Vertalerbouw - Huiswerk serie 1

Naam: M. Wienk
Studentnummer: 1238531
Studentassistent: Edwin Smulders

Opgave 1

Syntax

De syntax van de for expressie kan als volgt worden weergegeven in EBNF (extensie van de production rules in example 1.3).

single-Command ::= **for** *Identifier* : **Integer upto** *Integer-Literal* **yield** *Expression*

Contextbeperkingen

In de bovenstaande syntax geldt de volgende scope rule:

1. De *Identifier* kan alleen binnen de *Expression* gebruikt worden.

In de bovenstaande syntax gelden de volgende type rules:

1. *Identifier* moet een Integer type zijn.
2. *Integer-Literal* moet een Integer type zijn.

Semantiek

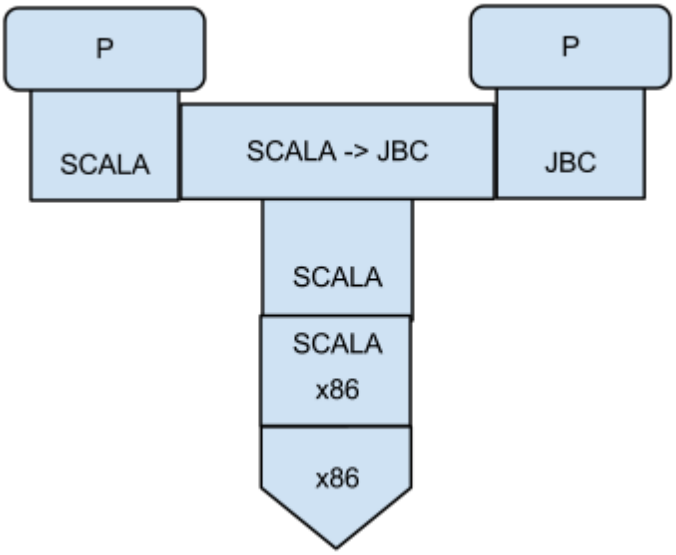
Het bovenstaande commando leidt tot de volgende actie:
Zolang *Identifier* lager is dan *Integer-Literal* wordt

1. Eerst de *Expression* uitgevoerd en
2. dan de *Identifier* met 1 verhoogd.

Identifier kan binnen de *Expression* de iteratie van de loop bijhouden.

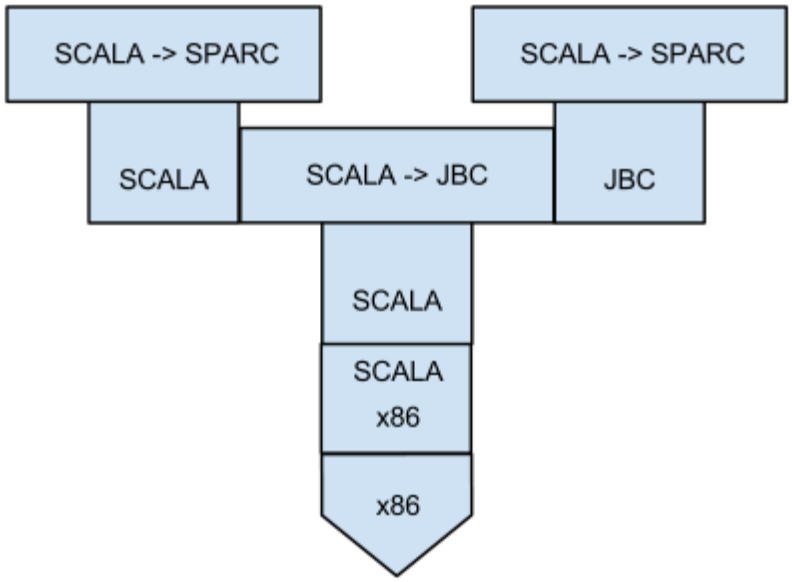
Opgave 2

a.

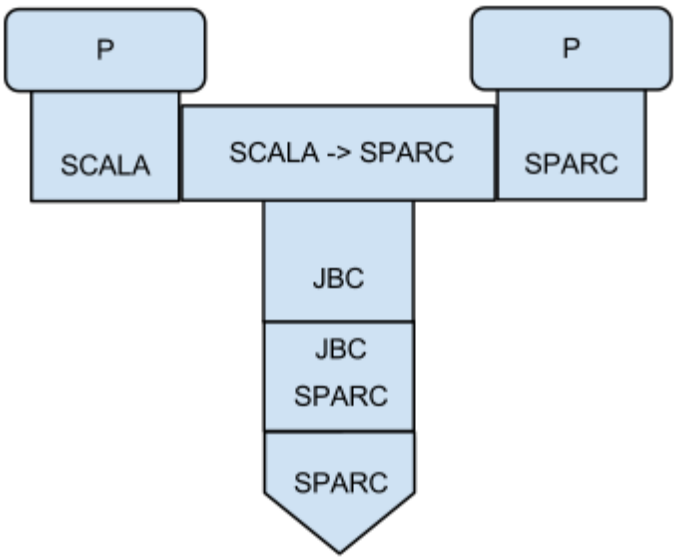


Een programma in Scala geschreven wordt omgezet via een x86 machine naar JBC code, die uitgevoerd kan worden in een virtuele machine (op een SPARC machine).

b.

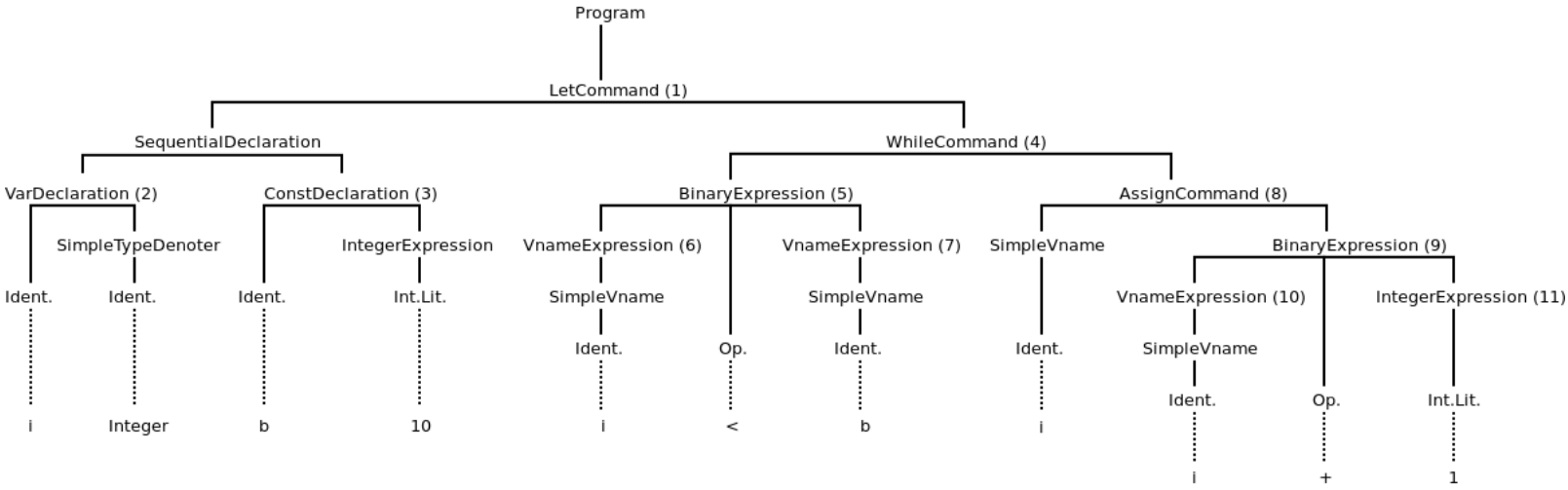


Stap 1: een nieuw geschreven Scala naar SPARC vertaler, geschreven in Scala, wordt vertaald naar een Scala naar SPARC vertaler, geschreven in JBC (met gebruik van de bestaande Scala naar JBC vertaler op de x86 machine.



Stap 2: een in Scala geschreven programma kan worden omgezet in een SPARC programma door middel van de gegenereerde vertaler in JBC. De vertaling maakt gebruik van de virtuele machine op de SPARC machine. Het gegenereerde programma kan op de SPARC machine gedraaid worden zonder tussenkomst van interpreters.

Opgave 3



AST weergave van het programma (a.d.h.v. Example 1.5 grammar)

Syntactische analyse

1. Het programma is een LetCommand, bestaande uit een deel voor de declaraties van variabelen (en constanten) en een deel waarin commando's worden uitgevoerd.
2. De variabele declaratie bestaat uit een identifier (i) en een type notatie (Integer).
3. De constate declaratie bestaat uit een identifier (b) en een type notatie (Integer-Literal).
4. Het while commando bestaat uit een expressie die wordt geëvalueerd voordat het commando (assignCommand) wordt uitgevoerd.

5. De binaryExpression heeft een logische operator. Er wordt dus een boolean waarde teruggegeven. De expressie evalueert de operanden aan de hand van de operator (<).
6. De variabele i wordt als eerste operand gebruikt in de expressie.
7. De constante b wordt als tweede operand gebruikt in de expressie.
8. Wanneer de binaryExpression (5) true teruggeeft wordt de assignCommand uitgevoerd. Dit commando heeft als operand een identifier (i) en een expressie.
9. De binaryExpression heeft als operator '+' en geeft daardoor een integer waarde terug (de som van de twee operanden).
10. De eerste operand is de identifier i.
11. De tweede operand is de waarde 1.

Contextuele analyse

De contextuele analysator trekt de volgende conclusies:

2. i is gedeclareerd als type integer
3. b is gedeclareerd als type integer
6. & 7. De declaratie van de gevonden identifier wordt bekeken. De link tussen de identifier en de declaratie wordt gemaakt en er geconcludeerd dat de identifier een integer betreft.
5. De operanden kloppen bij de gebruikte operator, er wordt geconcludeerd dat er een boolean waarde wordt teruggegeven.
8. ,10 & 11. De declaratie van de gevonden identifier wordt bekeken. De link tussen de identifier en de declaratie wordt gemaakt en er geconcludeerd dat de identifier een integer betreft.
9. De operanden van de '+' operator kloppen, er wordt geconcludeerd dat er een integer waarde teruggegeven wordt.
8. De operanden van het assignCommand kloppen (integer, integer). Het commando zelf geeft niets terug.
4. Het while commando bestaat uit de juiste onderdelen (boolean en een commando).
5. Het LetCommand bestaat uit de juiste onderdelen (Command's).

TAM code generatie

De gegeneerde code bevat labels 0-11 (in de opdracht), de waarde tussen haakjes is het betreffende label. De waarde tussen vierkante haakjes ([x]) is een verwijzing naar een stap in de Triangle code.

2. Er wordt ruimte gemaakt voor een variabele (0)
6. De waarde van de variabele wordt geladen (6)
7. De integer literal waarde wordt geladen, waarde 10 (7)
5. De twee waardes worden met een lt functie met elkaar vergeleken (8). Als de lt functie een true waarde oplevert, wordt de assignCommand van [8] uitgevoerd (9).
10. De waarde van i wordt geladen (2), de letterlijke waarde 1 wordt geladen [11] (3). Deze twee waarden worden bij elkaar opgeteld (4) [9].
8. De berekende waarde wordt opgeslagen in het geheugen op de plek van i [8] (5). Daarna wordt er teruggesprongen naar [6] in de Triangle code en verder naar (6) in de TAM code.
4. Wanneer stap (8) niet meer evalueert naar true wordt het gereserveerde geheugen vrijgemaakt (10) en het programma gestopt (11).

Opgave 4

a.
$S \rightarrow aSbS \mid \varepsilon$
first[S] = {a, ε}
follow[S] = {a, b, ε}
lookahead[S ::== aSbS] = {a}
lookahead[S ::== ε] = {b}
Aan de hand van het eerstvolgende teken kan bepaald worden welke regel gebruikt moet worden. De grammatica is dus LL(1).
b.
$S \rightarrow FaSb \mid \varepsilon$
$F \rightarrow S \mid \varepsilon$
first[F] = {a}
first[S] = {a}
follow[F] = {a,b}
follow[S] = {a,b}
lookahead[F ::== S] = {a}
lookahead[F ::== ε] = {b}
lookahead[S ::== FaSb] = {a}
lookahead[S ::== ε] = {b}
Aan het eerstvolgende teken is niet te zien of deze in de F of in de S regel valt. De grammatica is dus niet LL(1).
c.
$S \rightarrow IT\,E \mid \varepsilon$
$I \rightarrow iS$
$T \rightarrow tS$
$E \rightarrow eS \mid \varepsilon$
first[E] = {e,ε}
first[T] = {t}
first[I] = {i}
first[S] = {i, ε}
follow[E] = {i, ε}
follow[T] = {i, ε}
follow[I] = {i, ε}
lookahead[E ::== eS] {e}
lookahead[E ::== ε] {ε}
lookahead[T ::== tS] {t}
lookahead[S ::== ITE] {i}
lookahead[S ::== ε] {i,t,e,ε}
De grammatica is niet LL(1), aangezien er niet bepaald kan worden welke if bij welke else hoort (de lookahead [S ::==ε] heeft alle mogelijkheden).

Opgave 5

a. De AST nodes worden gerepresenteerd door integers. De input wordt gescand en opgedeeld in tokens String.match() operatie. Een switch statement bepaalt later hoe de tree opgebouwd wordt uit de tokens. Een alternatieve representatie kan werken met een HashMap,

waarin de keys de tokens zijn en de values de AST nodes. Het voordeel is dat een simpele HashMap.get(token) de node teruggeeft. Een nadeel ervan is dat er wel eerst een HashMap gevuld dient te worden en dat deze in het geheugen opgeslagen staat (waar de ANTLER implementatie alles in de code zelf doet).
b.

```
public class SymbolTable<Entry extends IdEntry> {
	protected Stack<HashMap<String, Entry>> symbolMapList;
	public SymbolTable() {
		symbolMapList = new Stack<HashMap<String, Entry>>();
	}
	public void openScope() {
		symbolMapList.push(new HashMap<String, Entry>());
	}
	public void closeScope() {
		symbolMapList.pop();
	}
	public int currentLevel() {
		return symbolMapList.size() - 1;
	}
	public void enter(String id, Entry entry) throws SymbolTableException {
		if (this.currentLevel() > -1 && !
	symbolMapList.peek().containsKey(id)) {
			entry.setLevel(this.currentLevel());
			symbolMapList.peek().put(id, entry);
		} else {
			throw new SymbolTableException("");
		}
	}
	public Entry retrieve(String id) {
		for (int i = this.currentLevel(); i > -1; i--) {
			HashMap<String, Entry> tempHM = symbolMapList.get(i);
			if (tempHM.containsKey(id)) return tempHM.get(id);
		}
		return null;
	}
}
```

Aangezien de standaard Stack functionaliteit wordt gebruikt, wordt er minder gebruik gemaakt van het geheugen. De structuur van de Stack houdt het level bij, wat dus ruimte bespaart omdat die informatie niet meer in een aparte lijst hoeft te komen.

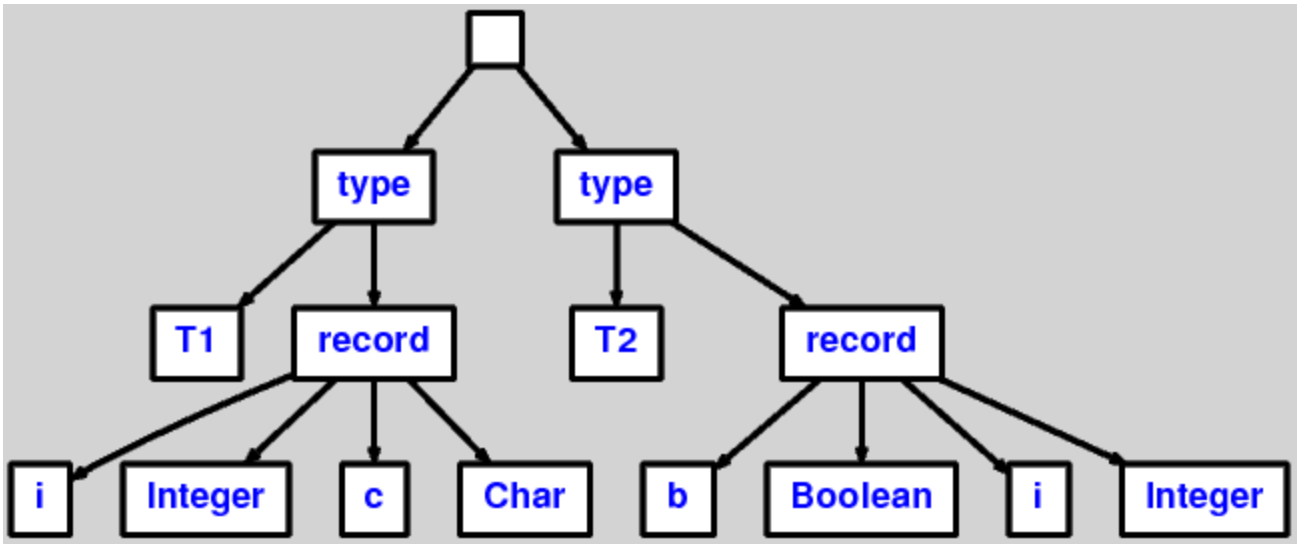
Opdracht 6

a.

```
grammar Opdracht6;

options {
  language = Java;
  output = AST;
  k = 1;
}

tokens {
  TYPE      = 'type'    ;
  RECORD    = 'record'  ;
```

```
  END       = 'end'     ;

  SEMICOLON = ';'       ;
  COLON     = ':'       ;
  COMMA     = ','       ;
  TILDE     = '~'       ;

  //Types
  INTEGER   = 'Integer' ;
  CHAR      = 'Char'    ;
  BOOLEAN   = 'Boolean' ;
}

@lexer::header {
package vb.homework1;
}
@header {
package vb.homework1;
}

declarations
      :   typeDenoter (SEMICOLON! typeDenoter)* ;

typeDenoter
      :   TYPE^ IDENTIFIER TILDE! recordTypeDenoter ;

recordTypeDenoter
      :   RECORD^ singleRecordAggregate multipleRecordAggregate END! ;

multipleRecordAggregate
      :   (COMMA! IDENTIFIER COLON! type)* ;

singleRecordAggregate
      :   IDENTIFIER COLON! type ;

type
      :   INTEGER | CHAR | BOOLEAN  ;

// Lexer rules
WS
      :   (' ' | '\t' | '\f' | '\r' | '\n')+    { $channel=HIDDEN; }    ;
IDENTIFIER
      :   LETTER (LETTER | DIGIT)*  ;

fragment DIGIT  :   ('0'..'9') ;
fragment LOWER  :   ('a'..'z') ;
fragment UPPER  :   ('A'..'Z') ;
fragment LETTER :   LOWER | UPPER ;
```

b.


type
type
T1
record
T2
record
i
Integer
c
Char
b
Boolean
i
Integer