SELMA

Vonk, J
s0132778
Matenweg 75-201

Florisson, M
s000000
Box Calslaan xx-30

June 28, 2011

Contents

1 Inleiding

Korte beschrijving van de practicumopdracht.

2 Beknopte beschrijving

van de programmeertaal (maximaal een A4-tje).

3 Problemen en oplossingen

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

4 Syntax, context-beperkingen en semantiek

4.1 Lexer - terminals

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

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

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

UMIN;
UPLUS;
COMPOUND;
```

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

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

4.2 Basis

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 SEMICOLON!)+
  ;
```

4.3 Declaraties en types

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

4.3.1 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)+
        ;

type
        : INT
        | BOOL
        | CHAR
        ;
```

4.3.2 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.

4.3.3 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.

4.3.4 Voorbeeld

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

4.4 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.

4.4.1 Syntax

```
expression
        : expr_assignment
        ;

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

4.4.2 Context

expr_arithmetic moet een identifier worden, in het eind, aangezien dat het enige is waaraan je een waarde kunt toekennen

deze identiefier moet dan verwijzen naar een geldige variabele

het type van expression en expression_arithmetic moet hetzelfde zijn

expression is van het type van expr_assignment

expr_assignment is van het type van expr_arithmetic

4.4.3 Semantiek

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

4.4.4 Voorbeeld

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

4.5 Expressies - OR

4.5.1 Syntax

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

```
expr_arithmetic
        : expr_al1
        ;

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

4.5.2 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

4.5.3 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.

4.5.4 Voorbeeld

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

4.6 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.

4.6.1 Syntax

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

4.6.2 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

4.6.3 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.

4.6.4 Voorbeeld

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

4.7 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.

4.7.1 Syntax

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

4.7.2 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

4.7.3 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.

4.7.4 Voorbeeld

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

4.8 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.

4.8.1 Syntax

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

4.8.2 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

4.8.3 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.

4.8.4 Voorbeeld

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

4.9 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.

4.9.1 Syntax

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

4.9.2 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

4.9.3 Semantiek

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

4.9.4 Voorbeeld

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

4.10 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

4.10.1 Syntax

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

4.10.2 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

4.10.3 Semantiek

Bij UMIN zal de waarde van expr_7 op de stack worden gezet en de operatie worden aangeroepen om het teken om te flippen.
Bij UPLUS gebeurd er niks, behalve dat de waarde van expr_7 netjes op de stack komt.

Bij NOT zal de waarde van expr_7 worden geinverteerd, dus true wordt false en visa versa.

4.10.4 Voorbeeld

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

4.11 Expressies - toplevel

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

4.11.1 Syntax

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

4.11.2 Context

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

4.11.3 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.

4.11.4 Voorbeeld

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

4.12 Unsigned constants

4.12.1 Syntax

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.

```
unsignedConstant
        : boolval
        | charval
        | intval
        ;

intval
        : NUMBER
        ;

boolval
        : BOOLEAN
        ;

charval
        : CHARV
        ;
```

4.12.2 Context

unsignedconstant is van het type van de gegeven waarde

boolval is een boolean type

charval is een char

intval is een integer

4.12.3 Semantiek

De desbetreffende waarde wordt op de stack gezet.

4.12.4 Voorbeeld

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

4.13 Identifier

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

4.13.1 Syntax

```
identifier
        : ID
        ;
```

4.13.2 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.

4.13.3 Semantiek

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

4.13.4 Voorbeeld

```
Answer42;
```

4.14 Read

4.14.1 Syntax

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

4.14.2 Context

4.14.3 Semantiek

4.14.4 Voorbeeld

4.14.5 Syntax

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

4.14.6 Context

4.14.7 Semantiek

4.14.8 Voorbeeld

4.14.9 Syntax

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

4.14.10 Context

4.14.11 Semantiek

4.14.12 Voorbeeld

4.14.13 Syntax

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

4.14.14 Context

4.14.15 Semantiek

4.14.16 Voorbeeld

4.14.17 Closed compoundexpression

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

4.14.18 Context

4.14.19 Semantiek

4.14.20 Voorbeeld

4.14.21 Syntax

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

4.14.22 Context

4.14.23 Semantiek

4.14.24 Voorbeeld

5 Vertaalregels

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

6 Beschrijving van Java programmatuur

Beknopte bespreking van de extra Java klassen die u gedefinieerd heeft voor uw compiler (b.v. symbol table management, type checking, code generatie, error handling, etc.). Geef ook aan welke informatie in de AST-nodes opgeslagen wordt.

7 Testplan en -resultaten

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

8 Conclusies

9 Appendix

9.1 ANTLR Lexer & Parser specificatie

```
grammar SELMA;


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

tokens {
        COLON           = ':';
        SEMICOLON       = ';';
        LPAREN          = '(';
        RPAREN          = ')';
        LCURLY          = '{';
        RCURLY          = '}';
        COMMA           = ',';
        EQ              = '=';
        APOSTROPHE      = '\'';

        //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';

        PROC            = 'procedure';
        FUNC            = 'function';

        UMIN;
        UPLUS;

        BEGIN;
        END;
        COMPOUND;
        EXPRESSION_STATEMENT;

}


@header {
  package SELMA;
}

@lexer::header {
  package SELMA;
}

// Parser rules

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

compoundexpression
        : cmp -> ^(COMPOUND cmp)
        ;

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

//declaration

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

type
        : INT
        | BOOL
        | CHAR
        ;

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

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

```
        ;

130
expression
        : expr_assignment
        ;

135 expr_assignment
        : expr_arithmetic (BECOMES^ expression)?
        ;

expr_arithmetic
140     : expr_al1
        ;

        expr_al1                                                //expression
            arithmetic level 1
                : expr_al2 (OR^ expr_al2)*
145             ;

        expr_al2
                : expr_al3 (AND^ expr_al3)*
                ;
150
        expr_al3
                : expr_al4 ((RELS|RELSE|RELG|RELGE|RELE|RELNE)^ expr_al4
                   )*
                ;

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

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

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

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

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

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

expr_if
```

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

expr_while
        : WHILE^ compoundexpression DO compoundexpression OD
        ;

expr_closedcompound
        : LCURLY compoundexpression RCURLY
        ;

expr_closed
        : LPAREN! expression RPAREN!
        ;

unsignedConstant
        : boolval
        | charval
        | intval
        ;

intval
        : NUMBER
        ;

boolval
        : BOOLEAN
        ;

charval
        : CHARV
        ;

identifier
        : ID
        ;

CHARV
  : APOSTROPHE LETTER APOSTROPHE
  ;

BOOLEAN
        : TRUE
        | FALSE
        ;

ID
        : LETTER (LETTER | DIGIT)*
        ;

NUMBER
        : DIGIT+
        ;

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

fragment DIGIT
```

```
        : ('0'..'9')
        ;

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

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

fragment LETTER
  : LOWER
  | UPPER
  ;

fragment TRUE
        : 'true'
        ;

fragment FALSE
        : 'false'
        ;


//EOF
```

9.2 ANTLR Checker specificatie

```
tree grammar SELMAChecker;

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

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

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

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

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

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

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

   switch (type){
     case INT:
     st.enter($id,new CheckerEntry(SR_Type.INT,SR_Kind.VAR));
     break;
     case BOOL:
     st.enter($id,new CheckerEntry(SR_Type.BOOL,SR_Kind.VAR));
     break;
     case CHAR:
     st.enter($id,new CheckerEntry(SR_Type.CHAR,SR_Kind.VAR));
     break;
   }
   }
```

```
65          | ^(node=CONST type val id=ID)
             {
    int type = node.getChild(0).getType();
    int val  = node.getChild(1).getType();

70           switch (type){
      case INT:
      if (val!=NUMBER) throw new SELMAException(id,"Expecting int-value")
           ;
      st.enter($id,new CheckerEntry(SR_Type.INT,SR_Kind.CONST));
      break;
75    case BOOL:
      if (val!=BOOLEAN) throw new SELMAException(id,"Expecting bool-value
           ");
      st.enter($id,new CheckerEntry(SR_Type.BOOL,SR_Kind.CONST));
      break;
      case CHAR:
80    if (val!=CHARV) throw new SELMAException(id,"Expecting char-value")
           ;
      st.enter($id,new CheckerEntry(SR_Type.CHAR,SR_Kind.CONST));
      break;
           }
           }
85        ;

type
  : INT
  | BOOL
90  | CHAR
  ;

val
  : NUMBER
95  | CHARV
  | BOOLEAN
  ;

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

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

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

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

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

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

```
        | ^(node=(OR|AND) expression expression)
{
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;

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=(RELE|RELNE) expression expression)
{
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;

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=(UPLUS|UMIN) expression)
{
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;

$node.SR_kind=e1.SR_kind;
}

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

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

$node.SR_kind=e1.SR_kind;
}

        | ^(node=IF {st.openScope();} compoundexpression
            THEN {st.openScope();} compoundexpression {st.closeScope()
                ;}
            (ELSE {st.openScope();} compoundexpression {st.closeScope()
                ;})?
          {st.closeScope();})
{
SELMATree e1 = (SELMATree)node.getChild(0);
SELMATree e2 = (SELMATree)node.getChild(2);
SELMATree e3 = (SELMATree)node.getChild(4);

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

 if (e3==null) { //no else
```

```
195         $node.SR_type=SR_Type.VOID;
            $node.SR_kind=null;
          } else { // there is a else
            if (e2.SR_type==e3.SR_type) {
              $node.SR_type=e3.SR_type;
200                           if (e2.SR_kind==SR_Kind.CONST && e3.SR_kind==
                                  SR_Kind.CONST)
                               $node.SR_kind=SR_Kind.CONST;
                              else
                               $node.SR_kind=SR_Kind.VAR;
            } else {
205           $node.SR_type=SR_Type.VOID;
              $node.SR_kind=null;
            }
          }
          }
210
              | ^(node=WHILE {st.openScope();}compoundexpression
                   DO {st.openScope();} compoundexpression {st.closeScope();}
                  OD{st.closeScope();})
        {
215     SELMATree e1 = (SELMATree)node.getChild(0);
        SELMATree e2 = (SELMATree)node.getChild(2);

        if (e1.SR_type!=SR_Type.BOOL)
         throw new SELMAException(e1,"Expression must be boolean");
220
        $node.SR_type=SR_Type.VOID;
        $node.SR_kind=null;

        }
225
              | ^(node=READ (id=ID
                 {
                   if (st.retrieve($id).kind!=SR_Kind.VAR)
                     throw new SELMAException($id,"Must be a variable");
230                })+
                   {
                     if ($node.getChildCount()==1){
                      $node.SR_type=((SELMATree)node.getChild(0)).SR_type;
                   $node.SR_kind=SR_Kind.VAR;
235             } else {
                  $node.SR_type=SR_Type.VOID;
                  $node.SR_kind=null;
              }
                    }
240                )

              | ^(node=PRINT expression+
                {
         for (int i=0; i<((SELMATree)node).getChildCount(); i++){
245        if (((SELMATree)node.getChild(i)).SR_type==SR_Type.VOID)
             throw new SELMAException($node,"Can not be of type void");
         }
             if ($node.getChildCount()==1){
                $node.SR_type=((SELMATree)node.getChild(0)).SR_type;
250             $node.SR_kind=SR_Kind.VAR;
              } else {
                $node.SR_type=SR_Type.VOID;
                $node.SR_kind=null;
              }
255             })

              | ^(node=BECOMES expression expression)
                {
        SELMATree e1 = (SELMATree)node.getChild(0);
260     SELMATree e2 = (SELMATree)node.getChild(1);
        if (e1.getType()!=ID)
```

```
     throw new SELMAException(e1,"Must be a identifier");

    CheckerEntry ident = st.retrieve(e1);
265
    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);
270
    $node.SR_type=ident.type;
    $node.SR_kind=SR_Kind.VAR;
           }

275       | LCURLY {st.openScope();} compoundexpression {st.closeScope();}
              RCURLY

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

          | node=BOOLEAN
    {
285 $node.SR_type=SR_Type.BOOL;
    $node.SR_kind=SR_Kind.CONST;
    }

          | node=LETTER
290 {
    $node.SR_type=SR_Type.CHAR;
    $node.SR_kind=SR_Kind.CONST;
    }

295       | node=ID
           {
           CheckerEntry entry = st.retrieve($node);
           $node.SR_type=entry.type;
           $node.SR_kind=entry.kind;
300        }
          ;
```

9.3 ANTLR Codegenerator specificatie

```
tree grammar SELMACompiler;

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

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

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

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

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

program
  : ^(node=BEGIN {st.openScope();} compoundexpression {st.closeScope();}
      END)
  -> program(instructions={$compoundexpression.st},
             source_file={SELMA.inputFilename},
             stack_limit={maxStackDepth},
             locals_limit={$node.localsCount})
  ;

compoundexpression
  : ^(node=COMPOUND (s+=declaration  | s+=expression)+)
  -> compound(instructions={$s})
  ;

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

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

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

  // store the const at a address? LOAD Or just copy LOADL?
  | ^(node=CONST INT val=NUMBER (id=ID)+)
```

```
    {st.enter($id,new CompilerEntry(SR_Type.INT,SR_Kind.CONST,st.nextAddr
        ())); }
    -> declareConst(id={$id.text},value={$val.text},type={"INT"},addr={st.
        nextAddr()-1})

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

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

expression_statement
    : ^(EXPRESSION_STATEMENT e1=expression) { curStackDepth--; }
    -> popStack()
    ;

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

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

    | ^(MOD e1=expression e2=expression) { curStackDepth--; }
    -> biExpr(e1={$e1.st},e2={$e2.st},instr={"imod"})

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

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

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

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

    | ^(RELS e1=expression e2=expression) { curStackDepth--; }
    -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmplt"})

    | ^(RELSE e1=expression e2=expression) { curStackDepth--; }
    -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmple"})

    | ^(RELG e1=expression e2=expression) { curStackDepth--; }
    -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpgt"})

    | ^(RELGE e1=expression e2=expression) { curStackDepth--; }
    -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpge"})

    | ^(RELE e1=expression e2=expression) { curStackDepth--; }
    -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpeq"})

    | ^(RELNE e1=expression e2=expression) { curStackDepth--; }
    -> biExprJump(e1={$e1.st},e2={$e2.st},instr={"if_icmpne"})

//single arg expression
    | ^(UPLUS e1=expression)
    {$st=$e1.st;}
```

```
  | ^(UMIN e1=expression)
  -> Expr(e1={$e1.st},op={"neg"})

  | ^(NOT e1=expression)
  -> Expr(e1={$e1.st},op={"not"})

//CONDITIONAL
  | ^(IF ec1=compoundexpression THEN ec2=compoundexpression (ELSE ec3=
      compoundexpression)?)
  -> if(ec1={$ec1.st},ec2={$ec2.st},ec3={$ec3.st})
  | ^(WHILE ec1=compoundexpression DO ec2=compoundexpression OD) {
      curStackDepth--; }
  -> while(ec1={$ec1.st},ec2={$ec2.st})

//IO
  | ^(READ id=ID)
       -> read(id={$id.text}, addr={st.retrieve($id).addr}, dup_top={
           TRUE})

  | ^(READ id1=ID (id2=ID)+) { curStackDepth--; }
       -> read(id={$id1.text}, dup_top={FALSE})

  | ^(PRINT expression+)

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

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

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

  | node=BOOLEAN { incrStackDepth(); }
    -> loadNum(val={($node.type==TRUE)?1:0}, iconst={TRUE})

  | node=LETTER { incrStackDepth(); }
    -> loadNum(val={Character.getNumericValue($node.text.charAt(1))},
        iconst={FALSE}, bipush={TRUE})

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

9.4 ANTLR Codegenerator Stringtemplate specificatie

```
//SELMA string template

group SELMA;

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

compound(instructions) ::= <<
                                                                    ; start compound
    <instructions; separator="\n">
                                                                    ; end compound
>>

popStack() ::= <<
    pop
>>

//Calculations
Expr(e1,op)::=<<
    <e1>                                                    ; e1 for operation <op>
    CALL <op>                                          ; single operation <op>
>>

biExpr(e1,e2,instr)::=<<
    <e1>                                                    ; e1 for operation <instr>
    <e2>                                                    ; e2 for operation <instr>
    <instr>
>>

biExprJump(e1, e2, instr)::=<<
    <e1>
    <e2>
```

```
    <instr> L1
    iconst_0
L1:
    iconst_1
>>

//Declare
declareConst(id,val,type,addr)::=<<
                                                                    ; declare var <id>: <type> = <val> @ <addr
                                                                       >[SB]
    ldc <val>                       ; declare var
>>

declareVar(id,type,addr)::=<<
                                                                    ; declare var <id>: <type> @ <addr>[SB]
    PUSH 1
>>

//Load

loadNum(val, num)::=<<
    <if (iconst)>
        iconst_<val>
    <elseif (bipush)>
        bipush <val>
    <else>
        ldc <val>
    <endif>
>>

loadVal(id,addr)::=<<
                                                                    ; loadVal <id> from <addr>[SB]
    iload <addr>                    ; load integer from <addr>
>>

//Assign
assign(id, type, addr, e1, isint)::=<<
    <e1>                            ; e1 right hand for assignment
    dup
    istore <addr>                   ; assign e1 to <id>: <type> @ <addr>[SB]
>>
```

```
read(id, addr, dup_top) ::= <<
    read here                          ; reading <id>
<if (dup_top)>
    dup
<endif>
    istore <addr>
>>

//conditionals
if(ec1, ec2, ec3)::=<<
    <ec1>                                                  ; e1 condition
    iconst_0
    ifeq L1                            ; e1 is false
    <ec2>                                                  ; e2 if true expression
    goto L2
L1:
    <ec3>                                                  ; e3 if false expression
L2:
>>

while(ec1,ec2)::=<<
L1:
    <ec1>                                                  ; e1 while condition
    iconst_0
    ifeq L2
    <ec2>                                                  ; e2 expression to evaluate (body)
    goto L1
L2:
>>
```

9.5 Invoer- en uitvoer van een uitgebreid testprogramma

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