SELMA

Vonk, J
s0132778
Matenweg 75-201

Florisson, M
s0165972
Box Calslaan 60-30

July 5, 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! (funcpars SEMICOLON!)* RPAREN
            ! funcbody
        ;
funcpars : identifier (COMMA identifier)* COLON type -> (identifier
     type)+;
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! (funcpars SEMICOLON!)* RPAREN
            ! funcbody
funcbody
        : COLON type LCURLY compoundexpression FUNCRETURN
            expression SEMICOLON RCURLY -> ^(FUNCRETURN type
            compoundexpression expression)
        | 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 expr_al7
                -> ^(UPLUS expr_al7)
        | MINUS expr_al7
                -> ^(UMIN expr_al7)
        | NOT 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

```
unsignedConstant
        : boolval
        | charval
        | intval
        ;

intval
        : NUMBER
        ;
```

```
boolval
        : BOOLEAN
        ;

charval
        : CHARV
        ;
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
        ;
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

```
expr_funccall
        : FUNCTION^ identifier LPAREN! (expression COMMA!)* 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

Deze sectie specificeerd hoe selma programmas naar Jasmin worden vertaald. Jasmin is een assembler die gegeven Jasmin assembly JVM bytecode genereerd in de vorm van een `.class` file.

De vertaling wordt gedaan door middel van een compiler (g-files/SELMACompiler.g) in ANTLR die executeerd na de checker en producties uit de Jasmin string template aanroept (SELMACodeJasmin.stg).

We gebruiken de volgende code functies:

- run : Program → Instruction*
- execute : ExpressionStatement → Instruction*
- evaluate : Expression → Instruction*
- elaborate : Declaration → Instruction*

Phase	Code Function	Effect
Program	run P	Run P en begin en eindig met een lege stack. Doe ook de nodige declaraties om classen en methoden te genereren. Return hierna.
ExprStat	execute E	Executeer statement S. Als S een expressie is dat een waarde op de stack genereerd, pop. Dit veranderd de stack niet.
Expression	evaluate E	Evalueer expressie E en laat de nieuwe waarde op de stack staan. Een expressie kan 1 of 2 oude waarden van de stack halen (e.g. AND).
Declaration	elaborate	Behandel een declaratie voor constanten, variabelen en functies. Dit maakt entries aan in de symbol table en genereerd simpele declaratie code zoals methoden en velden.

In onze vertaalregels zullen we variabelen omringen met <en >. Soms zijn er hulpvariabelen nodig die geen deel uitmaken van de syntax. We zullen dit aanduiden als extra argumenten aan de code function, e.g. `evaluate[ID := E, is_global, type]`, waarbij `is_global` en `type` extra argumenten zijn die geen deel uitmaken van de syntax maar van de context.

5.1 Run

5.1.1 Program

```
run[P, source_file, stack_limit, locals_limit, pop] =
  .source <source_file>
  .class public Main
  .super java/lang/Object
  .field public static scanner_field Ljava/Util/Scanner;

  .method public static main([Ljava/lang/String;)V
  .limit stack <stack_limit>
```

```
  .limit locals <locals_limit>
    new java/util/Scanner
    dup
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
    putstatic Main/scanner_field Ljava/util/Scanner;

    evaluate[P]
    if <pop>:
        pop

    return
.end method
```

De `source_file`, `stack_limit` en `locals_limit` geven respectievelijk aan wat de originele source file was (voor runtime excepties), de grootte van de stack en het aantal locale variabelen dat het programma gebruikt. `pop` geeft aan of P een expressie was en nog een waarde op de stack heeft achtergelaten. Dit zou ook opgelost kunnen worden door de regel

```
program -> expression_statement
expression_statement -> expression
expression -> ... | compoundexpression
compoundexpression -> (declaration | expression_statement)*
```

in plaats van

```
program -> compoundexpression
```

Helaas resulteerd dit in Left Recursion waar we geen tijd meer voor hadden dit op te lossen. Een simpele `pop = expression.type != type.VOID` lost dit echter gauw genoeg op.

Als er labels in code regels voorkomen als L1, L2, etc, zullen deze in werkelijkheid uniek genummerd zijn.

5.2 Execute

5.2.1 ExpressionStatement

```
execute[S, pop] =
    evaluate[S]
    if pop:
        pop
```

`S` is hierbij een expressie en pop is wederom true iff `expression.type != type.VOID`.

5.2.2 while

```
execute[while E; do S; od] =
    L1:
        evaluate[E]
        ifeq  L2
        execute[S]
        goto L1
    L2:
```

De `ifeq` instructie kijkt of de waarde op de top van de stack gelijk is aan 0 (boolean waarden zijn integer waarden 0 of 1), en als dat het geval is jumpt de interpreter naar de label L2 (naar de eerstvolgende instructie na de while loop). Als dit niet het geval is gaat hij verder met de eerste instructie van `S` en hierna volgt een jump naar het begin om te kijken of een volgende iteratie nodig is.

5.3 Evaluate

5.3.1 Compound Expression

Omdat een compound expression ook weer een expressie is, maar bestaat uit expressie statements, moet de codegenerator de mogelijk gegenereerde pop van de laatste expressie statement verwijderen. De regel is als volgt:

```
compoundexpression = COMPOUND (declaration | expression_statement)*

evaluate[E, last_expr_is_void] =
    evaluate[E]
    if not <last_expr_is_void>:
         remove_last_instruction
```

Hier geeft de variabele `last_expr_is_void` aan of de laatste expressie (als er tenminste 1 expressie is) van type VOID is. Als dit niet het geval is, is er een pop gegenereerd door `expression_statement` die verwijderd moet worden. Dit gebeurt door een dummy instructie `remove_last_instruction` te genereren die de laaste instructie verwijderd. Dit gebeurd voordat mhet resultaat naar een Jasmin aseembly file geschreven wordt.

5.3.2 if then else expression

```
evaluate[if E1 then E2 (else E3)?] =
        evaluate[E1]
        ifeq L1
        evaluate[E2]
        goto L2
    L1:
        evaluate[E3]
    L2:
```

5.3.3 Identifier

```
evaluate[ID, kind, is_global, type] =
    if kind == CONST:
        ldc getvalue(<ID>)
    else if is_global:
        getstatic Main/<ID> <type>
    else:
        iload address_of(<ID>)
```

Als de variabele een constante is wordt de waarde bijgehouden in de symbol table. De functie `getvalue` haalt hier de waarde van de constante op. `address_of` is hier een functie die gegeven een identifier zijn address als locale variabele ophaalt. Als de waarde een globale variabele is, is het een field zodat functies beschikking hebben tot deze variabele (als het een locale variabele in de statische `main` functie zou zijn zou dit niet het geval zijn). In dit geval is de `is_global` boolean true en is `type` het type van de variabele.

5.3.4 Integer Literal

```
evaluate[literal, iconst, bipush, ldc] =
    if iconst:
        iconst_<literal>
    elif bipush:
        bipush <literal>
    else:
        ldc <literal>
```

Als de integer literal in de juiste range van iconst of bipush zit, worden deze geprefereerd over ldc voor compactere bytecode.

5.3.5 Character Literal

```
evaluate[literal] =
    bipush <literal>
```

5.3.6 Boolean Literals

```
evaluate[true] =
    iconst_1
```

en

```
evaluate[false] =
    iconst_0
```

5.3.7 Arithmetic, AND en OR

```
evaluate[E1, op, E2, instruction] =
    evaluate[E1]
    evaluate[E2]
    <instruction>; E1 <op> E2
```

Hierbij is op een binaire arithmetic operator zoals `+`, `-`, etc, of AND/OR. `instruction` is de bijbehorende JVM Jasmin instructie. De operators mappen als volgt naar hun instructies:

- `+` : `iadd`
- `-` : `isub`
- `*` : `imul`
- `/` : `idiv`
- `%` : `irem`
- `&&` : `iand`
- —— : `ior`

5.3.8 Relational operators

```
evaluate[E1 op E2, instruction] =
    evaluate[E1]
    evaluate[E2]
    <instruction> L1   ; E1 <op> E2
    iconst_0
    goto L2
L1:
    iconst_1
L2:
```

Voor elke relational relational operator zoals `<=`, `==` etc wordt deze code gegenereerd met bijbehorende instructie. De opererator naar instructie mapping is als volgt:

```
<  : ifcmp_lt
<= : ifcmp_le
== : ifcmp_eq
!= : ifcmp_ne
>= : ifcmp_ge
>  : ifcmp_gt
```

5.3.9 Unary Plus and Minus

```
evaluate[-E] =
    evaluate[E]
    ineg
```

Voor unary `+` hoeft er niets te gebeuren.

5.3.10 NOT

```
evaluate[!E] =
    evaluate[E]
    ifeq L1
    iconst_0
    goto L2
L1:
    iconst_1
L2:
```

5.3.11 Assignment

```
evaluate[ID := E, is_global, type] =
    evaluate[E]
    dup

    if is_global:
        putstatic Main/<ID> <type>
    else:
        istore address_of(<ID>)
```

`address_of` is hier een functie die gegeven een identifier zijn address als locale variabele ophaalt. De boolean `is_global` geeft aan of de variabele een globale variabele variabele is (in welk geval het een statisch veld is van de `Main` class). `type` geeft vervolgens aan van welk type dit veld (de variabele) is. De `dup` is nodig om de waarde eerst te dupliceren aangezien assignment een expressie is.

5.3.12 Print

```
evaluate[print(E+), type_denoters, bools, dup_top] =
    for expr, type_denoter, is_bool in E, type_denoters, bools:
        evaluate[expr]

        if <dup_top>:
            dup

        if <is_bool>:
```

```
            ifeq L1
            ldc "true"
            goto L2
        L1:
            ldc "false"
        L2:

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

Hier krijgt de vertaalregel voor elke expressie mee of het type `boolean` is (`bools`), en welke overloaded versie van `System.out.println` moet worden aangeroepen (`type_denoters`). In het geval van een boolean moet de waarde `"true"` of `"false"` worden geladen in plaats van de integer waarde. De boolean dup_top geeft aan of de print een statement of expressie is. Als het een expressie is (in het geval van een enkele print), moet de waarde worden gedupliceerd op de stack voordat de println de waarde popt.

5.3.13 Read

```
evaluate[read(ID+), bools, ints, dup_top] =
    for id, is_bool, is_int in ID, bools, ints:
    getstatic Main/scanner_field Ljava/util/Scanner;

        if <is_bool>:
            invokevirtual java/util/Scanner/nextBoolean()Z
        elif <is_int>:
            invokevirtual java/util/Scanner/nextInt()I
        else:
            invokevirtual java/util/Scanner/nextByte()I

        if <dup_top>:
            dup

        evaluate[ID := top of stack]
```

Hier krijgt de vertaalregel voor elke expressie mee of het type `boolean` of int is (`bools`, textttints) en of de read een expressie is (`dup_top`). Afhankelijk van het type (boolean, integer of character) wordt een aanroep gedaan naar de methoden nextBoolean(), nextInt() of nextByte() van Scanner.

5.3.14 Function Call

```
evaluate[FUNCCALL ID (expr)*, param_types, return_type] =
    for each expr:
        evaluate[expr]
```

```
invokestatic Main/<ID>(<param_type> for param_type in <param_types>)<return_type>
```

We evalueren eerst alle argumenten (`expr`), waarna we een statische aanroep doen naar de methode in `Main` met de naam van de functie aanroep. `param_types` zijn de typen van de parameters van de functie en `return_type` is het return type van de functie.

5.4 Elaborate

De elaborate code functie handeld declaraties af van constanten en variabelen en van functie definities. In elk geval worden er symbol table entries aangemaakt. In het geval van constanten wordt er geen code gegenereerd, maar wordt de constante waarde onthouden in de symbol table.

5.4.1 Variabelen en Constanten

```
elaborate[VarDeclaration type ID] =
    entry = enter <ID> of <type> and kind VAR in symbol table
    if entry.level == 0:
        ; global scope, declare field
        .field public static <ID> <type>
    else:
        increase amount of local variables by one

elaborate[ConstDeclaration type ID value] =
    enter <ID> of <type> and kind CONST in symbol table
    set value on symbol table entry of <ID>
```

5.4.2 Function definition

```
elaborate[FUNCTION ID (param_type param)* body (RETURN return_expr)? return_type] =
    enter <ID> of <type> and kind FUNC in symbol table

    .method public static <funcname>(<param_type for each param_type>)<return_type>
    .limit stack <stack_limit>
    .limit locals <locals_limit>
        execute[body]
        if <return_expr>:
            evaluate[return_expr]
            ireturn
        else:
            return
    .end method
```

Hier is `ID` de naam van de functie, `param_type` en `param` het type en naam van elke respectievelijke parameter, `body` de function body, de optionele `return_expr` de expressie waarvan de waarde returned wordt, en `return_type` het return type

van de functie. We genereren dus een statische methode, aangezien we de `Main` class nooit instantieren.
Aangezien methoden niet gegeneerd kunnen worden in een andere methode, herschrijft onze "post-processor" methoden in methoden naar een lijst van methoden. i.e.

```
.method A
    .method B
        ...
    .end method
.end method
```

wordt

```
.method A
    ...
.end method

.method B
    ...
.end method
```

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 SR_Type SR_type = null;
        public SR_Kind SR_kind = null;
        public SR_Func SR_func = null;

    /* Given a type as AST Tree node, return the SR_Type */
```

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

```
        switch (token.getType()) {
            return SR_Type.BOOL;
                        SELMATree t = (SELMATree)children.get(i);
```

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:

```
    public SymbolTable() {
        currentLevel = -1;
        entries = new HashMap<String, Stack<Entry>>();
    }
     */
    public void openScope()  {
        currentLevel++;
    }

    /**
     * Closes the current scope. All identifiers in
    }

    /** Returns the current scope level. */
    public int currentLevel() {
        return currentLevel;
    }

    /** Return whether the entry takes up space on the stack */
    private boolean isLocal(Entry entry) {
        }

        if (s.isEmpty() || s.peek().level != currentLevel) {
                entry.level = currentLevel;
                s.push(entry);
                if (isLocal(entry))
                nextAddr++;
```

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.

```
import org.antlr.runtime.tree.Tree;
```

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 class CompilerEntry extends CheckerEntry {
```

7 Testplan en -resultaten

Voor het testen hebben we testprogramma's geschreven in onze taal. Ook zit er een testrunner bij die automatisch alle tests in de 'test' subdirectory vind en compileerd en optioneel executeerd. Tests kunnen van de volgende typen zijn:

- Compile - Compileer de test
- Error - Compileer en (als successvol), executeer
- Run - Compileer en executeer

Bij deze tests kunnen in het programma tags gezet worden, namelijk ¡input¿text¡/input¿ voor input voor het programma op stdin, en ¡output¿text¡/output¿ voor output van het programma (of de compiler, in het geval van een compile of error test). Error tests beginnen met de prefix 'error_' in de bestandsnaam, en compile tests met 'compile_'. Zo kan getest worden voor juiste syntax en semantiek, juiste error reporting bij onjuiste syntax en semantiek, en correctie vertaalregels door middel van correcte executie, en runtime error checking voor juiste programmas met runtime fouten. Om de tests te runnen is Python 2.5+ ¡ 3.0 nodig. De tests kunnen als volgt worden geexecuteerd:

```
\$ python test.py
```

of

```
\$ make tests
```

Als een run test geen output heeft gespecificeerd is de exit status van het programma bepalend of de test faalt of niet. Bij een error test geldt het tegenovergestelde: zonder gespecificeerde output moet de exit status nonzero zijn.

De tests in de test directory testen alle constructen uit de taal, zoals arithmetic, alle operators, typen, constanten, scope rules, etcetera. Hieronder is output gegeven van de test runner. Als een test faalt zal de output worden weergegeven:

```
[0] [11:11] ~/selma git(master!)      python test.py
Run         test/correct.selma
    ...  OK
Error       test/error_context.selma
    ...  OK
Error       test/error_if.selma
    ...  OK
Error       test/error_runtime_uninitialized.selma
    ...  OK
Error       test/error_runtime_zerodivision.selma
    ...  OK
Error       test/error_syntax.selma
    ...  OK
Error       test/error_while.selma
    ...  OK
```

```
Error       test/error_while_void.selma
    ...  OK
Run         test/sample.selma
    ...  FAIL (exit status 0)
  Got:
        h
    >>> a
        l
        l
        o
  Expected:
        h
    >>> e
        l
        l
        o
```

```
Run         test/test_if.selma
    ...  OK
Run         test/test_operators.selma
    ...  OK
Run         test/test_while.selma
    ...  OK
Run         test/test_functions/correct_functions.SELMA
    ...  FAIL (exit status 1)
        ERROR: recognition exception thrown by compiler: null
        org.antlr.runtime.EarlyExitException
        at SELMA.SELMACompiler.compoundexpression(
            SELMACompiler.java:271)
        at SELMA.SELMACompiler.program(SELMACompiler.java:170)
        at SELMA.SELMA.main(SELMA.java:99)
```

```
Error       test/test_functions/error_doublefunction.selma
    ...  OK
Error       test/test_functions/error_wrongparamcount.selma
    ...  OK
Error       test/test_functions/error_wrongparamtype.selma
    ...  OK
Error       test/test_functions/error_wrongreturntype.selma
    ...  OK
Ran 17 test(s), SUCCESS=15, FAILURE=2
```

Hier zien we dat `test/sample.selma` niet de correcte output heeft, maar wel executeerde zonder fouten (exit status 0), terwijl `test/test_functions/correct_functions.SELMA` een compilatie fout had met een exit status 1. De eerste colom geeft het type test aan, in dit geval 'Error' of 'Run'. Ter demonstratie is `test/sample.selma` bijgevoegd:

```
<output>
    h
    e
    l
    l
    o
</output>
print('h', 'a', 'l', 'l', 'o');
```

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 100 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! (funcpars SEMICOLON!)* RPAREN!
            funcbody
        ;
funcpars : identifier (COMMA identifier)* COLON type -> (identifier type
    )+;
type
        : INT
        | BOOL
```

```
        | CHAR
        ;

funcbody
        : COLON type LCURLY compoundexpression FUNCRETURN expression
            SEMICOLON RCURLY -> ^(FUNCRETURN type compoundexpression
            expression)
        | LCURLY! compoundexpression RCURLY!
        ;









//expression statement at line 146

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 expr_al7
                        -> ^(UPLUS expr_al7)
                | MINUS expr_al7
                        -> ^(UMIN expr_al7)
                | NOT 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
        ;

expr_funccall
        : FUNCTION^ identifier LPAREN! (expression COMMA!)* RPAREN!
        ;

expr_closedcompound
        : LCURLY^ compoundexpression RCURLY
        ;

expr_closed
        : LPAREN! expression RPAREN!
        ;









//unsigned at line 244

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;
        import SELMA.SELMATree.SR_Func;
}

// Alter code generation so catch-clauses get replaced with this action.
@rulecatch {
        catch (RecognitionException re) {
                /*
                if (node != null)
                    System.err.println(
                        String.format("Error on line \%d:\%d: \%s", node
                            .getLine(),
                                                                    node
                                                                        .
                                                                        getCharPositionInLine
                                                                        ()
                                                                        ,

                                                                    re.
                                                                        getMessage
                                                                        ()
                                                                        )
                                                                        )
                                                                        ;

                */
                throw re;
        }
}

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

        public void matchType(Tree expectedType, SR_Type exprType) {
            matchType(((SELMATree) expectedType).getSelmaType(),
                exprType);
        }

        public void matchType(Tree expectedType, Tree exprType) {
            matchType(((SELMATree) expectedType).getSelmaType(),
                    ((SELMATree) exprType).getSelmaType());
        }

        public void matchType(SR_Type expectedType, SR_Type exprType) {
            if (expectedType != exprType)
                throw new SELMAException(String.format(
                        "Expected type \%s, got type \%s",
                        expectedType,
                        exprType));
        }
}

program
    : ^(node=BEGIN
```

```
        {st.openScope();}
        compoundexpression
        {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);
            // System.err.println("..." + e1 + " " + e1.getLine());
            $node.SR_type = e1.SR_type;
            $node.SR_kind = e1.SR_kind;
        }
        ;

declaration
        : ^(node=VAR type id=ID)
   {
       st.enter($id, new CheckerEntry(((SELMATree) node.getChild(0)).
           getSelmaType(),
                                       SR_Kind.VAR));
   }
        | ^(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($id,new CheckerEntry(SR_Type.CHAR,SR_Kind.CONST));
             break;
         }
    }
        | ^(FUNCDEF funcname=ID
   {
       //enter as void
       st.enter($funcname, new CheckerEntry(SR_Type.VOID, SR_Kind.VAR,
           SR_Func.YES));
       st.openScope();
```

```
    }
        (param=ID typ1=(INT|BOOL|CHAR)
    {
         st.addParamToFunc($funcname, param, $typ1);
    }
                 )*
                 (
                         ^(node=FUNCRETURN type compoundexpression
                             expression
    {
         SELMATree type = (SELMATree) node.getChild(0);
         SELMATree expr = (SELMATree) node.getChild(2);
         st.retrieve($funcname).type = expr.SR_type;

         matchType(type, expr.SR_type);
    })
         | (compoundexpression))
    {
         //scope of function
         st.closeScope();
    });

type
   : node=INT
   | node=BOOL
   | node=CHAR
   ;

val
   : node=NUMBER
   | node=CHARV
   | node=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,
         String.format("Wrong types must be int (found \%s and \%s)", e1.
             SR_type, e2.SR_type));
    }

    $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)
     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;
    else
     $node.SR_kind=SR_Kind.VAR;
```

```
180     }

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

        if (e1.SR_type!=SR_Type.BOOL || e2.SR_type!=SR_Type.BOOL)
         throw new SELMAException($node,"Wrong type must be bool");
        $node.SR_type=SR_Type.BOOL;
190
        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;
195     }

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

        if (e1.SR_type!=e2.SR_type||e1.SR_type==SR_Type.VOID)
         throw new SELMAException($node,"Types must match and can't be void")
             ;
        $node.SR_type=SR_Type.BOOL;
205
        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;
210     }

        | ^(node=(UPLUS|UMIN) expression)
        {
        SELMATree e1 = (SELMATree)node.getChild(0);
215
        if (e1.SR_type!=SR_Type.INT)
         throw new SELMAException($node,"Wrong type must be int");
        $node.SR_type=SR_Type.INT;

220     $node.SR_kind=e1.SR_kind;
        }

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

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

230         node.SR_type = SR_Type.BOOL;
            node.SR_kind = e1.SR_kind;
        }

        | ^(node=IF {st.openScope();} compoundexpression
235           THEN {st.openScope();} compoundexpression {st.closeScope()
                  ;}
              (ELSE {st.openScope();} compoundexpression {st.closeScope()
                  ;})?
           {st.closeScope();})
        {
        SELMATree e1 = (SELMATree)node.getChild(0);
240     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");
```

```
245
     if (e3==null) { //no else
       $node.SR_type=SR_Type.VOID;
       $node.SR_kind=null;
     } else { // there is a else
250    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
255          $node.SR_kind=SR_Kind.VAR;
       } else {
         $node.SR_type=SR_Type.VOID;
         $node.SR_kind=null;
       }
260  }
     }

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

270 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;
275
    }

         | ^(node=READ (id=ID
         {
280          if (st.retrieve($id).kind!=SR_Kind.VAR)
                 throw new SELMAException($id,"Must be a variable");
         })+
         {
              if ($node.getChildCount() == 1) {
285               $node.SR_type = st.retrieve(node.getChild(0)).type;
                  $node.SR_kind = SR_Kind.VAR;
              } else {
                  $node.SR_type = SR_Type.VOID;
                  $node.SR_kind = null;
290           }
         }
            )

         | ^(node=PRINT expression+)
295       {
     for (int i=0; i<((SELMATree)node).getChildCount(); i++){
       if (((SELMATree)node.getChild(i)).SR_type == SR_Type.VOID)
          throw new SELMAException($node, "Can not be of type void");
     }
300     if ($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;
305       $node.SR_kind = null;
        }
       }
     -> ^(PRINT expression)+

310     | ^(node=FUNCTION ID expression*)
{
```

```
//retrieve function (if existent)
SELMATree func = (SELMATree)$node;
CheckerEntry entry = st.retrieve($ID);
$node.SR_type=entry.type;
$node.SR_kind=entry.kind;

//matchparamlists
//same length?
int argc = func.getChildCount()-1;
if (entry.params.size() != argc)
        throw new SELMAException(node, String.format(
                "\%s takes \%d arguments (\%d given)", $ID.text, entry.
                    params.size(), argc));
//every entry matches?
for (int i=1; i<func.getChildCount(); i++){
    SELMATree expr = (SELMATree)func.getChild(i);
    if (expr.SR_type != entry.params.get(i-1).type)
        throw new SELMAException(expr,"Param is not of the right type");
}
}



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

   CheckerEntry ident = st.retrieve(e1);

   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);

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

        | ^(node=LCURLY {st.openScope();} compoundexpression {st.
            closeScope();} RCURLY)
        {
            SELMATree e1 = (SELMATree) node.getChild(0);
            $node.SR_type = e1.SR_type;
            $node.SR_kind = e1.SR_kind;
        }

        | node=NUMBER
         {
         $node.SR_type=SR_Type.INT;
         $node.SR_kind=SR_Kind.CONST;
         }

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

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

        | node=ID
```

```
            {
            CheckerEntry entry = st.retrieve($node);
            $node.SR_type=entry.type;
380         $node.SR_kind=entry.kind;
            }
           ;
```

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;
  import SELMA.SELMATree.SR_Func;

  import java.lang.StringBuilder;
}

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

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

    int curStackDepth;
    int maxStackDepth;
    int labelNum = 0;

    class StackDepthLabelCounter {
        public int curStackDepth;
        public int maxStackDepth;
        public int labelNum;
        public int nextAddr;
        public int localCount;
    }

    Stack<StackDepthLabelCounter> stack = new Stack<
        StackDepthLabelCounter>();

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

    private void enterFuncScope() {
        StackDepthLabelCounter o = new StackDepthLabelCounter();
        o.curStackDepth = curStackDepth;
        o.maxStackDepth = maxStackDepth;
        o.labelNum = labelNum;
        o.nextAddr = st.nextAddr;
        o.localCount = st.localCount;

        stack.push(o);

        st.openScope();
        curStackDepth = maxStackDepth = labelNum = st.localCount = 0;
        st.nextAddr = 0;
    }

    private void leaveFuncScope() {
        StackDepthLabelCounter o = stack.pop();
```

```
        st.closeScope();
        curStackDepth = o.curStackDepth;
        maxStackDepth = o.maxStackDepth;
        labelNum = o.labelNum;
        st.nextAddr = o.nextAddr;
        st.localCount = o.localCount;
    }

    private String getTypeDenoter(SR_Type type) {
        return st.getTypeDenoter(type, false);
    }

    private String getTypeDenoter(SR_Type type, boolean printing) {
        return st.getTypeDenoter(type, printing);
    }
}

program
  : ^(node=BEGIN {st.openScope();} compoundexpression END)
  { SELMATree expr = (SELMATree) $node.getChild(0);
    int localsCount = st.getLocalsCount();
    List<String> globalVariableFields = st.getAllLocalVariablesWithTypes
        ();
    st.closeScope();
  }
  -> program(instructions={$compoundexpression.st},
             source_file={SELMA.inputFilename},
             stack_limit={maxStackDepth + 3}, // +3 for print
             locals_limit={localsCount + 1}, // +1 for the String[] argv
                 parameter
             fields={globalVariableFields},
             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, 0).setVal(
      $val.text)); }
  //-> declareConst(id={$id.text}, val={$val.text}, type={"integer"},
      addr={st.nextAddr()-1})

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

```
    | ^(node=CONST CHAR val=CHARV (id=ID)+)
125 { char c = $val.text.charAt(1);
      st.enter($id, new CompilerEntry(SR_Type.CHAR, SR_Kind.CONST, 0).
          setChar(c));
    }
    //-> declareConst(id={$id.text}, val={(int) c}, type={"character"},
        addr={st.nextAddr()-1})

130 | ^(node=FUNCDEF funcname=ID
    {
          CompilerEntry funcentry = new CompilerEntry(
                          SR_Type.VOID, SR_Kind.VAR, 0, SR_Func.YES);
          st.enter($funcname, funcentry);
135       enterFuncScope();
          int paramCount = 0;
          StringBuilder signatureBuilder = new StringBuilder("(");
          //List<String> paramTypeDenoters = new ArrayList<String>();
    } (param=ID typ1=(INT|BOOL|CHAR)
140 {
          SELMATree type1 = (SELMATree) $node.getChild(++paramCount * 2);
          signatureBuilder.append(getTypeDenoter(type1.getSelmaType()));
          //paramTypeDenoters.add(getTypeDenoter(type1.getSelmaType()));
          st.addParamToFunc($funcname, param, type1);
145 })*
    ( ^(return_node=FUNCRETURN (INT|BOOL|CHAR) (body+=compoundexpression)
        retexpr=expression)
    | (body+=compoundexpression)
    )
    {
150       SELMATree funcbody;
          int stackLimit = maxStackDepth + 3;
          int localsLimit = st.getLocalsCount();

          signatureBuilder.append(")");
155
          if ($return_node == null) {
              funcbody = (SELMATree) $node.getChild(paramCount * 2 + 1);
              signatureBuilder.append("V");
          } else {
160           funcbody = (SELMATree) $return_node.getChild(1);
              SELMATree returnType = (SELMATree) $return_node.getChild(0);
              signatureBuilder.append(getTypeDenoter(returnType.
                  getSelmaType()));
          }
          leaveFuncScope();
165
          String signature = signatureBuilder.toString();
          funcentry.signature = signature;
    })
    -> function(funcname={$funcname.text},
170             body={$body},
                signature={signature},
                return_expression={$retexpr.st},
                is_void={signature.endsWith("V")},
                pop={funcbody.SR_type != SR_Type.VOID},
175             stack_limit={stackLimit},
                locals_limit={localsLimit + 1},
                line={$node.getLine()})
    ;

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

185 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={"irem"}, 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={"ior"}, line={node.getLine()
       }, op={"or"})

  | ^(node=AND e1=expression e2=expression) { curStackDepth--; }
  -> biExpr(e1={$e1.st},e2={$e2.st},instr={"iand"}, 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;}
```

```
235
   | ^(node=UMIN e1=expression)
   -> uExpr(e1={$e1.st}, instr={"ineg"}, line={node.getLine()}, op={"-"})

   | ^(node=NOT e1=expression)
240 -> not(e1={$e1.st}, line={node.getLine()},
         label_num1={labelNum++}, label_num2={labelNum++})

 //CONDITIONAL
   | ^(node=IF { st.openScope(); } ec1=compoundexpression { st.closeScope
       (); } THEN
245            { st.openScope(); } ec2=compoundexpression { st.closeScope
                   (); }
         (ELSE { st.openScope(); } ec3=compoundexpression { st.closeScope
             (); })?)
         { boolean ec3NotEmpty = $ec3.st != null;
           SELMATree expr2 = (SELMATree) node.getChild(2);
           SELMATree expr3 = null;
           if (ec3NotEmpty)
250            expr3 = (SELMATree) node.getChild(4);
         }
   -> if(ec1={$ec1.st},ec2={$ec2.st},ec3={$ec3.st}, label_num1={labelNum
       ++},
         label_num2={ec3NotEmpty ? labelNum++ : 0}, ec3_not_empty={
             ec3NotEmpty},
255      pop1={$node.SR_type == SR_Type.VOID && expr2.SR_type != SR_Type.
             VOID},
         pop2={ec3NotEmpty && $node.SR_type == SR_Type.VOID && expr3.
             SR_type != SR_Type.VOID})

   | ^(node=WHILE
       { st.openScope(); } ec1=compoundexpression { st.closeScope(); } DO
260    { st.openScope(); } ec2=compoundexpression { st.closeScope(); } OD
           )
   { SELMATree expr2 = (SELMATree) node.getChild(2);
     boolean pop = expr2.SR_type != SR_Type.VOID;
     if (pop)
         curStackDepth--;
265 }
   -> while(ec1={$ec1.st}, ec2={$ec2.st}, pop={pop},
          label_num1={labelNum++}, label_num2={labelNum++})

 //IO
270
   | ^(node=READ ID+)
       /*
       {
           CompilerEntry entry = st.retrieve($id);
275    }))
       -> readSingle(id={$id.text}, addr={entry.addr},
                     is_bool={entry.type == SR_Type.BOOL},
                     is_int={entry.type == SR_Type.INT},
                     dup_top={$node.SR_type != SR_Type.VOID},
280                  is_global={entry.level == 0},
                     type_denoter={getTypeDenoter(entry.type)})
       */
         { boolean isExpr = $node.SR_type != SR_Type.VOID;
           List<Integer> addrs = new ArrayList<Integer>();
285        List<Boolean> isBool = new ArrayList<Boolean>();
           List<Boolean> isInt = new ArrayList<Boolean>();
           List<Boolean> globals = new ArrayList<Boolean>();
           List<String> ids = new ArrayList<String>();

290        for (int i = 0; i < $node.getChildCount(); i++) {
               SELMATree child = (SELMATree) $node.getChild(i);
               CompilerEntry entry = st.retrieve(child);
               addrs.add(entry.addr);
               isBool.add(child.SR_type == SR_Type.BOOL);
```

```
295                 isInt.add(child.SR_type == SR_Type.INT);
                    globals.add(entry.level == 0);
                    ids.add(child.getText());
                }
            }
300         -> read(ids={ids}, addrs={addrs}, dup_top={isExpr},
                    is_bool={isBool}, is_int={isInt},
                    globals={globals}, line={node.getLine()})

    | ^(node=PRINT (exprs+=expression)+)
305 {
        boolean isExpr = $node.SR_type != SR_Type.VOID;
        int childCount = ((SELMATree) node).getChildCount();
        List<Integer> labelNums1 = new ArrayList<Integer>();
        List<Integer> labelNums2 = new ArrayList<Integer>();
310     List<String> typeDenoters = new ArrayList<String>();
        List<Boolean> exprIsBool = new ArrayList<Boolean>();

        if (!isExpr)
            curStackDepth -= childCount;
315
        for (int i = 0; i < childCount; i++) {
            SELMATree child = (SELMATree) $node.getChild(i);
            boolean isBool = child.SR_type == SR_Type.BOOL;
            if (isBool) {
320             labelNums1.add(labelNum++);
                labelNums2.add(labelNum++);
            } else {
                labelNums1.add(0);
                labelNums2.add(0);
325         }
            typeDenoters.add(getTypeDenoter(child.SR_type, true));
            exprIsBool.add(isBool);
        }
    }
330 -> print(exprs={$exprs}, type_denoters={typeDenoters}, dup_top={
        isExpr},
             expr_is_bool={exprIsBool},
             label_nums1={labelNums1}, label_nums2={labelNums2}, line={
                 $node.getLine()})

  | ^(node=FUNCTION id=ID (exprs+=expression)*)
335     -> funccall(id={$id.text}, signature={st.retrieve($id).signature
            }, exprs={$exprs})
//ASSIGN
  | ^(BECOMES node=ID e1=expression)
  {
      CompilerEntry entry = st.retrieve(node);
340   boolean isConst = node.SR_kind == SR_Kind.CONST;
      boolean isGlobal = false;
      String typeDenoter = getTypeDenoter(entry.type);

      if (entry.level == 0) {
345       isGlobal = true;
      }
  }
      -> assign(id={$node.text},
                type={$node.type},
350             addr={st.retrieve($node).addr},
                e1={$e1.st},
                is_global={isGlobal},
                type_denoter={typeDenoter})

355 //closedcompound
  | ^(node=LCURLY {st.openScope();} cmp=compoundexpression {st.
      closeScope();} RCURLY)
        -> compound(instructions={$cmp.st}, line={$node.getLine()}, pop
            ={false})
```

```
//VALUES
  | node=NUMBER { incrStackDepth();
                  int num = Integer.parseInt($node.text); }
    -> loadNum(val={$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})

  | 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()
        })

  | node=ID
    {
        incrStackDepth();
        CompilerEntry entry = st.retrieve(node);
        boolean isConst = node.SR_kind == SR_Kind.CONST;
        boolean isGlobal = false;
        String typeDenoter = getTypeDenoter(entry.type);

        if (entry.level == 0) {
            isGlobal = true;
        }
    }
    -> loadVal(id={$node.text}, addr={entry.addr}, val={entry.val},
        is_const={isConst},
               is_global={isGlobal}, type_denoter={typeDenoter})
  ;
```

9.4 ANTLR Codegenerator Stringtemplate specificatie

```
//SELMA string template

group SELMA;

program(instructions, fields, source_file, stack_limit, locals_limit, pop) ::= <<
.source <source_file>
.class public Main
.super java/lang/Object
.field public static scanner_field Ljava/util/Scanner;

<fields : { f | .field public static <f> }; separator="\n">

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

.method public static main([Ljava/lang/String;)V
.limit stack <stack_limit>
.limit locals <locals_limit>
    new java/util/Scanner
    dup                                     ; dup for \<init\>
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/util/Scanner/\<init\>(Ljava/io/InputStream;)V
    putstatic Main/scanner_field Ljava/util/Scanner;

<instructions>

<if (pop)>
    pop

<endif>

    return
.end method
>>

compound(instructions, line, pop) ::= <<
.line <line>
```

```
<instructions; separator="\n">
<if (pop)>
    removeLastInstruction  ; line <line>
<endif>
>>

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

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

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

not(e1, label_num1, label_num2, line) ::= <<
.line <line>
    <e1>
        ifeq L<label_num1>
        iconst_0
        goto L<label_num2>
L<label_num1>:
        iconst_1
L<label_num2>:
>>

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) ::= <<
    ldc <val>                     ; const <id> : <type>
    istore <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, val, is_const, is_global, type_denoter) ::= <<
<if (is_const)>
    ldc <val>                   ; load constant <id>

<elseif (is_global)>
    getstatic Main/<id> <type_denoter> ; load global <id>
```

```
125 |
    |<else>
    |    iload <addr>                ; load <id> from <addr>
    |
    |<endif>
130 |>>
    |
    |loadChar(val, char, line) ::= <<
    |.line <line>
    |    bipush <val>                ; ldc <char>
135 |>>
    |
    |//Assign
    |assign(id, type, addr, e1, is_global, type_denoter) ::= <<
    |<e1>                            ; e1 right hand for assignment
140 |    dup
    |<if (is_global)>
    |    putstatic Main/<id> <type_denoter>
    |
    |<else>
145 |    istore <addr>                ; store e1 in <id>
    |
    |<endif>
    |
    |>>
150 |
    |read(ids, addrs, dup_top, is_bool, is_int, globals, line) ::= <<
    |.line <line>
    |<ids, addrs, is_bool, is_int, globals :
    |    { id, a, b, i, g | <readSingle(id=id, addr=a,
155 |                                    is_bool=b, is_int=i,
    |                                    dup_top=dup_top, is_global=g
    |                                    ) > }; separator="\n">
    |
    |>>
160 |
    |readSingle(id, addr, is_bool, is_int, dup_top, is_global) ::= <<
    |    getstatic Main/scanner_field Ljava/util/Scanner;
    |<if (is_bool)>
    |    invokevirtual java/util/Scanner/nextBoolean()Z
165 |
    |<elseif (is_int)>
```

```
    invokevirtual java/util/Scanner/nextInt()I

<else>
    invokevirtual java/util/Scanner/nextByte()I

<endif>

<if (dup_top)>
    dup

<endif>

<if (is_global)>
    <if (is_bool)>
        putstatic Main/<id> I
    <elseif (is_int)>
        putstatic Main/<id> I
    <else>
        putstatic Main/<id> C
    <endif>
<else>
    istore <addr>

<endif>
>>

print(exprs, type_denoters, dup_top, expr_is_bool, label_nums1, label_nums2, line) ::= <<
.line <line>
<exprs, type_denoters, expr_is_bool,
 label_nums1, label_nums2 : { e, t, b, L1, L2 | < printSingle(expr=e,
                                                  type_denoter=t,
                                                  dup_top=dup_top,
                                                  is_bool=b,
                                                  label_num1=L1,
                                                  label_num2=L2) > }>
>>


printSingle(expr, type_denoter, dup_top, is_bool, label_num1, label_num2) ::= <<
<expr>
<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/println(<type_denoter>)V

>>

//conditionals
if (ec1, ec2, ec3, label_num1, label_num2, ec3_not_empty, pop1, pop2) ::= <<
<ec1>                                                   ; e1 condition
    ifeq L<label_num1>          ; e1 is false
<ec2>                           ; e2 if true expression
<if (pop1)>
    pop

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

L<label_num1>:
<if (ec3_not_empty)>
<ec3>                                                   ; e3 if false expression
<if (pop2)>
    pop                         ; pop

<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>:
>>

function(funcname, signature, body, return_expression,
         pop, locals_limit, stack_limit, line) ::= <<
\<method\>
.method public static <funcname><signature>
.limit stack <stack_limit>
.limit locals <locals_limit>
.line <line>

<body; separator="\n\n">

<if (pop)>
    pop

<endif>
<return_expression>
<if (is_void)>
    return

<else>
    ireturn

<endif>

.end method
\</method\>
>>

funccall(id, signature, exprs) ::= <<
<exprs; separator="\n">
    invokestatic Main/<id><signature>
```

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.

9.5.1 SELMA-code van pasen

```
/*
Methode van Gau

De Duitse geleerde Carl Friedrich Gau  publiceerde in 1800 een
    wiskundig algoritme waarmee de paasdatum voor een willekeurig jaar
    berekend kan worden. Gau  maakte toch een fout: hij hield niet
    goed rekening met de maancorrectie, zodat bijvoorbeeld zijn
    paasdatum voor 4200 uitkomt op 13 april in plaats van 20 april. De
    methode van Gau  loopt als volgt:
*/

const maxyear: integer = 2099;

function checkjaar(jaar: integer;): boolean {
        var ok: boolean;
        if jaar < 0; then
                print('j','e','z','u','s','_','m','o','e','t','_','z','i
                    ','j','n','_','g','e','b','o','r','e','n');
        else
                if jaar > maxyear; then
                        print('h','e','b','t','_','g','e','d','u','l','d
                            ');
                fi;
        fi;
        ok := (jaar>=0)&& jaar<=maxyear;
        return ok;
};

print('H','a','l','l','o','_','v','a','n','_','w','e','l','k','_','j','a
    ','a','r','_','w','i','l','_','j','e','_','d','e','_','p','a','a','
    s','d','a','t','u','m','_','w','e','t','e','n');

var jaar: integer;
read(jaar);

const negentien: integer = 19;

if @checkjaar(jaar,); then
//    Bepaal het gulden getal:
//    Deel het jaartal door 19, neem de rest en tel er 1 bij op (zoals
    Dionysius). Noem dit getal G. Voor het jaar 1991 geldt G = 16.
        var G: integer;
        G := jaar/negentien;
        G := G+1;

//    Bepaal het eeuwtal:
//    Geheeldeel het jaartal door 100 en tel daar 1 bij op. Noem dit
    getal C. Voor het jaar 1991 geldt C = 20.
        var C: integer;
        C := jaar;
        C := C/100 + 1;

//    Corrigeer vervolgens voor jaren die geen schrikkeljaar zijn:
//    Vermenigvuldig C met 3, geheeldeel het resultaat door 4 en trek er
     12 van af. Noem dit getal X. Voor de twintigste en eenentwintigste
     eeuw geldt X = 3.
        var X: integer;
        const stap: integer = 1;
        const twaalf: integer = 12;
        function nest(stap,w: integer;): integer {
                var returnvalue: integer;
                if stap==1; then
                        returnvalue := 3*w;
                else
                        if stap == 2; then
                                returnvalue := w/4;
```

```
                                        var loop: integer;
                                        loop := 1;
                                        while loop <= twaalf; do
                                                returnvalue := returnvalue-1;
                                                loop := loop + 1;
                                        od;
                                fi;
                        fi;
                        return returnvalue;
                };
                X := @nest(stap+1,@nest(stap,C,),);

//      Maancorrectie:
//      Neem 8 maal C, tel er 5 bij op, deel het geheel door 25 en trek er 5 vanaf. Noem dit getal Y. Voor de twintigste en eenentwintigste eeuw geldt: Y = 1.
                var Y: integer;
                if jaar >= 1900; then
                        Y := 1;
                else
                        Y := (8*C)/25 - 5;
                fi;

//      Zoek de zondag:
//      Vermenigvuldig het jaartal met 5, geheeldeel de uitkomst door 4, trek er X en 10 vanaf, en noem dit getal Z. Voor 1991 geldt: Z = 2475.
                function nested(): integer {
                        var Z: integer;
                        Z := (((jaar*5)/4)-X)+-10;
                        function copy(jaar: integer;): integer {
                                42;
                                return jaar;
                        };
                        Z := @copy(Z,);
                        return Z;
                };
                var Z: integer;
                Z := @nested();

//      Bepaal de epacta:
//      11 maal G + 20 + Y. Trek daarvan X af, geheeldeel het resultaat door 30 en noem de rest E. Als E gelijk is aan 24, of als E gelijk is aan 25 en het gulden getal is groter dan 11, tel dan 1 bij E op. De Epacta voor 1991 is 14.
                var E,EE: integer;
                E := EE := ({const elf: integer = 11; elf*(G+20+Y);}-X)/30;
                const mnope: boolean = false;
                if EE==24 || (E==25&&G>11) || mnope; then
                        E := 1+E;
                fi;

//      Bepaal de volle maan:
//      Trek E af van 44. Noem dit getal N. Als N kleiner is dan 21, tel er dan 30 bij op. Voor 1991 geldt: N = 30
                var N: integer;
                const minus2: integer = 44;
                N := minus2-E;
                var tosmall: boolean;
                tosmall := N<21;
                N := N+
                        if tosmall; then
                                30;
                        else
                                0;
                        fi;

//      Nu door naar zondag:
```

```
//    Tel Z en N op. Geheeldeel het resultaat door 7 en trek de rest af
    van N+7. Noem dit getal P. Voor 1991 geldt: P = 31.
        var P: integer;
        P := (N+7)-(Z+N)%7;

//    Paasdatum: Als P groter is dan 31, trek er dan 31 vanaf, en de
    paasdatum valt in April. Anders is de paasdag P in Maart. Zo wordt
    voor 1991 gevonden 31 maart.
        var month: integer;
        var day: integer;
        if P>31; then
                month := 4;
                day := P%31;
        else
                month := 3;
                day := P;
        fi;

        function printdatum() {
                print(day);
                var under: character;
                under := print('-');
                if month==3; then
                        print('M','a','a','r','t');
                else
                        print('A','p','r','i','l');
                fi;
                print(under,jaar);
        };
        @printdatum();
fi;
```

9.5.2 Jasmin-code van pasen

```
.source pasen/pasen.SELMA
.class public Main
.super java/lang/Object
.field public static scanner_field Ljava/util/Scanner;
.field public static jaar I
.method public <init>()V
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method
.method public static main([Ljava/lang/String;)V
.limit stack 3
.limit locals 19
    new java/util/Scanner
    dup                                             ; dup for <init>
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
    putstatic Main/scanner_field Ljava/util/Scanner;
.line 7
.line 22
    bipush 72                   ; ldc 'H'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 22
    bipush 97                   ; ldc 'a'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
```

```
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 108                  ; ldc 'l'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 108                  ; ldc 'l'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 111                  ; ldc 'o'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 95                 ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 118                  ; ldc 'v'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 97                 ; ldc 'a'
```

```
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 110                  ; ldc 'n'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 95                   ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 119                  ; ldc 'w'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 108                  ; ldc 'l'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
```

```
85  .line 22
      bipush 107                ; ldc 'k'
      getstatic java/lang/System/out Ljava/io/PrintStream;
      swap
      invokevirtual java/io/PrintStream/println(C)V
90  .line 22
      bipush 95                 ; ldc '_'
      getstatic java/lang/System/out Ljava/io/PrintStream;
      swap
      invokevirtual java/io/PrintStream/println(C)V
95  .line 22
      bipush 106                ; ldc 'j'
      getstatic java/lang/System/out Ljava/io/PrintStream;
      swap
      invokevirtual java/io/PrintStream/println(C)V
100 .line 22
      bipush 97                 ; ldc 'a'
      getstatic java/lang/System/out Ljava/io/PrintStream;
      swap
      invokevirtual java/io/PrintStream/println(C)V
105 .line 22
      bipush 97                 ; ldc 'a'
      getstatic java/lang/System/out Ljava/io/PrintStream;
      swap
      invokevirtual java/io/PrintStream/println(C)V
110 .line 22
      bipush 114                ; ldc 'r'
      getstatic java/lang/System/out Ljava/io/PrintStream;
```

```
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 95                ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 119               ; ldc 'w'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 105               ; ldc 'i'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 108               ; ldc 'l'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 95                ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
```

```
    bipush 106                  ; ldc 'j'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 95                   ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 100                  ; ldc 'd'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 95                   ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
```

```
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 112                  ; ldc 'p'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 97                   ; ldc 'a'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 97                   ; ldc 'a'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 115                  ; ldc 's'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 100                  ; ldc 'd'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 97                   ; ldc 'a'
```

```
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 116                  ; ldc 't'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 117                  ; ldc 'u'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 109                  ; ldc 'm'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 95                  ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 119                  ; ldc 'w'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
```

```
.line 22
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 116                  ; ldc 't'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 22
    bipush 110                  ; ldc 'n'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 25
    getstatic Main/scanner_field Ljava/util/Scanner;
    invokevirtual java/util/Scanner/nextInt()I
    dup
        putstatic Main/jaar I
    pop
.line 29
    getstatic Main/jaar I ; load global jaar
```

```
    invokestatic Main/checkjaar(I)I
    ifeq L19          ; e1 is false
.line 32
.line 33
    getstatic Main/jaar I ; load global jaar
    ldc 19                 ; load constant negentien
    idiv                          ; e1 right hand for assignment
    dup
    istore 3              ; store e1 in G
    pop
.line 34
    iload 3               ; load G from 3
    iconst_1
    iadd                          ; e1 right hand for assignment
    dup
    istore 3              ; store e1 in G
    pop
.line 39
    getstatic Main/jaar I ; load global jaar
                           ; e1 right hand for assignment
    dup
    istore 4              ; store e1 in C
    pop
.line 40
    iload 4               ; load C from 4
    bipush 100
    idiv
    iconst_1
```

```
    iadd                            ; e1 right hand for assignment
    dup
    istore 4                ; store e1 in C
    pop
.line 64
    ldc 1                    ; load constant stap
    iconst_1
    iadd
    ldc 1                    ; load constant stap
    iload 4                 ; load C from 4
    invokestatic Main/nest(II)I
    invokestatic Main/nest(II)I                             ; e1 right hand for assignment
    dup
    istore 5                ; store e1 in X
    pop
.line 69
    getstatic Main/jaar I ; load global jaar
    ldc 1900
    if_icmpge L0        ; e1 >= e2
    iconst_0
    goto L1
L0:
    iconst_1
L1:
    ifeq L2          ; e1 is false
.line 70
    iconst_1
                             ; e1 right hand for assignment
```

```
    dup
    istore 7                ; store e1 in Y
    goto L3
L2:
.line 72
    bipush 8
    iload 4                 ; load C from 4
    imul
    bipush 25
    idiv
    iconst_5
    isub                            ; e1 right hand for assignment
    dup
    istore 7                ; store e1 in Y
L3:
    pop
.line 88
    invokestatic Main/nested()I                             ; e1 right hand for assignment
    dup
    istore 9                ; store e1 in Z
    pop
.line 93
    ldc 11                    ; load constant elf
.line 93
    iload 3                 ; load G from 3
    bipush 20
    iadd
    iload 7                 ; load Y from 7
```

```
    iadd
    imul
    iload 5                 ; load X from 5
    isub
    bipush 30
    idiv                            ; e1 right hand for assignment
    dup
    istore 11               ; store e1 in EE
                            ; e1 right hand for assignment
    dup
    istore 10               ; store e1 in E
    pop
.line 95
    iload 11                ; load EE from 11
    bipush 24
    if_icmpeq L4        ; e1 = e2
    iconst_0
    goto L5
L4:
    iconst_1
L5:
.line 95
    iload 10                ; load E from 10
    bipush 25
    if_icmpeq L6        ; e1 = e2
    iconst_0
    goto L7
L6:
```

```
    iconst_1
L7:
.line 95
    iload 3                 ; load G from 3
    bipush 11
    if_icmpgt L8        ; e1 > e2
    iconst_0
    goto L9
L8:
    iconst_1
L9:
    iand
    ior
    ldc 0                   ; load constant mnope
    ior
    ifeq L10          ; e1 is false
.line 96
    iconst_1
    iload 10                ; load E from 10
    iadd                            ; e1 right hand for assignment
    dup
    istore 10               ; store e1 in E
    pop
L10:
.line 103
    ldc 44                   ; load constant minus2
    iload 10                ; load E from 10
    isub                            ; e1 right hand for assignment
```

```
    dup
    istore 12               ; store e1 in N
    pop
.line 105
    iload 12                ; load N from 12
    bipush 21
    if_icmplt L11         ; e1 < e2
    iconst_0
    goto L12
L11:
    iconst_1
L12:                            ; e1 right hand for assignment
    dup
    istore 13               ; store e1 in tosmall
    pop
.line 106
    iload 12                ; load N from 12
.line 107
    iload 13                ; load tosmall from 13
    ifeq L13          ; e1 is false
.line 108
    bipush 30
    goto L14
L13:
.line 110
    iconst_0
L14:
    iadd                            ; e1 right hand for assignment
```

```
    dup
    istore 12               ; store e1 in N
    pop
.line 116
    iload 12                ; load N from 12
    bipush 7
    iadd
.line 116
    iload 9                 ; load Z from 9
    iload 12                ; load N from 12
    iadd
    bipush 7
    irem
    isub                            ; e1 right hand for assignment
    dup
    istore 14               ; store e1 in P
    pop
.line 121
    iload 14                ; load P from 14
    bipush 31
    if_icmpgt L15         ; e1 > e2
    iconst_0
    goto L16
L15:
    iconst_1
L16:
    ifeq L17          ; e1 is false
.line 122
```

```
    iconst_4
                            ; e1 right hand for assignment
    dup
    istore 15               ; store e1 in month
    pop
.line 123
    iload 14                ; load P from 14
    bipush 31
    irem                            ; e1 right hand for assignment
    dup
    istore 16               ; store e1 in day
    goto L18
L17:
.line 125
    iconst_3
                            ; e1 right hand for assignment
    dup
    istore 15               ; store e1 in month
    pop
.line 126
    iload 14                ; load P from 14
                            ; e1 right hand for assignment
    dup
    istore 16               ; store e1 in day
L18:
    pop
.line 140
    invokestatic Main/printdatum()V
```

```
                            ; e2 if true expression
L19:
    return
.end method
.method public static checkjaar(I)I
.limit stack 3
.limit locals 3
.line 9
.line 10
.line 11
    iload 0                 ; load jaar from 0
    iconst_0
    if_icmplt L0        ; e1 < e2
    iconst_0
    goto L1
L0:
    iconst_1
L1:
    ifeq L5          ; e1 is false
.line 12
    bipush 106                  ; ldc 'j'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 12
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
```

```
505     invokevirtual java/io/PrintStream/println(C)V
    .line 12
        bipush 122                  ; ldc 'z'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
510     invokevirtual java/io/PrintStream/println(C)V
    .line 12
        bipush 117                  ; ldc 'u'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
515     invokevirtual java/io/PrintStream/println(C)V
    .line 12
        bipush 115                  ; ldc 's'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
520     invokevirtual java/io/PrintStream/println(C)V
    .line 12
        bipush 95                 ; ldc '_'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
525     invokevirtual java/io/PrintStream/println(C)V
    .line 12
        bipush 109                  ; ldc 'm'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
530     invokevirtual java/io/PrintStream/println(C)V
    .line 12
        bipush 111                  ; ldc 'o'
```

```
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 116                  ; ldc 't'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 95                   ; ldc '_'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 122                  ; ldc 'z'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 105                  ; ldc 'i'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
```

```
.line 12
   bipush 106                ; ldc 'j'
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(C)V
.line 12
   bipush 110                ; ldc 'n'
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(C)V
.line 12
   bipush 95                 ; ldc '_'
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(C)V
.line 12
   bipush 103                ; ldc 'g'
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(C)V
.line 12
   bipush 101                ; ldc 'e'
   getstatic java/lang/System/out Ljava/io/PrintStream;
   swap
   invokevirtual java/io/PrintStream/println(C)V
.line 12
   bipush 98                 ; ldc 'b'
   getstatic java/lang/System/out Ljava/io/PrintStream;
```

```
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 111                  ; ldc 'o'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 114                  ; ldc 'r'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 101                  ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 12
    bipush 110                  ; ldc 'n'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
                            ; e2 if true expression
    goto L6
L5:
.line 14
    iload 0                 ; load jaar from 0
    ldc 2099                    ; load constant maxyear
```

```
    if_icmpgt L2        ; e1 > e2
    iconst_0
    goto L3
L2:
    iconst_1
L3:
    ifeq L4          ; e1 is false
.line 15
    bipush 104               ; ldc 'h'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 15
    bipush 101               ; ldc 'e'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 15
    bipush 98               ; ldc 'b'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 15
    bipush 116               ; ldc 't'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 15
```

```
645     bipush 95               ; ldc '_'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(C)V
    .line 15
650     bipush 103              ; ldc 'g'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(C)V
    .line 15
655     bipush 101              ; ldc 'e'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(C)V
    .line 15
660     bipush 100              ; ldc 'd'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(C)V
    .line 15
665     bipush 117              ; ldc 'u'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(C)V
    .line 15
670     bipush 108              ; ldc 'l'
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
```

```
    invokevirtual java/io/PrintStream/println(C)V
  .line 15
    bipush 100                ; ldc 'd'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
                              ; e2 if true expression
L4:
                                                          ; e3 if false expression
L6:
.line 18
    iload 0                   ; load jaar from 0
    iconst_0
    if_icmpge L7          ; e1 >= e2
    iconst_0
    goto L8
L7:
    iconst_1
L8:
.line 18
    iload 0                   ; load jaar from 0
    ldc 2099                      ; load constant maxyear
    if_icmple L9          ; e1 <= e2
    iconst_0
    goto L10
L9:
    iconst_1
L10:
```

```
    iand                            ; e1 right hand for assignment
    dup
    istore 1                ; store e1 in ok
    pop
    iload 1                 ; load ok from 1
    ireturn
.end method
.method public static nest(II)I
.limit stack 3
.limit locals 5
.line 47
.line 48
.line 49
    iload 0                 ; load stap from 0
    iconst_1
    if_icmpeq L0        ; e1 = e2
    iconst_0
    goto L1
L0:
    iconst_1
L1:
    ifeq L9          ; e1 is false
.line 50
    iconst_3
    iload 1                 ; load w from 1
    imul                            ; e1 right hand for assignment
    dup
    istore 2                ; store e1 in returnvalue
```

```
    pop
    goto L10
L9:
.line 52
    iload 0                 ; load stap from 0
    iconst_2
    if_icmpeq L2          ; e1 = e2
    iconst_0
    goto L3
L2:
    iconst_1
L3:
    ifeq L8           ; e1 is false
.line 53
    iload 1                 ; load w from 1
    iconst_4
    idiv                            ; e1 right hand for assignment
    dup
    istore 2                ; store e1 in returnvalue
    pop
.line 55
    iconst_1
                             ; e1 right hand for assignment
    dup
    istore 3                ; store e1 in loop
    pop
.line 56
L6:
```

```
.line 56
    iload 3                 ; load loop from 3
    ldc 12                    ; load constant twaalf
    if_icmple L4          ; e1 <= e2
    iconst_0
    goto L5
L4:
    iconst_1
L5:
    ifeq L7
.line 57
    iload 2                 ; load returnvalue from 2
    iconst_1
    isub                            ; e1 right hand for assignment
    dup
    istore 2                ; store e1 in returnvalue
    pop
.line 58
    iload 3                 ; load loop from 3
    iconst_1
    iadd                            ; e1 right hand for assignment
    dup
    istore 3                ; store e1 in loop
    pop
    goto L6
L7:
                            ; e2 if true expression
L8:
```

```
                                                        ; e3 if false expression
L10:
    iload 2                 ; load returnvalue from 2
    ireturn
.end method
.method public static copy(I)I
.limit stack 3
.limit locals 2
.line 80
.line 81
    bipush 42
    pop
    iload 0                 ; load jaar from 0
    ireturn
.end method
.method public static nested()I
.limit stack 3
.limit locals 3
.line 77
.line 78
.line 79
    getstatic Main/jaar I ; load global jaar
    iconst_5
    imul
    iconst_4
    idiv
    iload 5                 ; load X from 5
    isub
```

```
.line 79                    ; -     bipush 10
    bipush 10
    ineg
    iadd                            ; e1 right hand for assignment
    dup
    istore 0                ; store e1 in Z
    pop
.line 84
    iload 0                 ; load Z from 0
    invokestatic Main/copy(I)I                              ; e1 right hand for assignment
    dup
    istore 0                ; store e1 in Z
    pop
    iload 0                 ; load Z from 0
    ireturn
.end method
.method public static printdatum()V
.limit stack 3
.limit locals 2
.line 129
.line 130
    iload 16                ; load day from 16
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V
    pop
.line 132
```

```
    bipush 95                ; ldc '_'
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
                             ; e1 right hand for assignment
    dup
    istore 0                 ; store e1 in under
    pop
.line 133
    iload 15                 ; load month from 15
    iconst_3
    if_icmpeq L0         ; e1 = e2
    iconst_0
    goto L1
L0:
    iconst_1
L1:
    ifeq L2          ; e1 is false
.line 134
    bipush 77                ; ldc 'M'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 134
    bipush 97                ; ldc 'a'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
```

```
    invokevirtual java/io/PrintStream/println(C)V
.line 134
    bipush 97                ; ldc 'a'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 134
    bipush 114               ; ldc 'r'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 134
    bipush 116               ; ldc 't'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
                           ; e2 if true expression
    goto L3
L2:
.line 136
    bipush 65                ; ldc 'A'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
.line 136
    bipush 112               ; ldc 'p'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
```

```
    invokevirtual java/io/PrintStream/println(C)V
 .line 136
    bipush 114                ; ldc 'r'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 136
    bipush 105                ; ldc 'i'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
 .line 136
    bipush 108                ; ldc 'l'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
                                                    ; e3 if false expression
L3:
.line 138
    iload 0                ; load under from 0
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(C)V
     getstatic Main/jaar I ; load global jaar
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println(I)V
    return
```

```
925 .end method
```