

SELMA

Vonk, J
(herhaler)
s0132778
Matenweg 75-201

Florisson, M
(herhaler)
s0165972
Box Calslaan 60-30

Studentassistent:
Edwin Smulders

July 6, 2011

Contents

1	Inleiding	5
2	Beknopte beschrijving	6
3	Problemen en oplossingen	7
3.1	Taalconstructie	7
3.2	Checker	7
3.3	Compiler	7
3.4	Randcode	8
4	Syntax, context-beperkingen en semantiek	9
4.1	Lexer - terminals	9
4.2	De basis - Programma	11
4.3	Expression_statement	11
4.4	Declaraties en types	11
4.4.1	Syntax	11
4.4.2	Context	12
4.4.3	Semantiek	12
4.4.4	Voorbeeld	12
4.5	Functiedeclaratie	12
4.5.1	Syntax	13
4.5.2	Context	13
4.5.3	Semantiek	13
4.5.4	Voorbeeld	13
4.6	Expressies - assignment	14
4.6.1	Syntax	14
4.6.2	Context	14
4.6.3	Semantiek	14
4.6.4	Voorbeeld	14
4.7	Expressies - OR	14
4.7.1	Syntax	15
4.7.2	Context	15
4.7.3	Semantiek	15
4.7.4	Voorbeeld	15
4.8	Expressies - AND	15
4.8.1	Syntax	16
4.8.2	Context	16
4.8.3	Semantiek	16
4.8.4	Voorbeeld	16
4.9	Expressies - Relaties	16
4.9.1	Syntax	16
4.9.2	Context	17
4.9.3	Semantiek	17
4.9.4	Voorbeeld	17

4.10	Expressies - plus en minus	17
4.10.1	Syntax	17
4.10.2	Context	17
4.10.3	Semantiek	17
4.10.4	Voorbeeld	18
4.11	Expressies - delen en vermenigvuldigen	18
4.11.1	Syntax	18
4.11.2	Context	18
4.11.3	Semantiek	18
4.11.4	Voorbeeld	18
4.12	Expressies - unaries	18
4.12.1	Syntax	19
4.12.2	Context	19
4.12.3	Semantiek	19
4.12.4	Voorbeeld	19
4.13	Expressies - toplevel	19
4.13.1	Syntax	20
4.13.2	Context	20
4.13.3	Semantiek	20
4.13.4	Voorbeeld	20
4.14	Unsigned constants	20
4.14.1	Syntax	20
4.14.2	Context	21
4.14.3	Semantiek	21
4.14.4	Voorbeeld	21
4.15	Identifier	21
4.15.1	Syntax	21
4.15.2	Context	22
4.15.3	Semantiek	22
4.15.4	Voorbeeld	22
4.16	Read	22
4.16.1	Syntax	22
4.16.2	Context	22
4.16.3	Semantiek	22
4.16.4	Voorbeeld	23
4.17	Print	23
4.17.1	Syntax	23
4.17.2	Context	23
4.17.3	Semantiek	23
4.17.4	Voorbeeld	23
4.18	If	23
4.18.1	Syntax	23
4.18.2	Context	24
4.18.3	Semantiek	24
4.18.4	Voorbeeld	24
4.19	While	24

4.19.1	Syntax	24
4.19.2	Context	24
4.19.3	Semantiek	24
4.19.4	Voorbeeld	25
4.20	Functieaanroep	25
4.20.1	Syntax	25
4.20.2	Context	25
4.20.3	Semantiek	25
4.20.4	Voorbeeld	25
4.21	Closed expression	25
4.21.1	Syntax	26
4.21.2	Context	26
4.21.3	Semantiek	26
4.21.4	Voorbeeld	26
4.22	Closed compoundexpression	26
4.22.1	Syntax	26
4.22.2	Context	26
4.22.3	Semantiek	26
4.22.4	Voorbeeld	27
5	Vertaalregels	28
5.1	Run	28
5.1.1	Program	28
5.2	Execute	29
5.2.1	ExpressionStatement	29
5.2.2	while	30
5.3	Evaluate	30
5.3.1	Compound Expression	30
5.3.2	if then else expression	30
5.3.3	Identifier	31
5.3.4	Integer Literal	31
5.3.5	Character Literal	31
5.3.6	Boolean Literals	31
5.3.7	Arithmetic, AND en OR	32
5.3.8	Relational operators	32
5.3.9	Unary Plus and Minus	33
5.3.10	NOT	33
5.3.11	Assignment	33
5.3.12	Print	33
5.3.13	Read	34
5.4	Elaborate	34

6	Beschrijving van Java programmatuur	36
6.1	main - SELMA	36
6.2	SELMAException	36
6.3	SELMATreeAdaptor	36
6.4	SELMATree	36
6.5	SymbolTable	37
6.5.1	SymbolTableException	38
6.6	IDEntry	38
6.7	CheckerEntry	38
6.8	CompilerEntry	38
7	Testplan en -resultaten	39
7.1	Pasen	41
8	Conclusies	42
9	Appendix	43
9.1	ANTLR Lexer & Parser specificatie	43
9.2	ANTLR Checker specificatie	48
9.3	ANTLR Codegenerator specificatie	55
9.4	ANTLR Codegenerator Stringtemplate specificatie	62
9.5	Invoer- en uitvoer van een uitgebreid testprogramma	70
9.5.1	SELMA-code van pasen	70
9.5.2	Jasmin-code van pasen	74

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. (black-box testing)

Hoe uitgebreid de te definiëren 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:

- 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). De SEL 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 retourneren, 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 hadden 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*

NB: Aangezien CD's lang niet zo hip zijn als wat het internet heeft te bieden is ons gehele werk óók te vinden op github:

<http://github.com/markflorisson88/selma/>

3 Problemen en oplossingen

Tijdens het maken van de taal zijn we uiteraard af en toe tegen problemen aangelopen. Nu hebben wij tijdens het practicum de calc-taal al ontwikkeld dus we hadden al wat handigheid met ANTLR - en ANTLR's soms wat aparte foutmeldingen.

3.1 Taalconstructie

Wat ons is opgevallen is dat je van te voren goed moet specificeren hoe je taal er uit moet zien. Door bijvoorbeeld onze keuze om overal SEMICOLON's achter te zetten - wat op zich logisch is - kregen we soms wat onwennige taalconstructies. Zo dien je ook een semicolon na een functiedeclaratie te zetten, want het is een declaratie - en ook een semicolon na een if-expressie voelt wat ongebruikelijk. Omdat echter alles een expressie is in deze taal vonden we het passend hier geen uitzonderingen op te gaan maken door sommige expressies niet met een semicolon af te sluiten.

De eis om een taal LL(1) te maken heeft echter niet echt problemen opgeleverd, behalve dat we de declaratie voor een assignment op een andere plek wouden doen in eerste instantie (onder expressies-toplevel), hierbij was echter met LL(1) geen onderscheid te maken tussen een identifier en een assignment.

Een ander punt waar het onderscheid moeilijk was, waren functies. Deze zijn namelijk niet te onderscheiden van identifiers, tot je een haakje-openen na een identifier ziet. We hebben overwogen om een lokale forward-lookup te gebruiken, dit hebben we echter snel bestempeld als "slim valsspelen" en we hebben een '@' voor alle functie-aanroepen gezet. Klinkt ook mooi, aangezien je ook daadwerkelijk verwijst naar een stuk eerder gedefinieerde code.

Soms was het noodzakelijk om een stevige herschrijfregel te gebruiken, om in de checker en compiler wat meer gemak te hebben. Zo hebben we UMIN en EXPRESSION_STATEMENT toegevoegd. En hebben we vormen zoals (ID (COMMA ID)* COLON type) naar $\hat{(ID\ TYPE)}_+$ omgeschreven.

3.2 Checker

De checker heeft vrijwel geen problemen opgeleverd, aangezien onze randcode - Java-helperclasses etc. - gewoon netjes aansloot en een hoop werk uit handen nam.

Waar we wel consequent tegenaan liepen waren de wat vage manieren waarop er data uit de boom te halen is. Dollartekens voor Tokens, of juist niet, het was soms wat onduidelijk.

3.3 Compiler

In eerste instantie is de compiler in TAM geschreven, toen het echter een project van 2 werd in plaats van 1, is er besloten om een tandje bij te zetten en SELMA

in een wat algemener geaccepteerde code-vorm te compilen: JVM.

Mark, wat barft aan JV

En barfen stringtemplat

3.4 Randcode

De randcode is deels gebaseerd op de symboltables gebruikt tijdens het practicum en neemt een hoop werk uit handen. De symboltable-entries zijn per onderdeel (checker, compiler) anders. Dit omdat we merkten dat er soms te weinig gegevens waren over declaraties.

In de boom zelf konden we ook niet genoeg info kwijt, vandaar dat we een extension op de normale Tree hebben gemaakt, SELMATree, waarin is op te slaan wat het type is van elke expressie en of er variabele onderdelen in een expressie zitten.

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 definiëren 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 zestal 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;	
BEGIN;	
END;	
COMPOUND;	
EXPRESSION_STATEMENT;	

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 . '
UNDERSCORE	' _ '	IF	' if '
		THEN	' then '
		ELSE	' else '
		FI	' fi '
		WHILE	' while '
		DO	' do '
		OD	' od '
		FUNCDEF	' function '
		FUNCRETURN	' return '
		FUNCTION	' @ '
Operators			
NOT	' ! '		
MULT	' * '		
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 semicolon 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 definiëren. 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. De functie declaratie die je ziet staan wordt apart besproken.

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

```
funcbody
  | FUNCDEF^ identifier LPAREN! (funcpars SEMICOLON!)* RPAREN
  ! funcbody
  : COLON type LCURLY compoundexpression FUNCRETUR
    expression SEMICOLON RCURLY -> ^(FUNCRETUR type
      compoundexpression expression)
  | LCURLY! compoundexpression RCURLY!
  ;
```

4.5.2 Context

- De naam van de functie moet uniek zijn.
- 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 wordt 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 identifier 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 assignment 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_all` staat voor "expression arithmetic level 1"

4.7.1 Syntax

```
expr_arithmetic
: expr_all
;

expr_all                                     //
    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 || false;
true || false || 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 && bar;
foo && false && 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 geretourneerde 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 $\text{expr_al6} == - \text{expr_al7}$

Bij UPLUS zal $\text{expr_al6} == \text{expr_al7}$

Bij NOT zal $\text{expr_al6} == ! \text{expr_al7}$

4.12.4 Voorbeeld

```
one := +1;
evil := -42;
foo := ! foobar;
```

4.13 Expressies - toplevel

Op het hoogste niveau 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 identifier is kan op IL(1) geen onderscheid worden gemaakt tussen identifier of een assignment. Vandaar dat een assignment bij expr_al1 is gedefinieerd.

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;
(foo bar);
```

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 gebeurt 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 retourneren
- De if retourneert de waarde en type van de expressie die wordt uitgevoerd. (na de then of else)
- De if retourneert void als er geen expressie wordt uitgevoerd.

4.18.3 Semantiek

Als de waarde binnen het ifstatement waar is dan zal de eerste compoundexpression worden uitgevoerd (na de then). Anders zal de andere compoundexpression 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 retourneren
- De while retourneert 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 gedefinieerde functie

4.20.1 Syntax

```
expr_funcall
: 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 retourneert 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 geretourneerde 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

- retourneert 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;  
};
```

5 Vertaalregels

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

De vertaling wordt gedaan door middel van een compiler (`g-files/SELMACompiler.g`) in ANTLR die executeert 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*`
- `fetch : Identifier -> Instruction*`
- `assign : Identifier -> Instruction*`
- `elaborate : VarDeclaration -> Instruction*`

Phare	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 S	Executeer statement S. Als S een expressie is dat een waarde op de stack genereert, pop. Dit verandert 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).
ID	fetch I	Laad de waarde van I en zet het op de stack.
ID	assign I	Pop de top van de stack en sla het op in I.

In onze vertaalregels zullen we variabelen omringen met `<en >`.

5.1 Run

5.1.1 Program

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

```
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 resulteert 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 jumpst 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 laatste instructie verwijderd. Dit gebeurt voordat het resultaat naar een Jasmin assembly 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] =
    if kind == CONST:
        ldc getvalue(<ID>)
    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.

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 arithmetische 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 relationele operator zoals <=, == etc wordt deze code gegenereerd met bijbehorende instructie. De operator 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, address] =  
    evaluate[E]  
    dup  
    istore <address>
```

`address` is het address van de locale variabele. 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

    istore address_of(<id>)

```

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`). `address_of` is hier een functie die gegeven een identifier zijn address als locale variabele ophaalt. Afhankelijk van het type (`boolean`, `integer` of `character`) wordt een aanroep gedaan naar de methoden `nextBoolean()`, `nextInt()` of `nextByte()` van `Scanner`.

5.4 Elaborate

De `elaborate` code functie genereert geen code, maar maakt entries aan in de compiler's symbol table.

```

elaborate[VarDeclaration type ID] =
  increase amount of local variables by one
  enter <ID> of <type> in symbol table

elaborate[ConstDeclaration type ID value] =
  increase amount of local variables by one

```

```
enter <ID> of <type> in symbol table  
set value on symbol table entry of <ID>
```

Constanten nemen geen ruimte in op de stack of als locale variabelen maar worden in de symbol table bijgehouden. Variabelen zijn locale variabelen in de JVM. Dit betekent dat variabelen in de global scope niet beschikbaar zijn voor functies, aangezien dat andere methoden zijn en geen closures over de main functie. Hiervoor zouden het statische of non-statische fields moeten zijn. Zie ook de subsectie over Identifiers 5.3.3.

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 gecompileerd

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

6.3 SELMATreeAdaptor

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

6.4 SELMATree

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

```

public enum SR_Kind {VAR, CONST};
public enum SR_Func {YES, NO};

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

```

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

```

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

```

6.5 SymbolTable

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

De symboltable kent een aantal functies, de belangrijkste zijn:

```

/**
 *
 * @return Address voor een locale variabele
 */
public int nextAddr(){
    /**
     * Constructor.
     * @ensure this.currentLevel() == -1
     */
    public SymbolTable() {
        currentLevel = -1;
        entries = new HashMap<String , Stack<Entry>>();
    }
    public void closeScope() {
        for (Map.Entry<String , Stack<Entry>> entry: entries.
            entrySet()){
            Stack<Entry> stack = entry.getValue();
            if ((stack != null) && (!stack.isEmpty()) && (stack
                .peek().level >= currentLevel)){
                Entry e = stack.pop();
                localCount = nextAddr > localCount ?
                    nextAddr : localCount;
                if (isLocal(e))
                    nextAddr--;
            }
            or when the id is already declared on the current
            level.
        }
    }
}

```

```

public void enter(Tree tree, Entry entry) throws
    SymbolTableException {
    String id = tree.getText();
    if (currentLevel < 0) {
        throw new SymbolTableException(tree, "Not_in_a_
            valid_scope.");
    }
}

```

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 klassen 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 vier extra waarden 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. Dan is er nog een variabele waar wordt onthouden of deze entry een functie is. En de vierde staan alle parameters van de functie in.

```

/** Het type van de entry */
public SR.Type type;
/** Wat voor soort entry het is (e.g. constante, variabele) */
public SR.Kind kind;
/** Of deze entry voor een functie is */
public SR.Func func;
/** Parameters voor een functie */
public ArrayList<Param> params;

```

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 en nog een aantal andere dingen. Dit wordt bijgehouden door:

```

/** Address van een locale variabele */
public int addr;
/** Value van een constante */
public int val;

```

```
/** Signature van een functie */  
public String signature;  
/** Geeft aan of de entry bij een globale variabele hoort */  
public boolean isGlobal;
```


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 succesvol), 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+ of 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 arithmetiek, 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
```

<pre> 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 </pre>	
<pre> Run test/test_if.selma ... OK Run test/test_operators.selma ... OK Run test/test_pasen.selma ... OK Run test/test_while.selma ... OK Run test/test_functions/correct_functions.SELMA ... OK Error test/test_functions/error_doublefunction.selma ... OK Error test/test_functions/error_nested.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 Run test/test_functions/recursion.selma ... OK Ran 24 test(s), SUCCESS=22, FAILURE=2 </pre>	

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 kolom geeft het type test aan, in dit geval 'Error' of 'Run'. Ter demonstratie is `test/sample.selma` bijgevoegd:

<pre> <output> h </pre>	
-----------------------------------	--

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

7.1 Pasen

Het grootste testprogramma heet `pasen`, deze wordt ook meegenomen in de test. `Pasen` berekent middels het algorithm van Gauss wanneer `pasen` valt in een gegeven jaar.

De SELMA-code is goed gedocumenteerd. De rekenstappen zijn soms wat omslachtig geschreven om gebruik te maken van alle mogelijkheden van de taal. In de appendix is de code te vinden van `pasen`, zowel in SELMA als Jasmin.

8 Conclusies

We hebben een taal gedefinieerd, uitgeschreven, regels aan toegevoegd en dit omgezet naar een stuk Jasmin. Zonder ons in al te veel bochten te moeten wringen om de taal werkend te krijgen of binnen de LL(1) restrictie zien te krijgen.

Tot zover lijkt ons dat het over het algemeen wel goed gegaan is - begrijp ons niet verkeerd, we hadden graag nog arrays geïmplementeerd - maar we hebben een werkende taal, welke niet aan elkaar geplakt zit van de lelijke oplossingen.

We hebben zeker 40 pagina's verslaglegging (met appendices 100+). Waarin we hebben geprobeerd alles uit te leggen, hier en daar op een wat informele toon, maar we weten wie het leest. En om de lezer continu met 'u' aan te spreken - en geen kleine grapjes te kunnen maken - is het wel een erg droog verslag om te lezen.

Het had ons persoonlijk erg leuk geleken om een optimizer te schrijven, dit is echter nogal wat werk en we zijn daar niet aan toegekomen. Bovendien dienen we aan de boekhouding te denken, in het verslag stond niet duidelijk of dat ons pluspunten zou opleveren. De uitdaging aan de optimizer was denk ik het puzzelen geweest, om net overal de code net iets vlotter te maken. Zo is er bijvoorbeeld al rekening gehouden door van elke expressie bij te houden of deze variabele onderdelenbevat, als een expressie compleet constant zou zijn dan zou je immers het net zo goed een maal kunnen uitrekenen en als dergelijk in de code te zetten.

Mark, mening?

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';
}
```

```

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

70     UMIN;
        UPLUS;

        BEGIN;
        END;
        COMPOUND;
75     EXPRESSION_STATEMENT;
    }

    @header {
        package SELMA;
80    }

    @lexer::header {
        package SELMA;
85    }

90

95

// Parser rules – program at line 100 due to the report

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

105 compoundexpression
    : cmp -> ^(COMPOUND cmp)
    ;

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

    //declaration

115 declaration
    // : VAR^ identifier (COMMA! identifier)* COLON! type
    // | CONST^ identifier (COMMA! identifier)* COLON! type EQ!
    unsignedConstant
    : VAR identifier (COMMA identifier)* COLON type
      -> ^(VAR type identifier)+
120   | 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
125   )+;
    type
        : INT
        | BOOL

```

```

130         | CHAR
        ;

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

135

140

//expression statement at line 146
145 expression_statement
    : expression -> ^(EXPRESSION_STATEMENT expression)
    ;
    // note: - arithmetic can be "invisible" due to all the *-s that's why
    // it is nested
150 // - assignment can be "invisible" due to the ? that's why it can also
    // be only a identifier
    expression
    : expr_assignment
    ;

155 expr_assignment
    : expr_arithmetic (BECOMES^ expression)?
    ;

expr_arithmetic
160 : expr_al1
    ;

    expr_al1                                     //expression
    arithmetic level 1
    : expr_al2 (OR^ expr_al2)*
165 ;

    expr_al2
    : expr_al3 (AND^ expr_al3)*
    ;

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

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

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

    expr_al6
    : PLUS expr_al7
    -> ^(UPLUS expr_al7)
185 | MINUS expr_al7
    -> ^(UMIN expr_al7)
    | NOT expr_al7
    -> ^(NOT expr_al7)

```

```

190         | expr_al7
        ;

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

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

expr_print
210      : PRINT LPAREN expression (COMMA expression)* RPAREN
      -> ^(PRINT expression+)
      ;

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

expr_while
      : WHILE^ compoundexpression DO compoundexpression OD
      ;

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

225 expr_closedcompound
      : LCURLY^ compoundexpression RCURLY
      ;

expr_closed
230      : LPAREN! expression RPAREN!
      ;

235

240

//unsigned at line 244

unsignedConstant
245      : boolval
        | charval
        | intval
        ;

250 intval
      : NUMBER
      ;

boolval
255      : BOOLEAN
      ;

```



```

charval
    : CHARV
260     ;

identifier
    : ID
    ;

265 CHARV
    : APOSTROPHE (LETTER|UNDERSCORE) APOSTROPHE
    ;

270 BOOLEAN
    : TRUE
    | FALSE
    ;

275 ID
    : LETTER (LETTER | DIGIT)*
    ;

NUMBER
280     : DIGIT+
    ;

COMMENT
285     : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    | '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;

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

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

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

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

fragment LETTER
    : LOWER
    | UPPER
310     ;

fragment TRUE
    : 'true'
    ;

315 fragment FALSE
    : 'false'
    ;

320 //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>();
    // Keep track of whether we are assigning to an identifier
    int assigning = 0;

    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
55   {st.openScope();}
   compoundexpression
   {st.closeScope();}
   END)
;

60 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) {
65       node.SR_type=SR_Type.VOID;
       node.SR_kind=null;
    } else {
70       node.SR_type=e1.SR_type;
       node.SR_kind=e1.SR_kind;
    }
}
;

75 expression_statement
: ^(node=EXPRESSION_STATEMENT expression)
{
    SELMATree e1 = (SELMATree)node.getChild(node.getChildCount()
-1);
    // System.err.println("..." + e1 + " " + e1.getLine());
80    $node.SR_type = e1.SR_type;
    $node.SR_kind = e1.SR_kind;
}
;

85 declaration
: ^(node=VAR type id=ID)
{
    st.enter($id, new CheckerEntry(((SELMATree) node.getChild(0)).
getSelmaType(),
                                SR_Kind.VAR));
90 }
| ^(node=CONST type val id=ID)
{
    int type = node.getChild(0).getType();
    int val = node.getChild(1).getType();
95
    switch (type) {
        case INT:
            if (val!=NUMBER) throw new SELMAException(id," Expecting int
-value");
            st.enter($id, new CheckerEntry(SR_Type.INT, SR_Kind.CONST));
100            break;
        case BOOL:
            if (val!=BOOLEAN) throw new SELMAException(id," Expecting
bool-value");
            st.enter($id, new CheckerEntry(SR_Type.BOOL, SR_Kind.CONST));
            break;
105        case CHAR:
            if (val!=CHARV) throw new SELMAException(id," Expecting char
-value");
            st.enter($id, new CheckerEntry(SR_Type.CHAR, SR_Kind.CONST));
            break;
    }
110 }
| ^(node=FUNCDEF funcname=ID)
{
    //enter as void
    if (st.funclevel != 0)

```

```

115         throw new SELMAException($funcname, "Cannot nest functions");
        st.enter($funcname, new CheckerEntry(SR_Type.VOID, SR_Kind.VAR,
        SR_Func.YES));
        st.enterFuncScope();
    }
    (param=ID typ1=(INT|BOOL|CHAR)
120 {
        st.addParamToFunc($funcname, param, $typ1);
    }
        )*
    (
125         ^ (node=FUNCRETURN type
    {
        SELMATree type = (SELMATree) node.getChild(0);
        st.retrieve($funcname).type = type.getSelmaType();
    } compoundexpression expression
130 {
        SELMATree expr = (SELMATree) node.getChild(2);
        matchType(type, expr.SR_type);
    })
    | (compoundexpression))
135 {
        //scope of function
        st.leaveFuncScope();
    });

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

145 val
    : node=NUMBER
    | node=CHARV
    | node=BOOLEAN
150 ;

expression
    : ^ (node=(MULT|DIV|MOD|PLUS|MINUS) expression expression)
    {
155     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(
160             $node,
            String.format("Wrong types must be int (found %s and %s)", e1.
                SR_type, e2.SR_type));
    }

    $node.SR_type = SR_Type.INT;

165     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;
170 }

    | ^ (node=(RELS|RELSE|RELG|RELGE) expression expression)
    {
    SELMATree e1 = (SELMATree) node.getChild(0);
175     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;
180

```

```

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

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

195     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;
200     }

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

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

210     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;
215     }

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

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

225     $node.SR_kind=e1.SR_kind;
    }

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

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

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

    | ^(node=IF {st.openScope();} compoundexpression
240     THEN {st.openScope();} compoundexpression {st.closeScope()
        ;}
        (ELSE {st.openScope();} compoundexpression {st.closeScope()
        ;})?
        {st.closeScope();})
    {
245     SELMATree e1 = (SELMATree)node.getChild(0);
    SELMATree e2 = (SELMATree)node.getChild(2);

```

```

250 SELMATree e3 = (SELMATree)node.getChild(4);

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

255     if (e3==null) { //no else
        $node.SR_type=SR.Type.VOID;
        $node.SR_kind=null;
    } else { // there is a else
255         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
260                 $node.SR_kind=SR.Kind.VAR;
        } else {
            $node.SR_type=SR.Type.VOID;
            $node.SR_kind=null;
        }
265     }
}

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

275    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;
280 }

    | ^(node=READ (id=ID
285    {
        CheckerEntry entry = st.retrieve($id);
        entry.initialized = true;
        if (entry.kind!=SR.Kind.VAR)
            throw new SELMAException($id," Must be a variable");
    })+
290    {
        if ($node.getChildCount() == 1) {
            $node.SR_type = st.retrieve(node.getChild(0)).type;
            $node.SR_kind = SR.Kind.VAR;
        } else {
295            $node.SR_type = SR.Type.VOID;
            $node.SR_kind = null;
        }
    }
    )

300    | ^(node=PRINT expression+)
    {
    for (int i=0; i<((SELMATree)node).getChildCount(); i++){
    if (((SELMATree)node).getChild(i)).SR_type == SR.Type.VOID)
305        throw new SELMAException($node, "Can not be of type void");
    }
    if ($node.getChildCount() == 1){
        $node.SR_type = ((SELMATree) node.getChild(0)).SR_type;
        $node.SR_kind = SR.Kind.VAR;
310    } else {
        $node.SR_type = SR.Type.VOID;
        $node.SR_kind = null;
    }
}

```

```

    }
    }
315     -> ^ (PRINT expression)+
        | ^ (node=FUNCTION ID expression*)
    {
        //retrieve function (if existent)
320     SELMATree func = (SELMATree)$node;
        CheckerEntry entry = st.retrieve($ID);
        $node.SR_type=entry.type;
        $node.SR_kind=entry.kind;

325     //matchparamlists
        //same length?
        int argc = func.getChildCount()-1;
        if (entry.params.size() != argc)
            throw new SELMAException(node, String.format(
330                 "\%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)
335                 throw new SELMAException(expr,"Param is not of the right type");
        }
    }

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

        CheckerEntry ident = st.retrieve(e1);
        ident.initialized = true;

350     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);

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

    | ^ (node=LCURLY {st.openScope();} compoundexpression {st.
        closeScope();} RCURLY)
360     {
        SELMATree e1 = (SELMATree) node.getChild(0);
        $node.SR_type = e1.SR_type;
        $node.SR_kind = e1.SR_kind;
    }

365     | node=NUMBER
    {
        $node.SR_type=SR_Type.INT;
        $node.SR_kind=SR_Kind.CONST;
370     }

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

```

```

    | node=CHARV
    {
380 $node.SR_type=SR.Type.CHAR;
    $node.SR_kind=SR.Kind.CONST;
    }

    | node=ID
385 {
    CheckerEntry entry = st.retrieve($node);
    if (assigning == 0 && !entry.isInitialized(st))
        throw new SELMAException($node,
            "Variable " + $node.text + " is not initialized yet
            .");
390 $node.SR_type=entry.type;
    $node.SR_kind=entry.kind;
    }
    ;

```


9.3 ANTLR Codegenerator specificatie

```
tree grammar SELMACompiler;

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

10 @header {
    package SELMA;
    import SELMA.SELMA;
    import SELMA.SELMATree.SR_Type;
    import SELMA.SELMATree.SR_Kind;
15   import SELMA.SELMATree.SR_Func;

    import java.lang.StringBuilder;
}

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

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

    int curStackDepth = 0;
    int maxStackDepth = 0;
30   int labelNum = 0;

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

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

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

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

55         stack.push(o);

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

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

```

65         st.leaveFuncScope();
           curStackDepth = o.curStackDepth;
           maxStackDepth = o.maxStackDepth;
           labelNum = o.labelNum;
           st.nextAddr = o.nextAddr;
70         st.localCount = o.localCount;
       }

       private String getTypeDenoter(SR_Type type) {
           return st.getTypeDenoter(type, false);
75       }

       private String getTypeDenoter(SR_Type type, boolean printing) {
           return st.getTypeDenoter(type, printing);
       }
80   }

   program
   : ^(node=BEGIN {st.openScope();} compoundexpression END)
   { SELMATree expr = (SELMATree) $node.getChild(0);
85   int localsCount = st.getLocalsCount();
     st.closeScope();
   }
   -> program(instructions={$compoundexpression.st},
              source_file={SELMA.inputFilename},
              stack_limit={maxStackDepth + 3}, // +3 for print
              locals_limit={localsCount + 1}, // +1 for the String[] argv
              parameter
              fields={st.globals},
              pop={expr.SR_type != SR_Type.VOID})
   ;

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

100  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})
105   | ^(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})

110   | ^(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?
115   | ^(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)
120   {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);
        enterFuncScope();
135 int paramCount = 0;
        StringBuilder signatureBuilder = new StringBuilder("");
        boolean hasReturnType = false;
    } (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)
    {
        SELMATree returnTypeNode = (SELMATree) $return_node.getChild(0);

        signatureBuilder.append("");
150 signatureBuilder.append(getTypeDenoter(returnTypeNode.
            getSelmaType()));
        funcentry.signature = signatureBuilder.toString();

        hasReturnType = true;
    }

155 (body+=compoundexpression) retexpr=expression)
    | ({funcentry.signature = signatureBuilder.toString() + "V";} body+=
        compoundexpression)
    )
    {
160 SELMATree funcbody;
        int stackLimit = maxStackDepth + 3;
        int localsLimit = st.getLocalsCount();

        if ($return_node == null)
165 funcbody = (SELMATree) $node.getChild(paramCount * 2 + 1);
        else
            funcbody = (SELMATree) $return_node.getChild(1);

        leaveFuncScope();
170 }
    )
    -> function(funcname={$funcname.text},
        body={$body},
        signature={funcentry.signature},
175 return_expression={$retexpr.st},
        is_void={funcentry.signature.endsWith("V")},
        pop={funcbody.SR.type != SR.Type.VOID},
        stack_limit={stackLimit},
        locals_limit={localsLimit + 1},
180 line={$node.getLine()})
    ;

expression_statement
: ^(node=EXPRESSION.STATEMENT e1=expression) { curStackDepth--; }
185 -> exprStat(e1={e1.st}, line={$node.getLine()}, pop={$node.SR.type !=
    SR.Type.VOID})
    ;

```

```

expression
//double arg expression
190 : ^(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={"/"})
195 | ^(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--; }
200 -> 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={"-"})

205 | ^(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"})
210 | ^(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
    ++})

215 | ^(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--; }
220 -> 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() },
225 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
    ++})
230 | ^(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
    ++})
235 //single arg expression

```

```

| ^ (UPLUS e1=expression)
{ $st=$e1.st; }

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

| ^ (node=NOT e1=expression)
-> not(e1={$e1.st}, line={node.getLine()},
      label_num1={labelNum++}, label_num2={labelNum++})
245

//CONDITIONAL
| ^ (node=IF { st.openScope(); } ec1=compoundexpression { st.closeScope
(); } THEN
      { st.openScope(); } ec2=compoundexpression { st.closeScope
(); }
      (ELSE { st.openScope(); } ec3=compoundexpression { st.closeScope
250 (); } )?)
{ boolean ec3NotEmpty = $ec3.st != null;
  SELMATree expr2 = (SELMATree) node.getChild(2);
  SELMATree expr3 = null;
  if (ec3NotEmpty)
    expr3 = (SELMATree) node.getChild(4);
255
-> if (ec1={$ec1.st}, ec2={$ec2.st}, ec3={$ec3.st}, label_num1={labelNum
++},
      label_num2={ec3NotEmpty ? labelNum++ : 0}, ec3_not_empty={
ec3NotEmpty},
      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
      { st.openScope(); } ec2=compoundexpression { st.closeScope(); } OD
)
{ SELMATree expr2 = (SELMATree) node.getChild(2);
265 boolean pop = expr2.SR_type != SR.Type.VOID;
  if (pop)
    curStackDepth--;
}
-> while (ec1={$ec1.st}, ec2={$ec2.st}, pop={pop},
270 label_num1={labelNum++}, label_num2={labelNum++})

//IO

| ^ (node=READ ID+)
275 /*
{
  CompilerEntry entry = st.retrieve($id);
})
-> readSingle(id={$id.text}, addr={entry.addr},
280 is_bool={entry.type == SR.Type.BOOL},
is_int={entry.type == SR.Type.INT},
dup_top={$node.SR_type != SR.Type.VOID},
is_global={entry.level == 0},
type_denoter={getTypeDenoter(entry.type)})

*/
285 { boolean isExpr = $node.SR_type != SR.Type.VOID;
List<Integer> addrs = new ArrayList<Integer>();
List<Boolean> isBool = new ArrayList<Boolean>();
List<Boolean> isInt = new ArrayList<Boolean>();
290 List<Boolean> globals = new ArrayList<Boolean>();
List<String> ids = new ArrayList<String>();

for (int i = 0; i < $node.getChildCount(); i++) {
SELMATree child = (SELMATree) $node.getChild(i);
295 CompilerEntry entry = st.retrieve(child);

```

```

/*
System.err.println(String.format("\%s -> \%s = \%s (\%s)
    is_int=\%s",
    child.getText(), entry.getIdentifier(child.getText()),
    entry, getTypeDenoter(entry.type), entry.type == SR.Type
        .INT));
300
*/
    addrs.add(entry.addr);
    isBool.add(entry.type == SR.Type.BOOL);
    isInt.add(entry.type == SR.Type.INT);
    globals.add(entry.isGlobal);
305
    ids.add(entry.getIdentifier(child.getText()));
}
}
-> read(ids={ids}, addrs={addrs}, dup_top={isExpr},
    is_bool={isBool}, is_int={isInt},
310
    globals={globals}, line={node.getLine()})

| ^(node=PRINT (exprs+=expression)+)
{
    boolean isExpr = $node.SR_type != SR.Type.VOID;
    int childCount = ((SELMATree) node).getChildCount();
315
    List<Integer> labelNums1 = new ArrayList<Integer>();
    List<Integer> labelNums2 = new ArrayList<Integer>();
    List<String> typeDenoters = new ArrayList<String>();
    List<Boolean> exprIsBool = new ArrayList<Boolean>();
320

    if (!isExpr)
        curStackDepth -= childCount;

    for (int i = 0; i < childCount; i++) {
325
        SELMATree child = (SELMATree) $node.getChild(i);
        boolean isBool = child.SR_type == SR.Type.BOOL;
        if (isBool) {
            labelNums1.add(labelNum++);
            labelNums2.add(labelNum++);
330
        } else {
            labelNums1.add(0);
            labelNums2.add(0);
        }
        typeDenoters.add(getTypeDenoter(child.SR_type, true));
335
        exprIsBool.add(isBool);
    }
}
-> print(exprs={$exprs}, type_denoters={typeDenoters}, dup_top={
    isExpr},
    expr_is_bool={exprIsBool},
340
    label_nums1={labelNums1}, label_nums2={labelNums2}, line={
        $node.getLine()})

| ^(node=FUNCTION id=ID (exprs+=expression)*)
    -> funccall(id={$id.text}, signature={st.retrieve($id).signature
        }, exprs={$exprs})
//ASSIGN
345
| ^(BECOMES node=ID el=expression)
{
    CompilerEntry entry = st.retrieve(node);
    String ident = entry.getIdentifier($node.text);
    boolean isConst = entry.kind == SR.Kind.CONST;
350

    String typeDenoter = getTypeDenoter(entry.type);
    // System.err.println("assign " + ident + " " + entry + " " +
        typeDenoter);
}
-> assign(id={ident},
355
    type={entry.type},
    addr={st.retrieve($node).addr},
    el={$el.st},

```

```

360         is_global={entry.isGlobal},
           type_denoter={typeDenoter})

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

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

370 | 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 })
375 -> loadChar(val={ (int) c }, char={ $node.text }, line={ $node.getLine()
  })

| node=ID
  {
380   incrStackDepth();
     CompilerEntry entry = st.retrieve(node);
     String ident = entry.getIdentifier($node.text);
     boolean isConst = entry.kind == SR_Kind.CONST;
     String typeDenoter = getTypeDenoter(entry.type);
     // System.err.println("ID " + ident + " " + entry);
385   }
  -> loadVal(id={ ident }, addr={ entry.addr }, val={ entry.val }, is_const
    = { isConst },
     is_global={ entry.isGlobal }, type_denoter={ typeDenoter })

;

```

9.4 ANTLR Codegenerator Stringtemplate specificatie

```

//SELMA string template
group SELMA;

5  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;

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

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

15  .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;

25  <instructions>

    <if (pop)>
        pop
    <endif>

    return
    .end method
    >>

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

```



```

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

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

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

55     //Calculations
    uExpr (el, instr, line, op) ::= <<
        .line <line>
        <el>
        <instr>
    >>

60     not (el, label_num1, label_num2, line) ::= <<
        .line <line>
        <el>
        <instr>
    >>

65     ifeq L<label_num1>
        iconst_0
        goto L<label_num2>
        L<label_num1>;
        L<label_num2>;
    >>

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

80
>>

```

```

85 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>:
        >>
95 //Declare
    declareConst(id, val, type, addr) ::= <<
        ldc <val>
        istore <addr>
100 >>

    declareVar(id, type, addr) ::= <<
        >>
105 //Load
    loadNum(val, iconst, bipush) ::= <<
    <if (iconst)>
        iconst_<val>
    <elseif (bipush)>
        bipush <val>
110 <else>
        ldc <val>
    <endif>
    >>
115 loadVal(id, addr, val, is_const, is_global, type-denoter) ::= <<
    <if (is_const)>
        ldc <val>
        ; load constant <id>
120 <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, el, is_global, type_denoter) ::= <<
    <el>
    dup
    <if (is_global)>
      putstatic Main/<id> <type_denoter>
    <else>
      istore <addr>              ; store el in <id>
    <endif>
  >>
145
  read(ids, addrs, dup_top, is_bool, is_int, globals, line) ::= <<
    .line <line>
    <ids, is_bool, is_int, globals, addrs :
      { id, b, is_int, g, a | <readSingle(id=id,
150                                     is_bool=b, is_int=is_int,
                                     is_global=g, addr=addrs,
                                     dup_top=dup_top
                                     ) > }; separator="\n">
  >>
155
  >>
160
  readSingle(id, addr, is_bool, is_int, dup_top, is_global) ::= <<
    ; read <id> <addr> <is_bool> <is_int>
    ; getstatic Main/scanner_field Ljava/util/Scanner;
    <if (is_bool)>
      invokevirtual java/util/Scanner/nextBoolean()Z
165

```

```

170 <elseif (is_int)>>
      invokevirtual java/util/Scanner/nextInt() I
175 <else>
      invokevirtual java/util/Scanner/nextByte() B
      <endif>
      <if (dup_top)>
        dup
      <endif>
180 <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>
      >>
190
195 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) > }>
        >>
200
205 printSingle(expr, type_denoter, dup_top, is_bool, label_num1, label_num2) ::= <<
      <expr>

```

```

210 < if (dup_top)>
    dup
<endif>

215 < if (is_bool)>
    ifeq L<label_num1>
    ldc "true"
    goto L<label_num2>
    L<label_num1>:
    ldc "false"
    L<label_num2>:
<endif>

220     getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println(<type_denoter>)V
    >>
225 //conditionals
    if (ec1, ec2, ec3, label_num1, label_num2, ec3_not_empty, pop1, pop2) ::= <<
<ec1>
    ifeq L<label_num1>
    ; e1 is false
    ; e2 if true expression
<ec2>
    < if (pop1)>
    pop
235 <endif>
    < if (ec3_not_empty)>
    goto L<label_num2>
    <endif>
240 L<label_num1>:
    < if (ec3_not_empty)>
    <ec3>
    < if (pop2)>
    pop
    ; pop
    ; e3 if false expression
245 <endif>
    L<label_num2>:
    <endif>
    >>
250

```

```

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

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

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

290 funcall(id, signature, exprs) ::= <<

```

```
<exprs; separator="\n">
  invokestatic Main/<id>signature>
>>
```

9.5 Invoer- en uitvoer van een uitgebreid testprogramma

De code van pasen.

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:
5 */

const maxyear: integer = 2099;

function checkjaar(jaar: integer): boolean {
10     var ok: boolean;
    if jaar < 0; then
        print('T', 'E', ' ', 'V', 'R', 'O', 'E', 'G');
    else
        if jaar > maxyear; then
15             print('T', 'E', ' ', 'L', 'A', 'A', 'T');
        fi;
    fi;
    ok := (jaar>=0)&& jaar<=maxyear;
    return ok;
20 };

//<output>P</output>
//<output>A</output>
//<output>A</output>
25 //<output>S</output>
//<output>J</output>
//<output>A</output>
//<output>A</output>
//<output>R</output>
30 //<output>_</output>
//<output>_</output>
print('P', 'A', 'A', 'S', ' ', 'J', 'A', 'A', 'R', ' ', ' ', ' ');

var jaar: integer;
35 //<input>1991</input>
read(jaar);

const negentien: integer = 19;

40 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;
45     G := G+1;
    //<output>_</output>
    //<output>G</output>
    //<output>16</output>
50 print(' ');
print('G');
print(G);

// Bepaal het eeuwtaal:
```



```

55 //      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;
      //<output>_</output>
60 //<output>C</output>
      //<output>20</output>
      print('_');
      print('C');
      print(C);
65
      //      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;
          returnvalue := 0;
75          if stap==1; then
              returnvalue := 3*w;
          else
              if stap == 2; then
80                  returnvalue := w/4;
                  var loop: integer;
                  loop := 1;
                  while loop <= twaalf; do
                      returnvalue := returnvalue - 1;
                      loop := loop + 1;
85                  od;
              fi;
          fi;
          return returnvalue;
      };
90      X := @nest(stap+1,@nest(stap,C,));
      //<output>_</output>
      //<output>X</output>
      //<output>3</output>
      print('_');
      print('X');
95      print(X);

      //      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.
100      var Y: integer;
          if jaar >= 1900; then
              Y := 1;
          else
105              Y := (8*C)/25 - 5;
          fi;

      function copy(jaar: integer): integer {
          42;
          return jaar;
110      };
      //<output>_</output>
      //<output>Y</output>
      //<output>1</output>
      print('_');
      print('Y');
115      print(Y);

```

```

120 // Zoek de zondag:
// Vermenigvuldig het jaartal met 5, geheeldeelt 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;
    Z := @copy(Z,);
    return Z;
125 };
var Z: integer;
Z := @nested();
//<output>_</output>
//<output>Z</output>
130 //<output>2475</output>
print('_ ');
print('Z');
print(Z);

135 // Bepaal de epacta:
// 11 maal G + 20 + Y. Trek daarvan X af, geheeldeelt 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;
140 if EE==24 || (E==25&&G>11) || mnope; then
    E := 1+E;
fi;
//<output>_</output>
//<output>E</output>
145 //<output>14</output>
print('_ ');
print('E');
print(E);

150 // 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;
155 var tosmall: boolean;
tosmall := N<21;
N := N+
    if tosmall; then
        30;
    else
160         0;
    fi;
//<output>_</output>
//<output>N</output>
165 //<output>30</output>
print('_ ');
print('N');
print(N);

170 // Nu door naar zondag:
// Tel Z en N op. Geheeldeelt 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;
//<output>_</output>
175 //<output>P</output>
//<output>31</output>
print('_ ');
print('P');

```

```

180 print(P);
// 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
185     month := 4;
        day := P%31;
    else
        month := 3;
        day := P;
190 fi;

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

//<output>_</output>
205 //<output>_</output>
//<output>31</output>
//<output>_</output>
//<output>M</output>
//<output>a</output>
210 //<output>a</output>
//<output>r</output>
//<output>t</output>
//<output>_</output>
//<output>1991</output>
215     print('-');
        print('-');
        @printdatum();
fi;

```

9.5.2 Jasmin-code van pasen

```
.source test/test_pasen.selma
.class public Main
.super java/lang/Object
.field public static scanner_field Ljava/util/Scanner;
5  .field public static N_1 I
  .field public static C_1 I
  .field public static tosmall_1 I
  .field public static EE_1 I
  .field public static month_1 I
10  .field public static day_1 I
  .field public static Y_1 I
  .field public static P_1 I
  .field public static jaar_1 I
  .field public static E_1 I
15  .field public static Z_1 I
  .field public static G_1 I
  .field public static X_1 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 13
    .limit locals 20
    new java/util/Scanner
    dup
    ; dup for <init>
```

30	<pre> 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 32 bipush 80 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 32 bipush 65 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 32 bipush 65 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 32 bipush 83 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 32 bipush 74 getstatic java/lang/System/out Ljava/io/PrintStream; swap </pre>	35 40 45 50 55
----	---	--

```

        invokevirtual java/io/PrintStream/println (C)V
    .line 32
        bipush 65
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println (C)V
    .line 32
        bipush 65
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println (C)V
    .line 32
        bipush 82
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println (C)V
    .line 32
        bipush 95
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println (C)V
    .line 32
        bipush 95
        getstatic java/lang/System/out Ljava/io/PrintStream;
        swap
        invokevirtual java/io/PrintStream/println (C)V
    .line 36
        ; read jaar_1 2 false true

```

85	<pre> getstatic Main/scanner_field Ljava/util/Scanner; invokevirtual java/util/Scanner/nextInt() I dup putstatic Main/jaar_1 I pop .line 41 getstatic Main/jaar_1 I ; load global jaar_1 invokestatic Main/checkjaar(I) I ifeq L19 ; e1 is false .line 44 .line 45 getstatic Main/jaar_1 I ; load global jaar_1 ldc 19 ; load constant negentien_1 irem dup putstatic Main/G_1 I pop .line 46 getstatic Main/G_1 I ; load global G_1 iconst_1 iadd dup putstatic Main/G_1 I pop .line 50 bipush 95 dup getstatic java/lang/System/out Ljava/io/PrintStream; </pre>	<pre> ; e1 right hand for assignment ; e1 right hand for assignment </pre>
90		
95		
100		
105		
110		

	swap	
	invokevirtual java/io/PrintStream/println (C)V	
	pop	
115	.line 51	
	bipush 71	; ldc 'G'
	dup	
	getstatic java/lang/System/out Ljava/io/PrintStream;	
	swap	
120	invokevirtual java/io/PrintStream/println (C)V	
	pop	
	.line 52	
	getstatic Main/G_1 I ; load global G_1	
	dup	
	getstatic java/lang/System/out Ljava/io/PrintStream;	
	swap	
125	invokevirtual java/io/PrintStream/println (I)V	
	pop	
	.line 57	
	getstatic Main/jaar_1 I ; load global jaar_1	
		; e1 right hand for assignment
130	dup	
	putstatic Main/C_1 I	
	pop	
	.line 58	
	getstatic Main/C_1 I ; load global C_1	
	bipush 100	
	idiv	
	iconst_1	
135		

140	iadd		; e1 right hand for assignment
	dup		
	putstatic	Main/C_1 I	
	pop		
	.line 62		
145	bipush 95		; ldc '_'
	dup		
	getstatic	java/lang/System/out	Ljava/io/PrintStream;
	swap		
	invokevirtual	java/io/PrintStream/println (C)V	
150	pop		
	.line 63		
	bipush 67		; ldc 'C'
	dup		
	getstatic	java/lang/System/out	Ljava/io/PrintStream;
155	swap		
	invokevirtual	java/io/PrintStream/println (C)V	
	pop		
	.line 64		
	getstatic	Main/C_1 I	; load global C_1
	dup		
160	getstatic	java/lang/System/out	Ljava/io/PrintStream;
	swap		
	invokevirtual	java/io/PrintStream/println (I)V	
	pop		
165	.line 90		
	ldc 1		; load constant stap_1
	iconst_1		

170	iadd	
	ldc 1	; load constant stap_1
	getstatic Main/C_1 I	; load global C_1
	invokestatic Main/nest(II)I	
	invokestatic Main/nest(II)I	
	dup	
	putstatic Main/X_1 I	
	pop	
175	.line 94	
	bipush 95	; ldc ' _ '
	dup	
	getstatic java/lang/System/out Ljava/io/PrintStream;	
	swap	
	invokevirtual java/io/PrintStream/println(C)V	
	pop	
180	.line 95	
	bipush 88	; ldc 'X'
	dup	
	getstatic java/lang/System/out Ljava/io/PrintStream;	
	swap	
	invokevirtual java/io/PrintStream/println(C)V	
	pop	
185	.line 96	
	getstatic Main/X_1 I	; load global X_1
	dup	
	getstatic java/lang/System/out Ljava/io/PrintStream;	
	swap	
	invokevirtual java/io/PrintStream/println(I)V	
190		
195		

	pop	
	.line 101	
	getstatic Main/jaar_1 I ; load global jaar_1	
	ldc 1900	
	if_icmpge L0 ; e1 >= e2	
	iconst_0	
	goto L1	
	L0:	
	iconst_1	
	L1:	
	ifeq L2 ; e1 is false	
	.line 102	
	iconst_1	
		; e1 right hand for assignment
	dup	
	putstatic Main/Y_1 I	
	goto L3	
	L2:	
	.line 104	
	bipush 8	
	getstatic Main/C_1 I ; load global C_1	
	imul	
	bipush 25	
	idiv	
	iconst_5	
	isub	
	dup	
	putstatic Main/Y_1 I	
		; e1 right hand for assignment

225	L3:	pop	
230	.line 114	bipush 95	; ldc '_'
		dup	
		getstatic java/lang/System/out	Ljava/io/PrintStream;
		swap	
		invokevirtual java/io/PrintStream/println (C)V	
		pop	
	.line 115	bipush 89	; ldc 'Y'
		dup	
		getstatic java/lang/System/out	Ljava/io/PrintStream;
		swap	
		invokevirtual java/io/PrintStream/println (C)V	
		pop	
	.line 116	getstatic Main/Y_1 I	; load global Y_1
		dup	
		getstatic java/lang/System/out	Ljava/io/PrintStream;
		swap	
		invokevirtual java/io/PrintStream/println (I)V	
		pop	
	.line 127	invokestatic Main/nested () I	; e1 right hand for assignment
		dup	
		putstatic Main/Z_1 I	
		pop	
250			

```

255 .line 131
    bipush 95          ; ldc ' _ '
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println (C)V
    pop
260 .line 132
    bipush 90          ; ldc 'Z'
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println (C)V
    pop
265 .line 133
    getstatic Main/Z_1 I ; load global Z_1
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println (I)V
    pop
270 .line 138
    ldc 11
    ; load constant elf_1
275 .line 138
    getstatic Main/G_1 I ; load global G_1
    bipush 20
    iadd
    getstatic Main/Y_1 I ; load global Y_1

```

280	iadd	
	imul	
	getstatic Main/X_1 I ; load global X_1	
	isub	
	bipush 30	
285	irem	; e1 right hand for assignment
	dup	
	putstatic Main/EE_1 I	
		; e1 right hand for assignment
	dup	
	putstatic Main/E_1 I	
	pop	
	.line 140	
	getstatic Main/EE_1 I ; load global EE_1	
	bipush 24	
	if_icmpeq L4	; e1 = e2
295	iconst_0	
	goto L5	
	L4:	
	iconst_1	
300	L5:	
	.line 140	
	getstatic Main/E_1 I ; load global E_1	
	bipush 25	
	if_icmpeq L6	; e1 = e2
305	iconst_0	
	goto L7	
	L6:	

	iconst_1	
L7:		
310	.line 140	
	getstatic Main/G_1 I ; load global G_1	
	bipush 11	
	if_icmpgt L8 ; e1 > e2	
	iconst_0	
315	goto L9	
L8:		
	iconst_1	
L9:		
	iand	
	ior	
320	ldc 0 ; load constant mnope_1	
	ior	
	ifeq L10 ; e1 is false	
	.line 141	
325	iconst_1	
	getstatic Main/E_1 I ; load global E_1	
	iadd ; e1 right hand for assignment	
	dup	
	putstatic Main/E_1 I	
330	pop	
L10:		
	.line 146	
	bipush 95 ; ldc '_'	
	dup	
335	getstatic java/lang/System/out Ljava/io/PrintStream;	

	swap	
	invokevirtual java/io/PrintStream/println (C)V	
	pop	
.line 147		
340	bipush 69 ; ldc 'E'	
	dup	
	getstatic java/lang/System/out Ljava/io/PrintStream;	
	swap	
	invokevirtual java/io/PrintStream/println (C)V	
	pop	
345	.line 148	
	getstatic Main/E_1 I ; load global E_1	
	dup	
	getstatic java/lang/System/out Ljava/io/PrintStream;	
	swap	
	invokevirtual java/io/PrintStream/println (I)V	
	pop	
.line 154		
350	ldc 44 ; load constant minus2_1	
	getstatic Main/E_1 I ; load global E_1	
	isub ; e1 right hand for assignment	
	dup	
	putstatic Main/N_1 I	
	pop	
355	.line 156	
	getstatic Main/N_1 I ; load global N_1	
	bipush 21	
360	if_icmplt L11 ; e1 < e2	

365	iconst_0	
	goto L12	
L11:		
	iconst_1	
L12:		; e1 right hand for assignment
	dup	
370	putstatic Main/tosmall_1 I	
	pop	
	.line 157	
	getstatic Main/N_1 I ; load global N_1	
	.line 158	
375	getstatic Main/tosmall_1 I ; load global tosmall_1	
	ifeq L13	; e1 is false
	.line 159	
	bipush 30	
	goto L14	
380	L13:	
	.line 161	
	iconst_0	
L14:		
	iadd	
	dup	
385	putstatic Main/N_1 I	
	pop	
	.line 166	
	bipush 95	
	dup	; ldc '_'
390	getstatic java/lang/System/out Ljava/io/PrintStream;	

```

395 swap
    invokevirtual java/io/PrintStream/println (C)V
    pop
    .line 167
    bipush 78          ; ldc 'N'
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println (C)V
    pop
    .line 168
    getstatic Main/N_1 I ; load global N_1
    dup
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/println (I)V
    pop
    .line 173
    getstatic Main/N_1 I ; load global N_1
    bipush 7
    iadd
    .line 173
    getstatic Main/Z_1 I ; load global Z_1
    getstatic Main/N_1 I ; load global N_1
    iadd
    bipush 7
    irem
    isub
    ; el right hand for assignment

```


	iconst_0	
	goto L16	
450	L15:	
	iconst_1	
	L16:	
	ifeq L17	; e1 is false
	.line 185	
455	iconst_4	
		; e1 right hand for assignment
	dup	
	putstatic Main/month_1 I	
	pop	
460	.line 186	
	getstatic Main/P_1 I ; load global P_1	
	bipush 31	
	irem	
		; e1 right hand for assignment
	dup	
465	putstatic Main/day_1 I	
	goto L18	
	L17:	
	.line 188	
	iconst_3	
		; e1 right hand for assignment
470	dup	
	putstatic Main/month_1 I	
	pop	
	.line 189	
475	getstatic Main/P_1 I ; load global P_1	

```

                                ; e1 right hand for assignment
dup
putstatic Main/day_1 I
L18:
480   .line 215
      bipush 95
                                ; ldc ' '
      dup
      getstatic java/lang/System/outLjava/io/PrintStream;
      swap
      invokevirtual java/io/PrintStream/println(C)V
485   pop
      .line 216
      bipush 95
                                ; ldc ' '
      dup
      getstatic java/lang/System/outLjava/io/PrintStream;
      swap
      invokevirtual java/io/PrintStream/println(C)V
490   pop
      .line 217
      invokestatic Main/printdatum()V
495   ; e2 if true expression
      L19:
      return
500   .end method
      .method public static checkjaar(I)I
      .limit stack 11
      .limit locals 3

```

505	.line 9	
	.line 10	
	.line 11	
	iload 0	; load jaar_1 from 0
	iconst_0	
	if_icmplt L0	; e1 < e2
510	iconst_0	
	goto L1	
	L0:	
	iconst_1	
	L1:	
515	ifeq L5	; e1 is false
	.line 12	
	bipush 84	
	getstatic java/lang/System/out	Ljava/io/PrintStream;
	swap	
	invokevirtual java/io/PrintStream/println (C)V	
520	.line 12	
	bipush 69	
	getstatic java/lang/System/out	Ljava/io/PrintStream;
	swap	
	invokevirtual java/io/PrintStream/println (C)V	
525	.line 12	
	bipush 95	
	getstatic java/lang/System/out	Ljava/io/PrintStream;
	swap	
	invokevirtual java/io/PrintStream/println (C)V	
530	.line 12	

535	bipush 86 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 12
540	bipush 82 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 12
545	bipush 79 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 12
550	bipush 69 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 12
555	bipush 71 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V goto L6 L5: .line 14

560	iload 0		; load jaar_1 from 0
	ldc 2099		; load constant maxyear_1
	if_icmpgt L2		; e1 > e2
	iconst_0		
	goto L3		
565	L2:		
	iconst_1		
	L3:		
	ifeq L4		; e1 is false
	.line 15		
	bipush 84		; ldc 'T'
570	getstatic java/lang/System/out		Ljava/io/PrintStream;
	swap		
	invokevirtual java/io/PrintStream/println (C)V		
	.line 15		
	bipush 69		; ldc 'E'
575	getstatic java/lang/System/out		Ljava/io/PrintStream;
	swap		
	invokevirtual java/io/PrintStream/println (C)V		
	.line 15		
	bipush 95		; ldc ' _ '
580	getstatic java/lang/System/out		Ljava/io/PrintStream;
	swap		
	invokevirtual java/io/PrintStream/println (C)V		
	.line 15		
	bipush 76		; ldc 'L'
585	getstatic java/lang/System/out		Ljava/io/PrintStream;
	swap		


```

    invokevirtual java/io/PrintStream/printLn(C)V
.line 15
    bipush 65                ; ldc 'A'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/printLn(C)V
.line 15
    bipush 65                ; ldc 'A'
    getstatic java/lang/System/out Ljava/io/PrintStream;
    swap
    invokevirtual java/io/PrintStream/printLn(C)V
.line 15
    bipush 84                ; ldc 'T'
    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_1 from 0
    iconst_0
    if_icmpge L7             ; e1 >= e2
    iconst_0
    goto L8
L7:
    iconst_1

```

```

L8:
.line 18
    iload 0
    ldc 2099
    if_icmple L9
    iconst_0
    goto L10
L9:
    iconst_1
L10:
    iand
    dup
    istore 1
    pop
    iload 1
    ireturn
    .end method
    .method public static nest(II)I
    .limit stack 5
    .limit locals 5
    .line 72
    .line 73
    .line 74
    iconst_0
    dup
    istore 2
    pop
    ; load jaar_1 from 0
    ; load constant maxyear_1
    ; e1 <= e2
    ; e1 right hand for assignment
    ; store e1 in ok_1
    ; load ok_1 from 1
    ; e1 right hand for assignment
    ; store e1 in returnvalue_1
    pop

```

645	.line 75				
	iload 0			; load stap_1 from 0	
	iconst_1				
	if_icmpeq L0			; e1 = e2	
	iconst_0				
	goto L1				
650	L0:				
	iconst_1				
	L1:				
	ifeq L9			; e1 is false	
655	.line 76				
	iconst_3				
	iload 1			; load w_1 from 1	
	imul			; e1 right hand for assignment	
	dup				
	istore 2			; store e1 in returnvalue_1	
	pop				
660	goto L10				
	L9:				
665	.line 78				
	iload 0			; load stap_1 from 0	
	iconst_2				
	if_icmpeq L2			; e1 = e2	
	iconst_0				
	goto L3				
670	L2:				
	iconst_1				
	L3:				

	ifeq L8	; e1 is false
.line 79		
675	iload 1	; load w_1 from 1
	iconst_4	
	idiv	; e1 right hand for assignment
	dup	
	istore 2	; store e1 in returnvalue_1
	pop	
680	.line 81	
	iconst_1	
	dup	; e1 right hand for assignment
	istore 3	; store e1 in loop_1
	pop	
685	.line 82	
	L6:	
	.line 82	
	iload 3	; load loop_1 from 3
	ldc 12	; load constant twalf_1
690	if_icmple L4	; e1 <= e2
	iconst_0	
	goto L5	
	L4:	
695	iconst_1	
	L5:	
	ifeq L7	
	.line 83	
	iload 2	; load returnvalue_1 from 2

700	iconst_1		
	isub		; e1 right hand for assignment
	dup		
	istore 2		; store e1 in returnvalue_1
	pop		
705	.line 84		
	iload 3		; load loop_1 from 3
	iconst_1		
	iadd		; e1 right hand for assignment
	dup		
710	istore 3		; store e1 in loop_1
	pop		
	goto L6		
	L7:		
	L8:		; e2 if true expression
715			; e3 if false expression
	L10:		
	iload 2		; load returnvalue_1 from 2
	ireturn		
720	.end method		
	.method public static copy(I)I		
	.limit stack 4		
	.limit locals 2		
	.line 107		
725	.line 108		
	bipush 42		
	pop		

730	iload 0		; load jaar_1 from 0
	ireturn		
	.end method		
	.method public static nested() I		
	.limit stack 5		
	.limit locals 2		
735	.line 120		
	.line 121		
	.line 122		
	getstatic Main/jaar_1 I		; load global jaar_1
	iconst_5		
	imul		
740	iconst_4		
	idiv		
	getstatic Main/X_1 I		; load global X_1
	isub		
	.line 122		
			; -
	bipush 10		
745	ineg		
	iadd		; e1 right hand for assignment
	dup		
	istore 0		; store e1 in Z_1
750	pop		
	.line 123		
	iload 0		; load Z_1 from 0
	invokestatic Main/copy(I) I		; e1 right hand for assignment
	dup		
755	istore 0		; store e1 in Z_1

	pop	
	iload 0	; load Z_1 from 0
	ireturn	
	.end method	
760	.method public static printdatum()V	
	.limit stack 8	
	.limit locals 2	
	.line 192	
	.line 193	
765	getstatic Main/day_1 I	; load global day_1
	dup	
	getstatic java/lang/System/outLjava/io/PrintStream;	
	swap	
	invokevirtual java/io/PrintStream/println(I)V	
770	pop	
	.line 195	
	bipush 95	; ldc ' _'
	dup	
	getstatic java/lang/System/outLjava/io/PrintStream;	
775	swap	
	invokevirtual java/io/PrintStream/println(C)V	
		; e1 right hand for assignment
	dup	
	istore 0	; store e1 in under_1
780	pop	
	.line 196	
	getstatic Main/month_1 I	; load global month_1
	iconst_3	

785	if_icmpeq L0 ; e1 = e2 iconst_0 goto L1 L0: iconst_1 L1: ifeq L2 ; e1 is false .line 197 bipush 77 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 197 bipush 97 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 197 bipush 97 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 197 bipush 114 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println (C)V .line 197
790	
795	
800	
805	
810	

815	<pre> bipush 116 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V ; e2 if true expression goto L3 L2: .line 199 bipush 65 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 199 bipush 112 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 199 bipush 114 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 199 bipush 105 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V .line 199 </pre>
820	
825	
830	
835	

840	<pre> bipush 108 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V ; e3 if false expression </pre>
845	<pre> L3: .line 201 iload 0 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(C)V getstatic Main/jaar_1 I ; load global jaar_1 getstatic java/lang/System/out Ljava/io/PrintStream; swap invokevirtual java/io/PrintStream/println(I)V return .end method </pre>
850	
855	