
evil := -42;
foo := not foobar;
```

4.13 Expressies - toplevel

Op het hoogste nivau kan een expressie bestaan uit een semi-statement zoals een if-expressie of een print-expressie, of het kan een identifier of waarde zijn, of het kan een aparte (compound)expressie binnen haken zijn. Zoals je ziet stond in eerste de assignment hier. Maar aangezien het meest linkerdeel van een assignment een identiefier is kan op L=1 geen onderscheid worden gemaakt tussen identifier of een assignment. Vandaar dat een assignment bij expr_al1 is gedefineerd.

4.13.1 Syntax

```
        expr_al7
                : unsignedConstant
                | identifier
//              | expr_assignment                //can be identifier
                | expr_read
                | expr_print
                | expr_if
                | expr_while
                | expr_closedcompound
                | expr_closed
                | expr_funccall
                ;
```

4.13.2 Context

- expr_al7 is van hetzelfde type als de gegeven expressie of waarde.

4.13.3 Semantiek

Dit is enkel een lijst van mogelijke expressies en waardes en dus zal er in de compiler enkel deze expressie of waarde op stack hebben staan, maar wordt er geen operatie op uitgevoerd.

4.13.4 Voorbeeld

```
foo;
42;
(foobar);
```

4.14 Unsigned constants

Uiteraard bied onze taal ook de mogelijkheid aan om constanten te gebruiken zonder deze eerst te moeten declareren. Oftewel, je kunt gewoon nummers gebruiken bijvoorbeeld.

4.14.1 Syntax

```
        : boolval
        | charval
        | intval
        ;

intval
        : NUMBER
        ;

boolval
```

```
        : BOOLEAN
        ;

charval
        : CHARV
        ;
  : APOSTROPHE (LETTER|UNDERSCORE) APOSTROPHE
  ;
```

4.14.2 Context

- unsignedconstant is van het type van de gegeven waarde
- boolval is een boolean type
- charval is een char
- intval is een integer

4.14.3 Semantiek

De desbetreffende waarde wordt op de stack gezet.

4.14.4 Voorbeeld

```
'Y';
42;
true;
```

4.15 Identifier

Een identifier van een bestaande variabele of constante in de huidige of een hogere scope.

4.15.1 Syntax

```
identifier
        : ID
        ;
        : LETTER (LETTER | DIGIT)*
        ;
```

4.15.2 Context

- De identifier dient te verwijzen naar een geldige variabele of constante
- Het type is het type van de variabele of declaratie waar de identifier naar verwijst.

4.15.3 Semantiek

Er zal een commando aangeroepen worden om de waarde uit het geheugen te laden. Deze waarde wordt dan op de stack gezet. Bij constanten gebeurd dit ook. Eventueel zou je ook de constante zelf al kunnen neerzetten op de stack, dit scheelt weer wat werk voor de processor. Dit doen wij echter niet momenteel.

4.15.4 Voorbeeld

```
Answer42;
```

4.16 Read

Om contact te hebben met de buitenwereld kan onze taal lezen en schrijven naar de standard-out.

4.16.1 Syntax

```
expr_read
        : READ^ LPAREN! identifier (COMMA! identifier)* RPAREN!
        ;
```

4.16.2 Context

- Identifier dient te verwijzen naar een geldige identifier
- De ingelezen waarde dient van het zelfde type als identifier te zijn
- Als er 1 identifier is opgegeven dan geeft read de gelezen waarde/type terug
- Als er meer dan 1 identifier wordt ingelezen dan is het returntype void

4.16.3 Semantiek

Het read-commando wordt aangeroepen en de waarde wordt van de standard-out gelezen en op de stack gezet. Vervolgens wordt die waarde opgeslagen in de variabele.

4.16.4 Voorbeeld

```
read(foo);
read(foo,bar);
```

4.17 Print

De taal heeft ook de mogelijkheid om dat wat er bijvoorbeeld berekend is naar buiten te communiceren.

4.17.1 Syntax

```
expr_print
        : PRINT^ LPAREN! expression (COMMA! expression)* RPAREN!
                -> ^(PRINT expression+)
```

4.17.2 Context

- -

4.17.3 Semantiek

De waarde van de expressie staat op de stack. Vervolgens wordt deze netjes naar het scherm uitgevoerd. Afhankelijk van het type zal dat anders gebeuren.

4.17.4 Voorbeeld

```
print(42);
print('4','2');
```

4.18 If

Om keuzes in het programma mogelijk te maken zal er een conditioneel statement nodig zijn, het IF-statement is een dergelijk statement. Een ELSE-deel is optioneel.

4.18.1 Syntax

```
expr_if
        : IF^ compoundexpression THEN compoundexpression (ELSE
            compoundexpression)? FI!
        ;
```

4.18.2 Context

- De eerste compoundexpression moet een boolean-type retouneren
- De if retouneerd een type void **retourtype**

4.18.3 Semantiek

Als de waarde binnen het ifstatement waar is dan zal de eerste compoundexpressie worden uitgevoerd (na de then). Anders zal de andere compoundexpressie worden uitgevoerd, mits deze is gedeclareerd.

4.18.4 Voorbeeld

```
if true; then i := 42; fi
if false; then i := 0; else i:=42; fi
```

4.19 While

De while zal net zolang een blok code uitvoeren tot een gegeven expressie waar is.

4.19.1 Syntax

```
expr_while
        : WHILE^ compoundexpression DO compoundexpression OD
        ;
```

4.19.2 Context

- De eerste compoundexpression moet een boolean-type retouneren
- De while retouneerd een type void

4.19.3 Semantiek

De tweede compoundexpression zal worden uitgevoerd tot de eerste compoundexpression waar is. Het kan zijn dat de tweede compoundexpression nooit wordt uitgevoerd dus.

4.19.4 Voorbeeld

```
while false; do
\\ this is not gonna be executed
tru := false;
od

while foo<5; do
foo := foo + 1;
od
```

4.20 Functieaanroep

Een functieaanroep naar een eerder gedefineerde functie

4.20.1 Syntax

```
funccall
        : FUNCTION^ identifier LPAREN! (identifier SEMICOLUMN!)*
          RPAREN!
        ;
```

4.20.2 Context

- Het aantal expressies en hun type dient overeen te komen met de declaratie van de functie
- De functie retouneert het eerder gespecificeerde type. Als er geen type was gedeclareerd dan is dat dus void.

4.20.3 Semantiek

Het returnadres wordt op de stack gezet, zodat de functie weer hiernaartoe kan terugkeren. De expressies worden op de stack gezet in de gespecificeerde volgorde. De functie wordt aangeroepen. De functie returned en het result staat op de stack.

4.20.4 Voorbeeld

```
foo();
i := foo('b','a','r');
```

4.21 Closed expression

Een expressie tussen haakjes is soms handig, bijvoorbeeld bij sommetjes: (5+2)*6;

4.21.1 Syntax

```
expr_closed
        : LPAREN! expression RPAREN!
        ;
```

4.21.2 Context

- De geretouneerde waarde zal de waarde van de expressie zijn binnen de haakjes.
- Het retourneerde type is ook hetzelfde als die van de expressie.

4.21.3 Semantiek

De expressie binnen de haakjes zal worden uitgevoerd binnen de haakjes.

4.21.4 Voorbeeld

```
(3*(6+8))%102;
```

4.22 Closed compoundexpression

Is een compoundexpressie binnen haakjes. Verschil met de expressie tussen haakjes is dat deze ook toestaat om declaraties te gebruiken. Een compound tussen haakjes zal een eigen scope hebben.

4.22.1 Syntax

```
expr_closedcompound
        : LCURLY compoundexpression RCURLY
        ;
```

4.22.2 Context

- retouneerd het waarde en de type van de laatste expressie in de compound, dit kan van het type void zijn.

4.22.3 Semantiek

De compoundexpressie zal in een eigen scope worden uitgevoerd.

4.22.4 Voorbeeld

```
{
var foo: integer;
foo := 40;
foo+2;
}
```

expression_statement

5 Vertaalregels

voor de taal, d.w.z. de transformaties waaruit blijkt op welke wijze een opeenvolging van symbolen die voldoet aan een produktieregel wordt omgezet in een opeenvol- ging van TAM-instructies. Vertaalregels zijn de code templates van hoofdstuk 7 van Watt & Brown.

6 Beschrijving van Java programmatuur

6.1 main - SELMA

SELMA.java is het main-programma. Je kunt een aantal opties en een SELMA-sourcecodefile meegeven. Hierna zal SELMA desbetreffende file parsen en compileren. De opties die mogelijk zijn zijn:

- -ast Er zal een ast-diagram naar de stdOut worden geprint van de source-code.
- -dot Er zal een dot-diagram naar de stdOut worden geprint van de source-code.
- -no_checker De source-code wordt geparsed maar niet gechecked.
- -code_generator De source-code zal worden gecompiled

De sourcecode zal de volgende stappen doorlopen:

Lexer	Parser	-no_checker		-ast	Ast-diagram
Lexer	Parser	-no_checker		-dot	Dot-diagram
Lexer	Parser		Checker	-ast	Ast-diagram
Lexer	Parser		Checker	-dot	Dot-diagram
Lexer	Parser		Checker	-code_generator	Code

Alle resultaten zullen altijd naar de stdOut worden geprint.

6.2 SELMAException

Als er wat fout gaat in bijvoorbeeld de checker dan zal er een exception worden gegooid. Deze exception is een SELMAException. Aan de exception wordt de node meegegeven waar de checker op dat moment mee bezig is. En de toString()-functie van SELMAException zal dat dan ook mooi formatten in de vorm van "(regelnummer:columnnummer) Errormessage", toch wel fijn als je moet debuggen.

6.3 SELMATreeAdaptor

Deze TreeAdaptor heeft SELMATree als nodes, in plaats van een normale Tree.

6.4 SELMATree

SELMATree is een uitbreiding op de normale tree. En kan een aantal extra dingen bijhouden, namelijk of een expressie constant is of variabel, wat later handig is voor optimizing. En wat het type is van de expressie, dat is zeer handig voor de checker. Daarvoor heeft SELMATree een paar extra attributen, zijnde:

```
public enum SR_Type {INT,BOOL,CHAR,VOID};
public enum SR_Kind {VAR, CONST};

public SR_Type SR_type = null;
public SR_Kind SR_kind = null;
```

En verder kent SELMATree nog drie functies om mooi te kunnen printen:

```
public String toStringTree() {
public String toStringTree(int level) {
public String toString() {
```

6.5 SymbolTable

De symboltable houdt al onze variabelen en constanten bij. Ook kun je in de symboltable scopes aanmaken, om bijvoorbeeld variabelen binnen een compoundexpressie te kunnen declareren. De dataopslag van de symboltable geschiedt middels een Map waarin een string aan een stack van IDEntries wordt gekoppeld. De string verwijst naar de naam van de variabele of constante. De stack bevat meerdere declaraties van die variabele met die naam in verschillende scopes. Zodat het mogelijk is de zelfde naam tweemaal te gebruiken, mits ze in een andere scope gebruikt worden.

De symboltable kent een aantal functies, de belangrijkste zijn:

```
/**
 * Opens a new scope.
 * @ensure  this.currentLevel() == old.currentLevel()+1
 */
public void openScope()  {
/**
 * Closes the current scope. All identifiers in
 * the current scope will be removed from the SymbolTable.
 * @require old.currentLevel() > -1
 * @ensure  this.currentLevel() == old.currentLevel()-1
 */
public void closeScope() {
/**
 * Enters an id together with an entry into this SymbolTable
     using the
 * current scope level. The entry's level is set to
     currentLevel().
 * @require String != null && String != "" && entry != null
 * @ensure  this.retrieve(id).getLevel() == currentLevel()
 * @throws  SymbolTableException when there is no valid current
      scope level,
 *          or when the id is already declared on the current
     level.
 */
public void enter(Tree tree, Entry entry) throws
    SymbolTableException {
/**
```

```
    * Get the Entry corresponding with id whose level is the
      highest.
    * In other words, the method returns the Entry that is defined
      last.
    * @return  Entry of this id on the highest level
    *          null if this SymbolTable does not contain id
    */
   public Entry retrieve(Tree tree) throws SymbolTableException{
```

6.5.1 SymbolTableException

SymbolTableException is er om fouten in de symboltable aan te geven. Deze fouten zullen vergelijkbaar worden geformat als die van SELMAException, namelijk "(line:column) ErrorMsg.

6.6 IDEntry

De symboltable bevat voor elke variabele of constante een IDEntry. Een IDEntry bevat de scopelevel van desbetreffende declaratie. Wij gebruiken in onze code echter een tweetal klasses die ge-extend zijn op IDEntry; CheckerEntry en CompilerEntry.

6.7 CheckerEntry

De CheckerEntry wordt gebruikt in de Checker. Een checkerEntry verschilt van een IDEntry op het punt dat een checkerEntry twee extra waardes heeft om bij te houden wat het type is van de variabele of constante (int,bool of char). De tweede waarde is om bij te houden of we met een constante of een variabele te maken hebben.

```
    public SR_Type type;
    public SR_Kind kind;
```

6.8 CompilerEntry

De compilerEntry is weer een uitbreiding op de CheckerEntry. Voor de compiler is het namelijk noodzakelijk om te weten op welk adres in de te genereren code de variabele staat. Dit wordt bijgehouden door:

```
        public int addr;
```

7 Testplan en -resultaten

Bespreking van de correctheids-tests aan de hand van de criteria zoals deze zijn beschreven in het A.5 van deze appendix. Aan de hand van deze criteria moet een verzameling test-programmas in het taal geschreven worden die de juiste werking van de vertaler en interpreter controleren. Tot deze test-set behoren behalve correcte programmas die de verschillende taalconstructies testen, ook programmas met syntactische, semantische en run-time fouten. Alle uitgevoerde tests moeten op de CD-R aanwezig zijn; van een testprogramma moet de uitvoer in de appendix opgenomen worden (zie onder).

8 Conclusies

9 Appendix

9.1 ANTLR Lexer & Parser specificatie

```
grammar SELMA;


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          = ')';
        LCURLY          = '{';
        RCURLY          = '}';
        COMMA           = ',';
        EQ              = '=';
        APOSTROPHE      = '\'';
        UNDERSCORE      = '_';
        //arethemithic
        NOT             = '!';

        MULT            = '*';
        DIV             = '/';
        MOD             = '%';

        PLUS            = '+';
        MINUS           = '-';

        RELS            = '<';
        RELSE           = '<=';
        RELGE           = '>=';
        RELG            = '>';
        RELE            = '==';
        RELNE           = '<>';

        AND             = '&&';

        OR              = '||';

        //expressions
        BECOMES         = ':=';
        PRINT           = 'print';
        READ            = 'read';

        //declaration
        VAR             = 'var';
        CONST           = 'const';

        //types
        INT             = 'integer';
        BOOL            = 'boolean';
        CHAR            = 'character';
        //keywords
        IF              = 'if';
        THEN            = 'then';
        ELSE            = 'else';
        FI              = 'fi';

        WHILE           = 'while';
        DO              = 'do';
        OD              = 'od';
```

```
        FUNCDEF = 'function ';
        FUNCRETURN = 'return ';
        FUNCTION = '@';

        UMIN;
        UPLUS;

        BEGIN;
        END;
        COMPOUND;
        EXPRESSION_STATEMENT;
}

@header {
  package SELMA;
}

@lexer::header {
  package SELMA;
}


// Parser rules - program at line 90 due to the report

program
        : compoundexpression EOF
                -> ^(BEGIN compoundexpression END)
        ;

compoundexpression
        : cmp -> ^(COMPOUND cmp)
        ;

cmp
  : ((declaration SEMICOLON!)* expression_statement SEMICOLON! )+
  ;

//declaration

declaration
//      : VAR^ identifier (COMMA! identifier)* COLON! type
//      | CONST^ identifier (COMMA! identifier)* COLON! type EQ!
    unsignedConstant
        : VAR identifier (COMMA identifier)* COLON type
                -> ^(VAR type identifier)+
        | CONST identifier (COMMA identifier)* COLON type EQ
            unsignedConstant
                -> ^(CONST type unsignedConstant identifier)+
        | FUNCDEF^ identifier LPAREN! (identifier (COMMA! identifier)*
          COLUMN! type SEMICOLUMN!)* RPAREN! funcbody;
        ;

type
        : INT
        | BOOL
        | CHAR
        ;

funcbody
        : COLUMN! type LCURLY! compoundexpression FUNCRETURN expression
          SEMICOLUMN! RCURLY!
        | LCURLY! compoundexpression RCURLY!
        ;

expression_statement
        : expression -> ^(EXPRESSION_STATEMENT expression)
```

```
        ;
// note: - arithmetic can be "invisible" due to all the *-s that's why
    it is nested
// - assignment can be "invisible" due to the ? that's why it can also
    be only a identifier
expression
        : expr_assignment
        ;

expr_assignment
        : expr_arithmetic (BECOMES^ expression)?
        ;

expr_arithmetic
        : expr_al1
        ;

        expr_al1                                                //expression
            arithmetic level 1
                : expr_al2 (OR^ expr_al2)*
                ;

        expr_al2
                : expr_al3 (AND^ expr_al3)*
                ;

        expr_al3
                : expr_al4 ((RELS|RELSE|RELG|RELGE|RELE|RELNE)^ expr_al4
                   )*
                ;

        expr_al4
                : expr_al5 ((PLUS|MINUS)^ expr_al5)*
                ;

        expr_al5
                : expr_al6 ((MULT|DIV|MOD)^ expr_al6)*
                ;

        expr_al6
//              : (PLUS|MINUS|NOT)? expr_al7
                : PLUS expr_al7
                        -> UPLUS expr_al7
                | MINUS expr_al7
                        -> UMIN expr_al7
                | NOT expr_al7
                | expr_al7
                ;

        expr_al7
                : unsignedConstant
                | identifier
//              | expr_assignment               //can be identifier
                | expr_read
                | expr_print
                | expr_if
                | expr_while
                | expr_closedcompound
                | expr_closed
                | expr_funccall
                ;
expr_read
        : READ^ LPAREN! identifier (COMMA! identifier)* RPAREN!
        ;

expr_print
        : PRINT^ LPAREN! expression (COMMA! expression)* RPAREN!
                -> ^(PRINT expression+)
```

```
        ;
expr_if
        : IF^ compoundexpression THEN compoundexpression (ELSE
          compoundexpression)? FI!
        ;

expr_while
        : WHILE^ compoundexpression DO compoundexpression OD
        ;

funccall
        : FUNCTION^ identifier LPAREN! (identifier SEMICOLUMN!)* RPAREN!
        ;

expr_closedcompound
        : LCURLY compoundexpression RCURLY
        ;

expr_closed
        : LPAREN! expression RPAREN!
        ;

unsignedConstant
        : boolval
        | charval
        | intval
        ;

intval
        : NUMBER
        ;

boolval
        : BOOLEAN
        ;

charval
        : CHARV
        ;

identifier
        : ID
        ;

CHARV
  : APOSTROPHE (LETTER|UNDERSCORE) APOSTROPHE
  ;

BOOLEAN
        : TRUE
        | FALSE
        ;

ID
        : LETTER (LETTER | DIGIT)*
        ;

NUMBER
        : DIGIT+
        ;

COMMENT
        : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
        | '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
        ;
WS
        : ( ' '
        | '\t'
```

```
            |  '\r'
            |  '\n'
            ) {$channel=HIDDEN;}
            ;

fragment DIGIT
            : ('0'..'9')
            ;

fragment LOWER
            : ('a'..'z')
            ;

fragment UPPER
            : ('A'..'Z')
            ;

fragment LETTER
   : LOWER
   | UPPER
   ;

fragment TRUE
            : 'true'
            ;

fragment FALSE
            : 'false'
            ;


//EOF
```

9.2 ANTLR Checker specificatie

```
tree grammar SELMAChecker;

options {
        tokenVocab=SELMA;
        ASTLabelType=SELMATree;
        output=AST;
}

@header {
        package SELMA;
        import SELMA.SELMATree.SR_Type;
        import SELMA.SELMATree.SR_Kind;
}

// Alter code generation so catch-clauses get replaced with this action.
@rulecatch {
        catch (RecognitionException re) {
                throw re;
        }
}

@members {
        public SymbolTable<CheckerEntry> st = new SymbolTable<
            CheckerEntry>();
}

program
    : ^(node=BEGIN
        {st.openScope();}
        compoundexpression
        {$node.localsCount = st.getLocalsCount(); st.closeScope();}
        END)
        ;

compoundexpression //do not open and close scope here (IF/WHILE)
        : ^(node=COMPOUND (declaration|expression_statement)+)
        {
            SELMATree e1 = (SELMATree)node.getChild(node.getChildCount()
                -1);
            if (e1.SR_type==SR_Type.VOID) {
                $node.SR_type=SR_Type.VOID;
                $node.SR_kind=null;
            } else {
                $node.SR_type=e1.SR_type;
                $node.SR_kind=e1.SR_kind;
            }
        }
        ;

expression_statement
        : ^(node=EXPRESSION_STATEMENT expression)
        {
            SELMATree e1 = (SELMATree)node.getChild(node.getChildCount()
                -1);
            $node.SR_type = e1.SR_type;
            $node.SR_kind = e1.SR_kind;
        }
        ;

declaration
        : ^(node=VAR type id=ID)
   {
   int type = node.getChild(0).getType();

   switch (type){
     case INT:
```

```
      st.enter($id,new CheckerEntry(SR_Type.INT,SR_Kind.VAR));
      break;
      case BOOL:
      st.enter($id,new CheckerEntry(SR_Type.BOOL,SR_Kind.VAR));
      break;
      case CHAR:
      st.enter($id,new CheckerEntry(SR_Type.CHAR,SR_Kind.VAR));
      break;
    }
    }
          | ^(node=CONST type val id=ID)
           {
    int type = node.getChild(0).getType();
    int val  = node.getChild(1).getType();

           switch (type){
      case INT:
      if (val!=NUMBER) throw new SELMAException(id,"Expecting int-value")
           ;
      st.enter($id,new CheckerEntry(SR_Type.INT,SR_Kind.CONST));
      break;
      case BOOL:
      if (val!=BOOLEAN) throw new SELMAException(id,"Expecting bool-value
        ");
      st.enter($id,new CheckerEntry(SR_Type.BOOL,SR_Kind.CONST));
      break;
      case CHAR:
      if (val!=CHARV) throw new SELMAException(id,"Expecting char-value")
           ;
      st.enter($id,new CheckerEntry(SR_Type.CHAR,SR_Kind.CONST));
      break;
           }
           }
          ;

type
  : INT
  | BOOL
  | CHAR
  ;

val
  : NUMBER
  | CHARV
  | BOOLEAN
  ;

expression
        : ^(node=(MULT|DIV|MOD|PLUS|MINUS) expression expression)
         {
    SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(1);

    if (e1.SR_type!=SR_Type.INT || e2.SR_type!=SR_Type.INT)
     throw new SELMAException($node,"Wrong type must be int");
    $node.SR_type=SR_Type.INT;

    if (e1.SR_kind==SR_Kind.CONST && e2.SR_kind==SR_Kind.CONST)
     $node.SR_kind=SR_Kind.CONST;
    else
     $node.SR_kind=SR_Kind.VAR;
    }

          | ^(node=(RELS|RELSE|RELG|RELGE) expression expression)
    {
    SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(1);
```

```
    if (e1.SR_type!=SR_Type.INT || e2.SR_type!=SR_Type.INT)
130  throw new SELMAException($node,"Wrong type must be int");
    $node.SR_type=SR_Type.BOOL;

    if (e1.SR_kind==SR_Kind.CONST && e2.SR_kind==SR_Kind.CONST)
     $node.SR_kind=SR_Kind.CONST;
135 else
     $node.SR_kind=SR_Kind.VAR;
    }

         | ^(node=(OR|AND) expression expression)
140 {
    SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(1);

    if (e1.SR_type!=SR_Type.BOOL || e2.SR_type!=SR_Type.BOOL)
145  throw new SELMAException($node,"Wrong type must be bool");
    $node.SR_type=SR_Type.BOOL;

    if (e1.SR_kind==SR_Kind.CONST && e2.SR_kind==SR_Kind.CONST)
     $node.SR_kind=SR_Kind.CONST;
150 else
     $node.SR_kind=SR_Kind.VAR;
    }

         | ^(node=(RELE|RELNE) expression expression)
155 {
    SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(1);

    if (e1.SR_type!=e2.SR_type||e1.SR_type==SR_Type.VOID)
160  throw new SELMAException($node,"Types must match and can't be void")
        ;
    $node.SR_type=SR_Type.BOOL;

    if (e1.SR_kind==SR_Kind.CONST && e2.SR_kind==SR_Kind.CONST)
     $node.SR_kind=SR_Kind.CONST;
165 else
     $node.SR_kind=SR_Kind.VAR;
    }

         | ^(node=(UPLUS|UMIN) expression)
170 {
    SELMATree e1 = (SELMATree)node.getChild(0);

    if (e1.SR_type!=SR_Type.INT)
     throw new SELMAException($node,"Wrong type must be int");
175 $node.SR_type=SR_Type.INT;

    $node.SR_kind=e1.SR_kind;
    }

180      | ^(node=(NOT) expression)
    {
    SELMATree e1 = (SELMATree)node.getChild(0);

    if (e1.SR_type!=SR_Type.BOOL)
185  throw new SELMAException($node,"Wrong type must be bool");
    $node.SR_type=SR_Type.BOOL;

    $node.SR_kind=e1.SR_kind;
    }
190
         | ^(node=IF {st.openScope();} compoundexpression
              THEN {st.openScope();} compoundexpression {st.closeScope()
                  ;}
              (ELSE {st.openScope();} compoundexpression {st.closeScope()
                   ;})?
```

```
            {st.closeScope();})
    {
    SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(2);
    SELMATree e3 = (SELMATree)node.getChild(4);

     if (e1.SR_type!=SR_Type.BOOL)
       throw new SELMAException(e1,"Expression must be boolean");

     if (e3==null) { //no else
       $node.SR_type=SR_Type.VOID;
       $node.SR_kind=null;
     } else { // there is a else
       if (e2.SR_type==e3.SR_type) {
         $node.SR_type=e3.SR_type;
           if (e2.SR_kind==SR_Kind.CONST && e3.SR_kind==SR_Kind.CONST)
             $node.SR_kind=SR_Kind.CONST;
           else
             $node.SR_kind=SR_Kind.VAR;
       } else {
         $node.SR_type=SR_Type.VOID;
         $node.SR_kind=null;
       }
     }
     }

         | ^(node=WHILE {st.openScope();}compoundexpression
              DO {st.openScope();} compoundexpression {st.closeScope();}
              OD{st.closeScope();})
    {
    SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(2);

    if (e1.SR_type!=SR_Type.BOOL)
     throw new SELMAException(e1,"Expression must be boolean");

    $node.SR_type=SR_Type.VOID;
    $node.SR_kind=null;

    }
/*
         | ^(node=READ (id=ID
            {
              if (st.retrieve($id).kind!=SR_Kind.VAR)
                throw new SELMAException($id,"Must be a variable");
            })+
              {
                if ($node.getChildCount()==1){
                 $node.SR_type=((SELMATree)node.getChild(0)).SR_type;
          $node.SR_kind=SR_Kind.VAR;
        } else {
          $node.SR_type=SR_Type.VOID;
          $node.SR_kind=null;
        }
              }
            )
*/
         | ^(node=PRINT expression+)
           {
     for (int i=0; i<((SELMATree)node).getChildCount(); i++){
       if (((SELMATree)node.getChild(i)).SR_type == SR_Type.VOID)
         throw new SELMAException($node, "Can not be of type void");
     }
        if ($node.getChildCount() == 1){
           $node.SR_type = ((SELMATree) node.getChild(0)).SR_type;
           $node.SR_kind = SR_Kind.VAR;
        } else {
          $node.SR_type = SR_Type.VOID;
```

```
            $node.SR_kind = null;
        }
            }
265
          | ^(node=BECOMES expression expression)
            {
    SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(1);
270 if (e1.getType()!=ID)
     throw new SELMAException(e1,"Must be a identifier");

    CheckerEntry ident = st.retrieve(e1);

275 if (ident.kind!=SR_Kind.VAR)
     throw new SELMAException(e1,"Must be a variable");
    if (ident.type!=e2.SR_type)
     throw new SELMAException(e1,"Right side must be the same type "+
          ident.type+"/"+e2.SR_type);

280 $node.SR_type=ident.type;
    $node.SR_kind=SR_Kind.VAR;
          }

          | LCURLY {st.openScope();} compoundexpression {st.closeScope();}
              RCURLY
285
          | node=NUMBER
            {
            $node.SR_type=SR_Type.INT;
            $node.SR_kind=SR_Kind.CONST;
290         }

          | node=BOOLEAN
    {
    $node.SR_type=SR_Type.BOOL;
295 $node.SR_kind=SR_Kind.CONST;
    }

          | node=CHARV
    {
300 $node.SR_type=SR_Type.CHAR;
    $node.SR_kind=SR_Kind.CONST;
    }

          | node=ID
305         {
            CheckerEntry entry = st.retrieve($node);
            $node.SR_type=entry.type;
            $node.SR_kind=entry.kind;
            }
310       ;
```

9.3 ANTLR Codegenerator specificatie

```
tree grammar SELMACompiler;

options {
  language = Java;
  output = template;
  tokenVocab = SELMA;
  ASTLabelType = SELMATree;
}

@header {
  package SELMA;
  import SELMA.SELMA;
  import SELMA.SELMATree.SR_Type;
  import SELMA.SELMATree.SR_Kind;
}

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

@members {
    public SymbolTable<CompilerEntry> st = new SymbolTable<CompilerEntry
        >();
    int curStackDepth;
    int maxStackDepth;

    int labelNum = 0;

    private void incrStackDepth() {
        if (++curStackDepth > maxStackDepth)
            maxStackDepth = curStackDepth;
    }

    private String getTypeDenoter(SR_Type type) {
        if (type == SR_Type.INT) {
            return "I";
        } else if (type == SR_Type.BOOL) {
            return "Ljava/lang/String;";
        } else {
            return "C";
        }
    }

    private void printSingle(SELMATree expr, boolean isExpr,
        StringBuilder sb) {
          String typeDenoter = getTypeDenoter(expr.SR_type);

          if (isExpr)
              sb.append("    dup\n");

          if (expr.SR_type == SR_Type.BOOL) {
              int label1 = labelNum++, label2 = labelNum++;
              sb.append(String.format(
                      "    ifeq L\%s\n"    +
                      "    ldc \"true\"\n" +
                      "    goto L\%s\n"    +
                      "L\%s\n"             +
                      "    ldc \"false\"\n" +
                      "L\%s\n", label1, label2, label1, label2));
          }

          sb.append(String.format(
                  "    getstatic java/lang/System/out Ljava/io/PrintStream
                      ;\n" +
```

```
                    "     swap\n" +
                    "     invokevirtual java/io/PrintStream/print(\%s)V\n",
                        getTypeDenoter(expr.SR_type)));
    }
}

program
  : ^(node=BEGIN {st.openScope();} compoundexpression {st.closeScope();}
      END)
  { SELMATree expr = (SELMATree) $node.getChild(0); }
  -> program(instructions={$compoundexpression.st},
             source_file={SELMA.inputFilename},
             stack_limit={maxStackDepth + 2}, // +2 for print and other
                 additionally loaded constants
             locals_limit={$node.localsCount + 1}, // +1 for the String
                 [] argv parameter
             pop={expr.SR_type != SR_Type.VOID})
  ;

compoundexpression
  : ^(node=COMPOUND (s+=declaration  | s+=expression_statement)+)
  -> compound(instructions={$s}, line={node.getLine()}, pop={$node.
      SR_type != SR_Type.VOID})
  ;

declaration
  : ^(node=VAR INT id=ID)
  {st.enter($id,new CompilerEntry(SR_Type.INT,SR_Kind.VAR,st.nextAddr())
      ); }
  -> declareVar(id={$id.text},type={"INT"},addr={st.nextAddr()-1})

  | ^(node=VAR BOOL id=ID)
  {st.enter($id,new CompilerEntry(SR_Type.BOOL,SR_Kind.VAR,st.nextAddr()
      )); }
  -> declareVar(id={$id.text},type={"BOOL"},addr={st.nextAddr()-1})

  | ^(node=VAR CHAR id=ID)
  {st.enter($id,new CompilerEntry(SR_Type.CHAR,SR_Kind.VAR,st.nextAddr()
      )); }
  -> declareVar(id={$id.text},type={"CHAR"},addr={st.nextAddr()-1})

  // store the const at a address? LOAD Or just copy LOADL?
  | ^(node=CONST INT val=NUMBER (id=ID)+)
  {st.enter($id,new CompilerEntry(SR_Type.INT,SR_Kind.CONST,st.nextAddr
      ())); }
  -> declareConst(id={$id.text},value={$val.text},type={"INT"},addr={st.
      nextAddr()-1})

  | ^(node=CONST BOOL BOOLEAN (id=ID)+)
  {st.enter($id,new CompilerEntry(SR_Type.BOOL,SR_Kind.CONST,st.nextAddr
      ()-1)); }
  -> declareConst(id={$id.text},value={($val.text.equals("true"))
      ?"1":"0"},type={"BOOL"},addr={st.nextAddr()})

  | ^(node=CONST CHAR CHARV (id=ID)+)
  {st.enter($id,new CompilerEntry(SR_Type.CHAR,SR_Kind.CONST,st.nextAddr
      ())); }
  -> declareConst(id={$id.text},value={Character.getNumericValue($val.
      text.charAt(1))},type={"CHAR"},addr={st.nextAddr()-1})
  ;


expression_statement
  : ^(node=EXPRESSION_STATEMENT e1=expression) { curStackDepth--; }
  -> exprStat(e1={e1.st}, line={$node.getLine()}, pop={$node.SR_type !=
      SR_Type.VOID})
  ;
```

```
expression
//double arg expression
  : ^(node=MULT e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st}, e2={$e2.st}, instr={"imul"}, line={node.getLine
       ()}, op={"*"})

  | ^(node=DIV e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st},e2={$e2.st},instr={"idiv"}, line={node.getLine()
       }, op={"/"})

  | ^(node=MOD e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st},e2={$e2.st},instr={"imod"}, line={node.getLine()
       }, op={"\%"})

  | ^(node=PLUS e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st},e2={$e2.st},instr={"iadd"}, line={node.getLine()
       }, op={"+"})

  | ^(node=MINUS e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st},e2={$e2.st},instr={"isub"}, line={node.getLine()
       }, op={"-"})

  | ^(node=OR e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st},e2={$e2.st},instr={"or"}, line={node.getLine()},
       op={"or"})

  | ^(node=AND e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st},e2={$e2.st},instr={"and"}, line={node.getLine()
       }, op={"and"})

  | ^(node=RELS e1=expression e2=expression) { curStackDepth--; }
  -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmplt"}, line={node.
       getLine()},
                op={"<"}, label_num1={labelNum++}, label_num2={labelNum
                    ++})

  | ^(node=RELSE e1=expression e2=expression) { curStackDepth--; }
  -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmple"}, line={node.
       getLine()},
                op={"<="}, label_num1={labelNum++}, label_num2={labelNum
                    ++})

  | ^(node=RELG e1=expression e2=expression) { curStackDepth--; }
  -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpgt"}, line={node.
       getLine()},
                op={">"}, label_num1={labelNum++}, label_num2={labelNum
                    ++})

  | ^(node=RELGE e1=expression e2=expression) { curStackDepth--; }
  -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpge"}, line={node.
       getLine()},
                op={">="}, label_num1={labelNum++}, label_num2={labelNum
                    ++})

  | ^(node=RELE e1=expression e2=expression) { curStackDepth--; }
  -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpeq"}, line={node.
       getLine()},
                op={"="}, label_num1={labelNum++}, label_num2={labelNum
                    ++})

  | ^(node=RELNE e1=expression e2=expression) { curStackDepth--; }
  -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpne"}, line={node.
       getLine()},
                op={"!="}, label_num1={labelNum++}, label_num2={labelNum
                    ++})

//single arg expression
  | ^(UPLUS e1=expression)
```

```
  {$st=$e1.st;}

  | ^(node=UMIN e1=expression)
  -> uExpr(e1={$e1.st}, instr={"ineg"}, line={node.getLine()}, op={"-"})

  | ^(node=NOT e1=expression)
  -> biExprJump(e1={$e1.st}, e2={"iconst_0"}, instr={"if_icmpeq"}, line
      ={node.getLine()},
                op={"not"}, label_num={labelNum++})

//CONDITIONAL
  | ^(node=IF ec1=compoundexpression THEN ec2=compoundexpression (ELSE
      ec3=compoundexpression)?)
        { boolean ec3NotEmpty = $ec3.st != null;
          SELMATree expr2 = (SELMATree) node.getChild(2);
          SELMATree expr3 = null;
          if (ec3NotEmpty)
              expr3 = (SELMATree) node.getChild(4);
        }
  -> if(ec1={$ec1.st},ec2={$ec2.st},ec3={$ec3.st}, label_num1={labelNum
      ++},
        label_num2={ec3NotEmpty ? labelNum++ : 0}, ec3_not_empty={
            ec3NotEmpty},
        is_void={!ec3NotEmpty || expr2.SR_type != expr3.SR_type || expr2
            .SR_type == SR_Type.VOID})

  | ^(node=WHILE ec1=compoundexpression DO ec2=compoundexpression OD)
  { curStackDepth--; SELMATree expr2 = (SELMATree) node.getChild(2); }
  -> while(ec1={$ec1.st}, ec2={$ec2.st}, pop={expr2.SR_type != SR_Type.
      VOID}, label_num1={labelNum++}, label_num2={labelNum++})

//IO
  | ^(node=READ (ids=ID)+)
    /*
        { boolean isExpr = $node.SR_type != SR_Type.VOID;
          String typeDenoter = getTypeDenoter($node.SR_type);
          boolean isBool = $node.SR_type == SR_Type.BOOL;
      if (!isExpr)
          curStackDepth -= $node.getChildrenCount();
        }
        -> read(ids={$ids.st}, type_denoter={typeDenoter}, dup_top={
            isExpr},
                label_num1={labelNum++}, label_num2={labelNum++},
            line={node.getLine()})
    */
  | ^(node=PRINT (exprs+=expression)+)
        {
        boolean isExpr = $node.SR_type != SR_Type.VOID;
        int childCount = ((SELMATree) node).getChildCount();
        StringBuilder sb = new StringBuilder();

        System.err.println($node.getChild(0));
        System.err.println($node.getChild(1));
        System.err.println($node.getChild(2));

        sb.append(String.format(".line \%s\n", $node.getLine()));

        if (isExpr) {
            // print(e1) - this is an expression
            printSingle((SELMATree) $node.getChild(0), true, sb);
        } else {
            // print(e1, e2, ...) - this is NOT an expression

            // Not an expression, don't reserve space on the stack
            // for all the results of the expressions
            curStackDepth -= childCount;

            for (int i = 0; i < childCount; i++) {
```

```
                printSingle((SELMATree) $node.getChild(i), false, sb);
            }
        }
230
    }
    -> print(exprs={exprs}, code={sb.toString()})

//ASSIGN
235 | ^(BECOMES node=ID e1=expression) { boolean isint = ($node.type ==
      NUMBER ||
                                        $node.type == BOOLEAN ||
                                        $node.type == LETTER); }
        -> assign(id={$node.text},
                  type={$node.type},
240               addr={st.retrieve($node).addr},
                      e1={$e1.st},
                      isint={isint})

//closedcompound
245 | LCURLY {st.openScope();} compoundexpression {st.closeScope();}
      RCURLY

//VALUES
  | node=NUMBER { incrStackDepth();
                  int num = Integer.parseInt($node.text); }
250   -> loadNum(val={$node.text}, iconst={num >= -1 && num <= 5}, bipush
        ={num >= -128 && num <= 127})

  | node=BOOLEAN { incrStackDepth(); }
    -> loadNum(val={($node.text.equals("true")) ? 1 : 0}, iconst={true})

255 | node=CHARV { incrStackDepth();
                 char c = $node.text.charAt(1); }
    //-> loadNum(val={(int) c}, iconst={false}, bipush={true})
    -> loadChar(val={(int) c}, char={$node.text}, line={$node.getLine()
        })

260 | node=ID { incrStackDepth(); }
    -> loadVal(id={$node.text}, addr={st.retrieve($node).addr})
  ;
```

9.4 ANTLR Codegenerator Stringtemplate specificatie

```
//SELMA string template

group SELMA;

program(instructions, source_file, stack_limit, locals_limit, pop) ::= <<
.source <source_file>
.class  public Main
.super  java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack <stack_limit>
.limit locals <locals_limit>
.line 1
<instructions>
<if (pop)>
    pop

<endif>
    return
.end method
>>

compound(instructions, line, pop) ::= <<
.line <line>
<instructions; separator="\n">
<if (pop)>
    removeLastInstruction  ; line <line>
<endif>
>>

exprStat(e1, pop, line) ::= <<
.line <line>
<e1>
<if (pop)>
    pop
<endif>
>>

//Calculations
uExpr(e1, instr, line, op)::=<<
.line <line>                    ; <op> <e1>
```

```
<e1>
    <instr>
>>

biExpr(e1, e2, instr, line, op) ::= <<
.line <line>
<e1>
<e2>
    <instr>
>>

biExprJump(e1, e2, instr, label_num1, label_num2, line, op) ::= <<
.line <line>
<e1>
<e2>
    <instr> L<label_num1>         ; e1 <op> e2
    iconst_0
    goto L<label_num2>
L<label_num1>:
    iconst_1
L<label_num2>:
>>

//Declare
declareConst(id,val,type,addr)::=<<

>>

declareVar(id,type,addr)::=<<

>>

//Load

loadNum(val, iconst, bipush) ::= <<
<if (iconst)>
    iconst_<val>
<elseif (bipush)>
    bipush <val>

<else>
    ldc <val>
```

```
<endif>
>>

loadVal(id, addr)::=<<
    iload <addr>                ; load <id> from <addr>
>>

loadChar(val, char, line) ::= <<
.line <line>
    bipush <val>                ; ldc <char>
>>

//Assign
assign(id, type, addr, e1, isint) ::= <<
<e1>                            ; e1 right hand for assignment
    dup
    istore <addr>
>>

read(addrs, type_denoters, dup_top, label_num1, label_num2, line) ::= <<
    read here                   ; reading <id>
<if (dup_top)>
    dup

<endif>
    istore <addr>
>>

print(exprs, code) ::= <<
<exprs>
<code>
>>

/*
printSingle(expr, type_denoter, dup_top, is_bool, label_num1, label_num2, line) ::= <<
<expr>
.line <line>
<if (dup_top)>
    dup
<endif>

<if (is_bool)>
```

```
    ifeq L<label_num1>
    ldc "true"
    goto L<label_num2>
L<label_num1>:
    ldc "false"
L<label_num2>:
<endif>

    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/print(<type_denoter>)V
>>    */

//conditionals
if (ec1, ec2, ec3, label_num1, label_num2, ec3_not_empty, is_void) ::= <<
<ec1>                                                   ; e1 condition
    ifeq L<label_num1>          ; e1 is false
<ec2>                           ; e2 if true expression
<if (is_void)>
    pop                         ; is_void <is_void>

<endif>
<if (ec3_not_empty)>
    goto L<label_num2>
<endif>

L<label_num1>:
<if (ec3_not_empty)>
<ec3>                                                   ; e3 if false expression
<if (is_void)>
    pop                         ; is_void <is_void>

<endif>
L<label_num2>:
<endif>
>>

while(ec1, ec2, label_num1, label_num2)::=<<
L<label_num1>:
<ec1>                                                   ; e1 while condition
    ifeq L<label_num2>
<ec2>                                                   ; e2 expression to evaluate (body)
```

```
<if (pop)>
    pop

<endif>
    goto L<label_num1>
L<label_num2>:
>>
```

9.5 Invoer- en uitvoer van een uitgebreid testprogramma

Van een correct en uitgebreid test- programma (met daarin alle features van uw programmeertaal) moet worden bijgevoegd: de listing van het oorspronkelijk programma, de listing van de gegenereerde TAM-code (be- standsnaam met extensie .tam) en een of meer executie voorbeelden met in- en uitvoer waaruit de juiste werking van de gegenereerde code blijkt.