НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Лабораторна робота № 1

ПРОЄКТУВАННЯ ТА СПЕЦИФІКАЦІЯ
ПРОСТОЇ ІМПЕРАТИВНОЇ МОВИ ПРОГРАМУВАННЯ
ЗАГАЛЬНОГО ПРИЗНАЧЕННЯ
(з дисципліни «Побудова компіляторів»)

Учасники проєкту:
Пашковський Євгеній Сергійович
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Звіт студента І курсу, групи ІП-41мн
спеціальності 121 «Інженерія програмного забезпечення»
Пашковського Євгенія Сергійовича _ _ _ _ _ _ _ _
(Прізвище, ім'я, по батькові) (Підпис)

Перевірив: доцент, к.т.н. Стативка Ю.І. _ _ _ _ _ _ _ _
(Посада, науковий ступінь, прізвище та ініціали) (Підпис)

Київ – 2024

Зміст

Специфікація мови програмування KotlinScript

1. Вступ

Представлена тут мова програмування KotlinScript – імперативна мова загального призначення. Прототип синтаксичних конструкцій – мова Kotlin. Назва промовляється як ”котлін скрипт”. Назва асоціюється з мовою Kotlin, яка виступає прототипом синтаксичних конструкцій для цієї мови.

1.1. Обробка

Програма, написана мовою KotlinScript, подається на вхід інтерпретатора для трансляції до цільової форми (мови). Результат трансляції виконується у системі часу виконання (run-time system), для чого приймає вхідні дані та надає результат виконання програми.

Трансляція передбачає фази лексичного, синтаксичного та семантичного аналізу, а також фазу генерації коду. Фаза лексичного аналізу здійснюється окремим проходом.

1.2. Нотація

Для опису мови KotlinScript використовується розширена нотація (форма) Бекуса-Наура, див. табл. 1.

Ланцюжки, що починаються з великої літери вважаються нетерміналами (нетермінальними символами). Термінали (термінальні символи) — ланцюжки, що починаються з маленької літери, або ланцюжки, що знаходяться між одинарними чи подвійними лапками. Для графічного представлення граматики використовуються синтаксичні діаграми Вірта.

Метасимвол	Значення
=	визначається як
|	альтернатива
[x]	0 або 1 екземпляр x
{ x }	0 або більше екземплярів x
(x | y)	групування: будь -який з x або y
Abc	нетермінал
abc	термінал
'1'	термінал (не літера)
”1”	термінал (не літера)
”+”	термінал (не літера)

Таблиця 1 – Прийнята нотація РБНФ

2. Лексична структура

Лексичний аналіз виконується окремим проходом, отже не залежить від синтаксичних та семантичних аспектів. Лексичний аналізатор розбиває текст програми на лексеми.

2.1. Алфавіт

Текст програми мовою KotlinScript може містити тільки такі символи (characters): літери, цифри та спеціальні знаки.

```
LowerLetter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k'
| 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
| 'x' | 'y' | 'z';
UpperLetter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K'
| 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W'
| 'X' | 'Y' | 'Z';
Letter = UpperLetter | LowerLetter;
NonZeroDigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Digit = '0' | NonZeroDigit;
SpecSign = '.' | ',' | ':' | ';' | '(' | ')'
| '=' | '+' | '-' | '*' | '/' | '<' | '>' | '!'
| WhiteSpace | EOL;
WhiteSpace = ' ' | '\t';
EOL = ['\r'],'\n';
```

2.2. Спеціальні символи

У програмі мовою KotlinScript використовуються лексеми, що класифікуються як спеціальні символи, ідентифікатори, беззнакові цілі константи, беззнакові дійсні константи, логічні константи та ключові слова.

```
SpecSymbols = ArithmOps | RelOps | BracketsOps | AssignOp | Punct;
ArithmOps = AddOps | MultOps;
AddOps = '+' | '-';
MultOps = '*' | '/' | '^';
RelOps = '===' | '==' | '<= ' | '<' | '>' | '>=' | '!=';
BracketsOps = '(' | ')';
AssignOp = '=';
```

```
Punct = '.' | ',' | ':' | ';';
InlineCommentOp = '//';
```

2.3. Ідентифікатори

У мові KotlinScript ідентифікатори використовуються для позначення змінних, функцій, пакетів тощо.

Першим символом ідентифікатора може бути тільки літери, наступні символи, якщо вони є, можуть бути цифрами або літерами, або у формі правила:

```
Id = Letter, { Letter | Digit };
```

Довжина ідентифікатора не обмежена.

Жоден ідентифікатор не може збігатись із ключовим (вбудованим, зарезервованим) словом або словами true, false. Якщо у фазі лексичного аналізу визначена лексема, яка синтаксично могла би бути ідентифікатором, але збігається із ключовим словом, то вона вважається ключовим словом. Якщо збігається з словами true чи false, то вважається значенням логічної константи.

Діаграма Вірта ідентифікатора представлена на рис. 2.1.

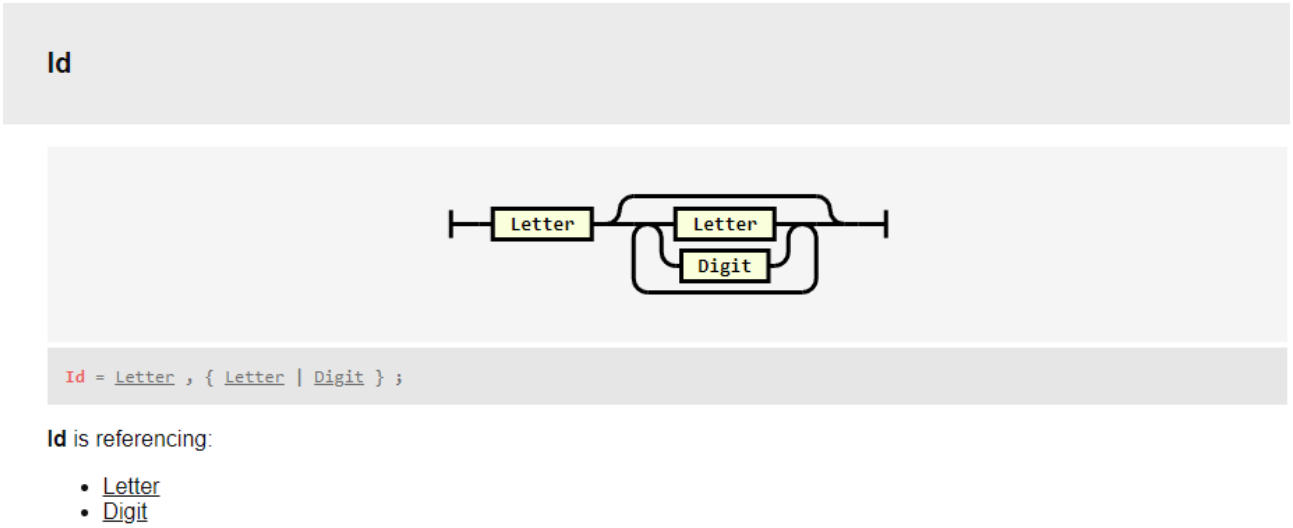


Рисунок 2.1 – Синтаксична діаграма ідентифікатора

Приклади:

b, o3, var11

2.4. Константи

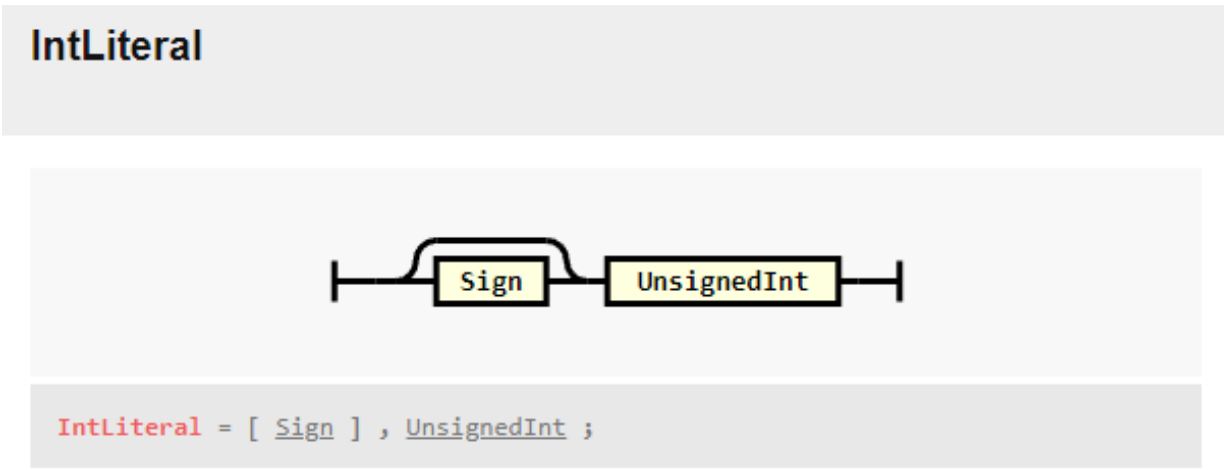
Літерали використовуються для представлення постійних значень: цілих числових констант IntLiteral, дійсних числових констант RealLiteral та логічних констант BoolLiteral.

Їх синтаксис:

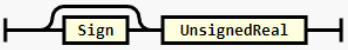
```
Literal = IntLiteral | RealLiteral | BoolLiteral;
IntLiteral = [Sign], UnsignedInt;
RealLiteral = [Sign], UnsignedReal;
Sign = '+' | '-';
UnsignedInt = Digit, {Digit};
UnsignedReal = ('.', UnsignedInt)
| (UnsignedInt, '.')
| (UnsignedInt, '.', UnsignedInt);
BoolLiteral = 'true' | 'false';
```

Тип кожної константи визначається її формою, а значення має знаходитись у діапазоні репрезентативних значень для її типу.

На етапі лексичного аналізу виявляються тільки беззнакові цілі константи UnsignedInt, беззнакові дійсні константи UnsignedReal та логічні константи BoolLiteral. Унарні мінус чи плюс розпізнаються при синтаксичному розборі. Синтаксичні діаграми констант представлені на рис. 2.2.

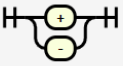


RealLiteral




```
RealLiteral = [ Sign ] , UnsignedReal ;
```

Sign




```
Sign = "+" | "-" ;
```

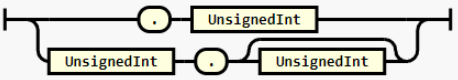
Items referencing **Sign**:

- IntLiteral
- RealLiteral
- ArithmExpression

UnsignedInt

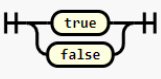

```
UnsignedInt = { Digit }- ;
```

UnsignedReal




```
UnsignedReal
  = "." , UnsignedInt
  | UnsignedInt , "."
  | UnsignedInt , "." , UnsignedInt
  ;
```

BoolLiteral




```
BoolLiteral = "true" | "false" ;
```

Рисунок 2.2 – Константи

Приклади беззнакових числових констант:

12, 234, 1.54, 34.567, 23., true, false

2.5. Ключові слова

Мова KotlinScript містить такі ключові слова:

fun, var, while, if, else, Int, Float, package

2.6. Токени

З потоку символів вхідної програми на етапі лексичного аналізу виокремлюються послідовності символів з певним сукупним значенням – токени. Список токенів мови KotlinScript див. у табл. 2.1.

Єдиний приклад засвідчує унікальність лексеми. Текст "символ ...", наприклад "символ for", означає, що вказаний символ (symbol) є терміналом.

Код	Приклади лексем	Токен	Неформальний опис
1	a, r1, b1s1	id	ідентифікатор
2	123, 0, 521	int_literal	ціле без знака
3	.012, 34.76, 876.	real_literal	дійсне без знака
4	true	bool_literal	логічне значення
5	false	bool_literal	логічне значення
6	fun	keyword	символ fun
7	var	keyword	символ var
8	while	keyword	символ while

9	if	keyword	символ if
10	else	keyword	символ else
11	Int	keyword	символ Int
12	Float	keyword	символ Float
13	package	keyword	символ package
14	=	assign_op	символ =
15	+	add_op	символ +
16	-	add_op	символ -
17	*	mult_op	символ *
18	/	mult_op	символ /
19	^	mult_op	символ ^
20	<	rel_op	символ <
21	<=	rel_op	символ <=
22	==	rel_op	символ ==
23	===	rel_op	символ ===
24	>=	rel_op	символ >=
25	!=	rel_op	символ !=
26	(	brackets_op	символ (
27	)	brackets_op	символ)
28	.	punct	символ .
29	,	punct	символ ,
30	:	punct	символ :
31	;	punct	символ ;
32	\32	ws	пробільні символи (пробіл)
33	\t	ws	пробільні символи (г. табуляція)

34	\n	eol	роздільник рядків (новий рядок)
35	\r\n	eol	роздільник рядків

Таблиця 2 – Таблиця лексем мови KotlinScript

3. Типи даних і змінні

Значення у мові KotlinScript можуть бути представлені або литералами (константами), або значеннями змінних, представлених у тексті програми своїми ідентифікаторами.

3.1. Типи даних

Мова KotlinScript підтримує значення трьох скалярних типів: Int, Float та Bool.

1. Цілий тип Int може бути представлений оголошеною змінною типу Int, або константою IntLiteral. Діапазон значень визначається реалізацією мови (4 байти).

2. Дійсний тип Float може бути представлений оголошеною змінною типу Float, або константою RealLiteral. Діапазон значень визначається реалізацією мови (8 байти).

3. Логічний тип Bool може бути представлений оголошеною змінною типу Bool, або константою BoolLiteral (true або false).

Прийнято, що false < true, false == 0, true == 1.

Інших типів чи механізмів конструювання типів у мові KotlinScript не передбачено.

При обчисленні виразів, за необхідності, виконується неявне приведення числових типів, див. розд. 5.2.1.

3.2. Змінні

Тип кожної змінної має бути визначений у розділі оголошень, див. розд. 4.

Використання не оголошеної змінної, викликає помилку на етапі трансляції.

Використання змінної, що не набула значення, викликає помилку. Змінна набуває значення в інструкціях (statements) присвоювання AssignStatement та/або введення InStatement.

4. Оголошення

Оскільки мова KotlinScript не передбачає конструювання типів чи структур даних, то оголошення стосуються тільки змінних.

Оголошення (декларація) специфікує набір ідентифікаторів, які можуть бути використані у програмі.

Синтаксис розділу оголошень визначається такими правилами:

```
Type = 'Int' | 'Float' | 'Bool';
VarTypeSpecifier = ':', Type;

VarDeclaration = Id, [VarTypeSpecifier];
VarDeclarationsList = VarDeclaration, { ',', VarDeclaration };
```

Кожен ідентифікатор має бути оголошений і тільки один раз. Отже, повторне оголошеної змінної та використання неоголошеної змінної викликають помилку на етапі трансляції.

Значення оголошеної змінної залишається невизначеним аж до присвоєння їй значення у інструкції присвоєння або введення. Використання змінної, що не набула значення, викликає помилку.

Область видимості змінної (scope) – вся програма.

Діаграми Вірта розділу декларації див. на рис. 4.1.

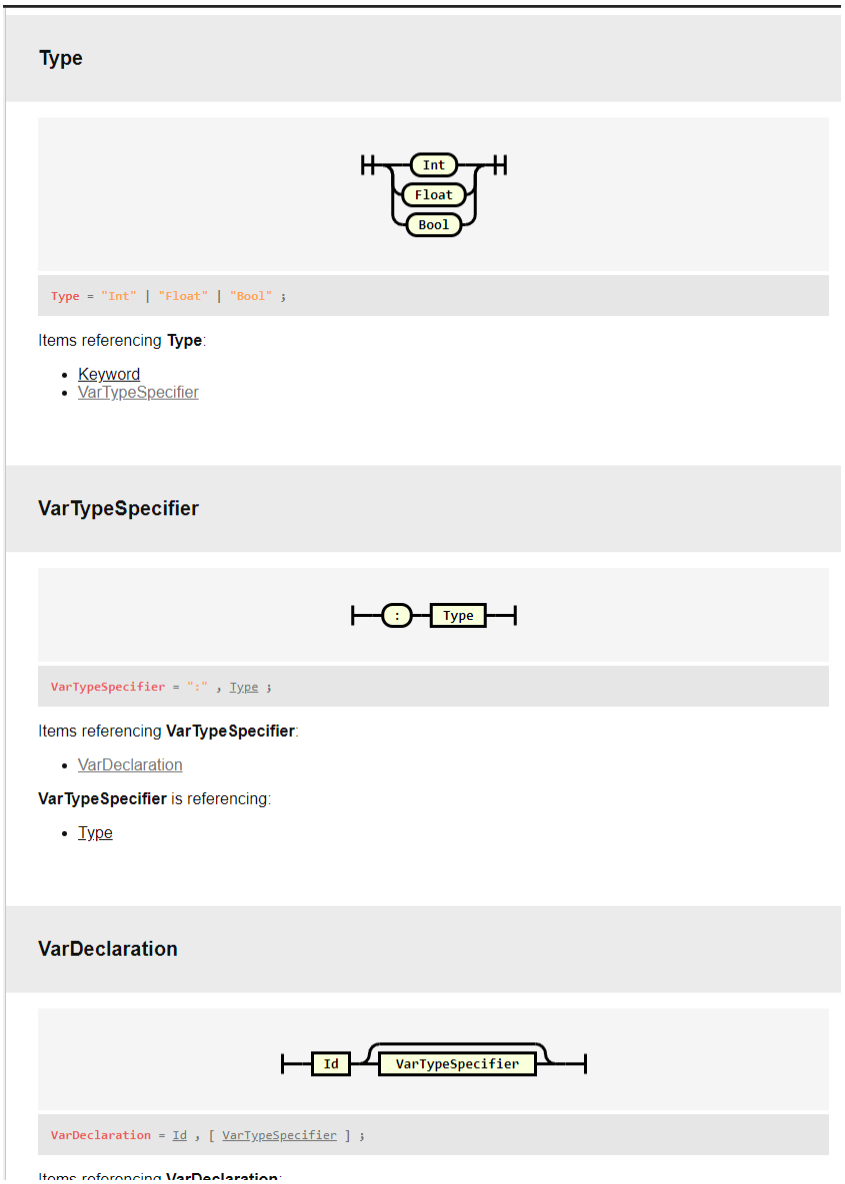


Рисунок 4.1 – Синтаксичні діаграми декларації

5. Вирази

Вираз — це послідовність операторів і операндів, що визначає порядок об числення нових значень.

5.1. Граматика виразів

У мові KotlinScript розрізняються арифметичні та логічні вирази. Значення, обчислене за арифметичним виразом, має тип Float або Int. Значення, обчислене за логічним виразом, має тип Bool.

Синтаксичні правила виразів такі:

```
Expression = ArithmExpression | BoolExpression;

ArithmExpression = [Sign], Term
| (ArithmExpression, '+', Term)
| (ArithmExpression, '-', Term);

Term = Factor
| (Term, '*', Factor)
| (Term, '/', Factor)
| (Term, '^', Factor);

Factor = Id
| Literal
| ('(', ArithmExpression, ')');

BoolExpression = (Expression, RelOps, Expression) | BoolLiteral;
```

Приклади нетерміналів:

Factor:

y, 11, (g + 2)

Term:

j*3, 2^(4 - 1)

ArithmExpression:

3

-k, 11 + (3*(x - 1)), w - 21

BoolExpr:

9 - 3 * k < 3, true > false, true, false

5.2. Оператори

Всі бінарні оператори крім оператора возведення у степінь (^) у виразах цієї мови лівоасоціативні. Оператор возведення у степінь є правоасоціативним.

Найвищий пріоритет мають унарний мінус та унарний плюс, далі, у порядку зменшення пріоритету слідують MultOp, AddOp та RelOp.

5.2.1. Арифметичні оператори

Арифметичні оператори граматично представлені у формі

```
AddOps = '+' | '-';
MultOps = '*' | '/' | '^';
```

Тип результату при застосуванні бінарних операторів див. у табл. 3.

Оператор	Операція	Типи операндів	Тип результату
+	додавання	Int або Float	Float, якщо хоч один Float, інакше Int
-	віднімання	Int або Float	
*	множення	Int або Float	
/	ділення	Int або Float	Float
^	возведення у степінь	Int або Float	Float, якщо хоч один Float, інакше Int

5.2.2. Оператори відношення

Оператори відношення представлені у граматиці правилами:

```
RelOps = '===' | '==' | '<= ' | '<' | '>' | '>=' | '!=';
```

Значення обох операндів мають бути або числовими (типу Int або Float), або логічними (типу Bool). Результат завжди має тип Bool. Якщо один з операндів має тип Int, а інший — Float, то значення типу Int приводиться до типу Float.

6. Розділ інструкцій (statements)

Інструкції (statements) визначають алгоритмічні дії, які мають бути виконані у програмі.

Мова KotlinScript передбачає наявність у блоках списка інструкцій (statements), представленої граматично правилами:

```
Block = '{', [StatementsList], '}';
StatementsList = Statement, {';', Statement};
```

```
Statement = AssignStatement | InStatement | OutStatement | WhileStatement |
IfStatement;

AssignStatement =  ['var'], VarDeclaration, AssignOp, Expression;

InStatement='var', VarDeclaration, AssignOp, 'readln', '(', ')';
OutStatement='print','(', Expression ,')';

WhileStatement='while', '(', Expression , ')', Block;
IfStatement='if', '(', Expression, ')', Block, ['else', (IfStatement | Block)];
```

Блок може не містити інструкцій взагалі. Детальна синтаксична діаграма блоку показана на рисунку 6.1.

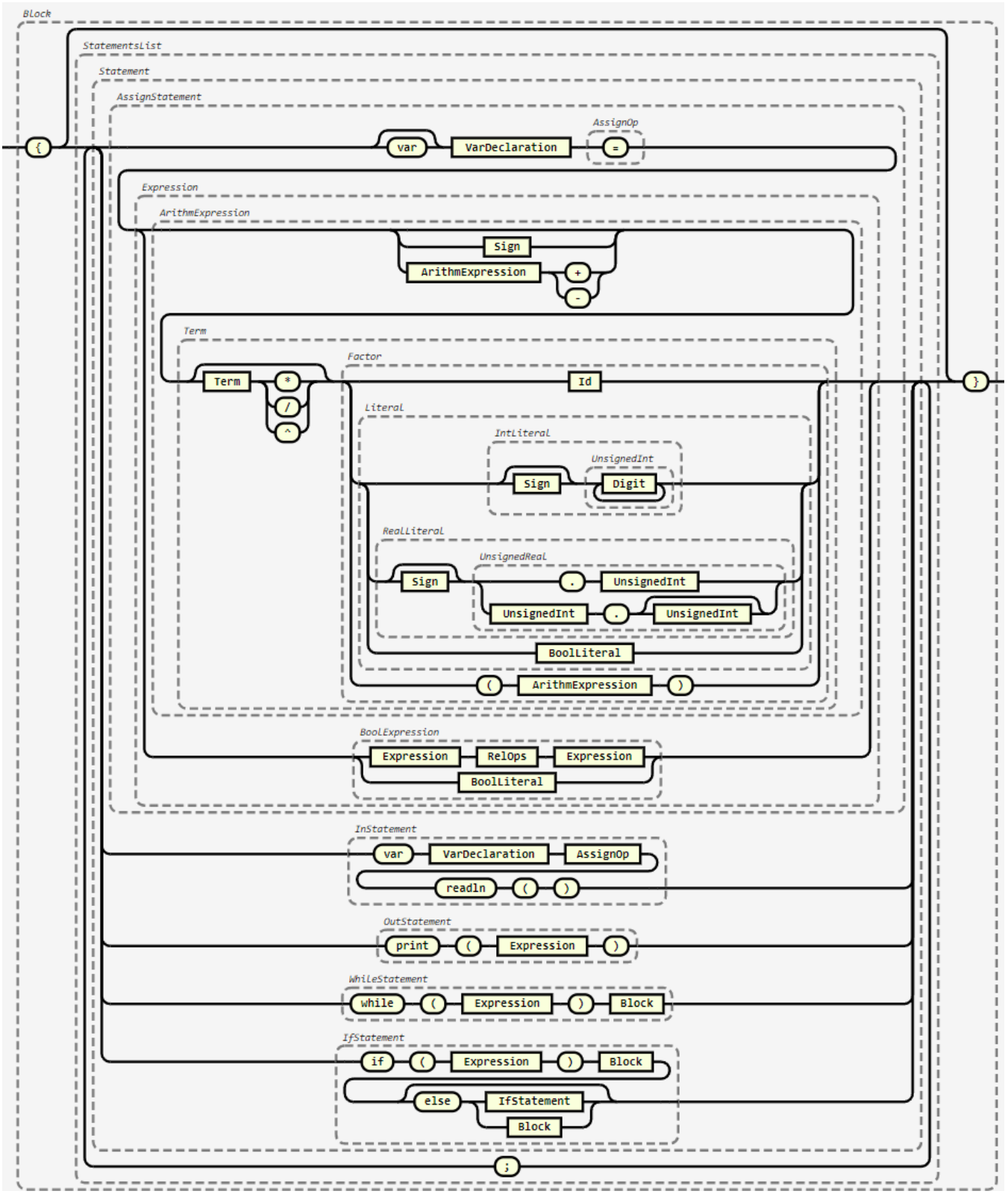


Рисунок 6.1 – Детальна синтаксична діаграма блоку

6.1. Оператор (інструкція) присвоєння значення

Синтаксис інструкції присвоєння визначається правилом

```
AssignStatement =  ['var'], VarDeclaration, AssignOp, Expression;
```

В інструкції присвоєння змінна, позначена ідентифікатором (Id) отримує значення виразу (Expression).

В інструкції присвоєння змінні використовуються по різному, в залежності від того, знаходиться вона зліва чи справа від символу ‘=’. Якщо змінна зліва від оператора (operator) присвоєння ’=’, то кажуть про її l-значення (lvalue чи left-value). l-значення має тип вказівника на місце зберігання значення змінної з ідентифікатором Id.

r-значенням (або lvalue та rvalue, або left-value та right-value). Значення, які можуть використовуватись у лівій та правій частинах інструкції присвоювання називають l-значенням та r-значенням (або lvalue та rvalue, або left-value та right-value).

Якщо змінна знаходиться справа від оператора (operator) присвоєння ’=’, то кажуть про її r-значення (rvalue, right-value).

Про значення виразу Expression, який теж знаходиться праворуч від оператора присвоювання ’=’, також можна говорити як про r-значенням. Тип змінної з ідентифікатором Id повинен збігатись з типом значенням виразу Expression (типом r-значення).

Діаграма Вірта представлена на рис. 6.2

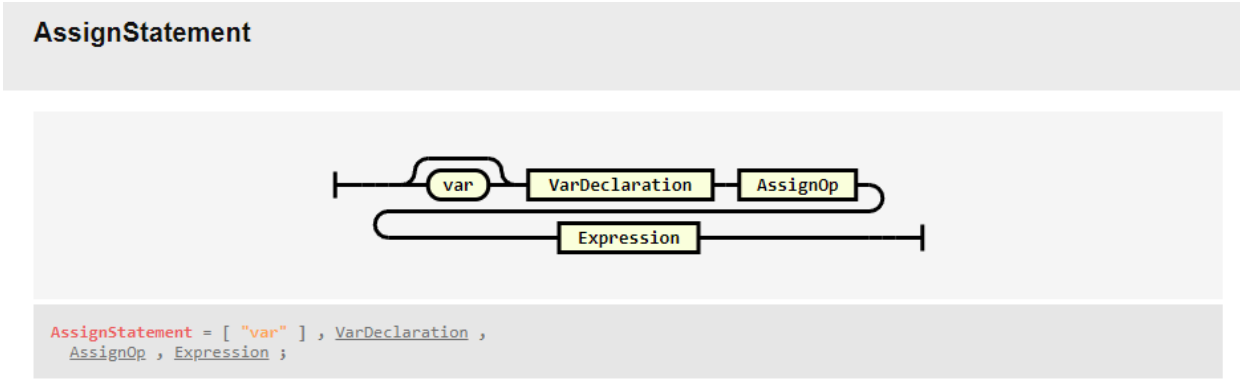


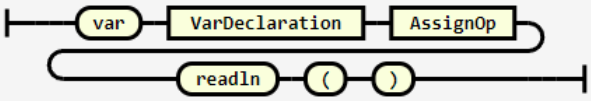
Рисунок 6.2 – Діаграма інструкції присвоєння значення

6.2. Інструкція введення

Значення вводяться з клавіатури. Введення значення підтверджується клавішею Enter.

Прочитане з клавіатури значення присвоюється змінній Id всередині VarDeclaration.

InStatement



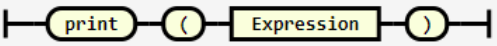

```
InStatement = "var" , VarDeclaration , AssignOp ,
  "readln" , "(" , ")" ;
```

6.3. Інструкція виведення

Значення виводить у один рядок консолі.

Спроба виведення змінної з невизначеним значенням викликає помилку.

OutStatement

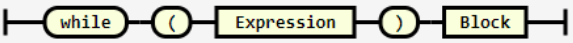



```
OutStatement = "print" , "(" , Expression , ")" ;
```

6.4. Оператор циклу (інструкція повторення)

Стандартна конструкція while використовується у інструкції циклу.

WhileStatement

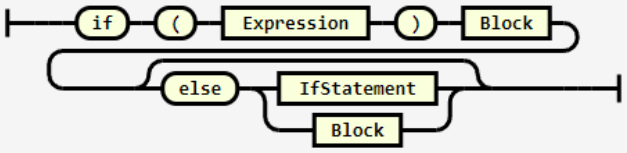



```
WhileStatement = "while" , "(" , Expression , ")" , Block ;
```

6.5. Оператор розгалуження

Рекурсивна конструкція if-else використовується у якості інструкції розгалуження. Якщо результат expression числовий, то відбувається його конвертація у BoolExpression шляхом порівняння з 0: Expression != 0.

IfStatement




```
IfStatement = "if" , "(" , Expression , ")" , Block ,
  [ "else" , ( IfStatement | Block ) ] ;
```

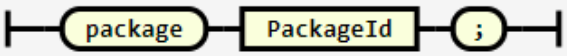
7. Пакети та початкова точка запуску

У мові KotlinScript, яка не надає інших засобів комунікації із оточенням крім взаємодії зі стандартним потоком введення/виведення даних, кожна програма є монолітною програмною сутність, ізольованою від будь-якого іншого коду.

7.1. Структура програми

Код має починатись з декларації пакету, який визначений таким чином:

PackageDeclaration




```
PackageDeclaration = "package" , PackageId , ";" ;
```

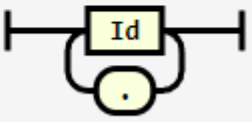
Items referencing **PackageDeclaration**:

- Grammar

PackageDeclaration is referencing:

- PackageId

PackageId




```
PackageId = Id , { "." , Id } ;
```

Items referencing **PackageId**:

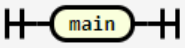
- PackageDeclaration

PackageId is referencing:

- Id

Слідом за декларацією пакету має йти EntryPoint (точка входу), що представляє собою декларацію функції main, у блоці якої будуть знаходитись усі інструкції програми:

EntryPointFunName

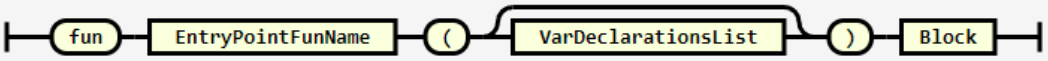



```
EntryPointFunName = "main" ;
```

Items referencing **EntryPointFunName**:

- EntryPoint

EntryPoint




```
EntryPoint = "fun" , EntryPointFunName , "(" ,
  [ VarDeclarationsList ] , ")" , Block ;
```

Items referencing **EntryPoint**:

- Grammar

EntryPoint is referencing:

- EntryPointFunName
- VarDeclarationsList
- Block

7.2. Базовий приклад програми

```
package com.example.lab;

fun main() {
  var a = 1;
  var b = 2;
  var c = a + b;

  var d: Int;

  d = 3;
```

```
  d = d * a;

  var e: Float = .2;

  e = e / 2 - 1;

  var readNum: Int = readln();

  var f = -1 + e * (a ^ c) + b * (1.2 + a * (readNum - 1));

  // this is variable i
  var i = 0;
  var l = 2;
  while(i < 3) {
    if (l > 410) {
      l = l / 2;
    } else if (l <= 0) {
      l = l + 1;
    } else if (l >= 20) {
      l = l + 12;
    } else {
      l = l * 2;
    }

    if (i == true) {
      if (false == 0) {
        print(true);
      }
    }

    if (i === 2) {
    } else if (i != 1) {
      print(f + i);
    }

    i = i + 1;
  }
}
```

8. Повна граматика мови KotlinScript

```
LowerLetter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k'
| 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
| 'x' | 'y' | 'z';
UpperLetter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K'
| 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W'
| 'X' | 'Y' | 'Z';
Letter = UpperLetter | LowerLetter;
NonZeroDigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Digit = '0' | NonZeroDigit;
SpecSign = '.' | ',' | ':' | ';' | '(' | ')'
| '=' | '+' | '-' | '*' | '/' | '<' | '>' | '!'
| WhiteSpace | EOL;
WhiteSpace = ' ' | '\t';
EOL = ['\r'],'\n';



SpecSymbols = ArithmOps | RelOps | BracketsOps | AssignOp | Punct;
ArithmOps = AddOps | MultOps;
AddOps = '+' | '-';
MultOps = '*' | '/' | '^';
RelOps = '===' | '==' | '<= ' | '<' | '>' | '>=' | '!=';
BracketsOps = '(' | ')';
AssignOp = '=';
Punct = '.' | ',' | ':' | ';';
InlineCommentOp = '//';



Id = Letter, { Letter | Digit };
IdList = Id, {',', Id };

Literal = IntLiteral | RealLiteral | BoolLiteral;
IntLiteral = [Sign], UnsignedInt;
RealLiteral = [Sign], UnsignedReal;
Sign = '+' | '-';
UnsignedInt = Digit, {Digit};
UnsignedReal = ('.', UnsignedInt)
| (UnsignedInt, '.')
```

```
| (UnsignedInt, '.', UnsignedInt);
BoolLiteral = 'true' | 'false';

Keyword = 'fun' | 'var' | 'while' | 'if' | 'else' | 'package' | Type;

Type = 'Int' | 'Float' | 'Bool';
VarTypeSpecifier = ':', Type;

VarDeclaration = Id, [VarTypeSpecifier];
VarDeclarationsList = VarDeclaration, { ',', VarDeclaration };


Expression = ArithmExpression | BoolExpression;

ArithmExpression = [Sign], Term
| (ArithmExpression, '+', Term)
| (ArithmExpression, '-', Term);

Term = Factor
| (Term, '*', Factor)
| (Term, '/', Factor)
| (Term, '^', Factor);

Factor = Id
| Literal
| ('(', ArithmExpression, ')');

BoolExpression = (Expression, RelOps, Expression) | BoolLiteral;


Block = '{', [StatementsList], '}';
StatementsList = Statement, {';', Statement};

Statement = AssignStatement | InStatement | OutStatement | WhileStatement |
IfStatement;

AssignStatement =  ['var'], VarDeclaration, AssignOp, Expression;

InStatement='var', VarDeclaration, AssignOp, 'readln', '(', ')';
OutStatement='print','(', Expression ,')';
```

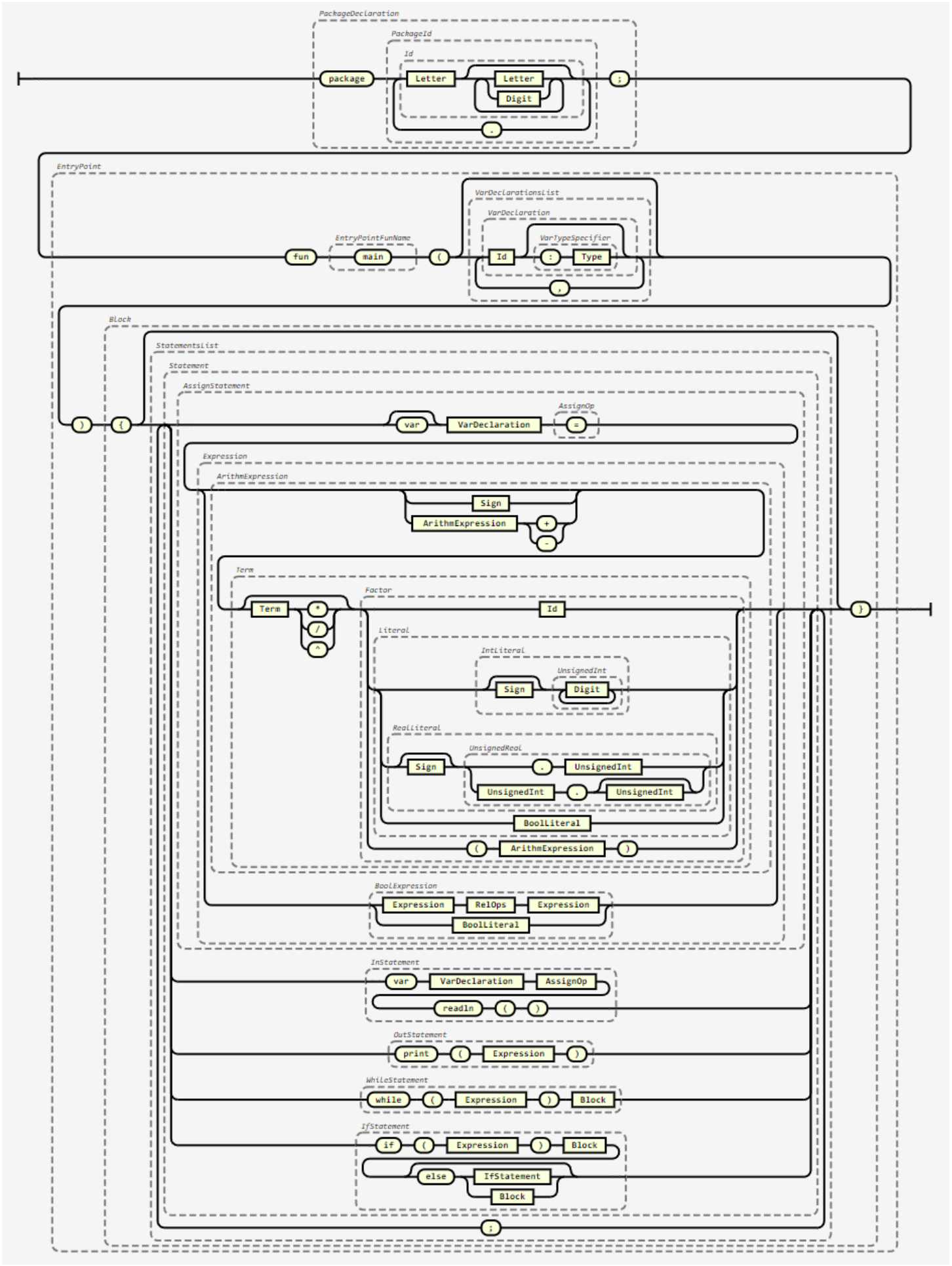
```
WhileStatement='while', '(', Expression , ')', Block;
IfStatement='if', '(', Expression, ')', Block, ['else', (IfStatement | Block)];

PackageId = Id, { '.', Id };
PackageDeclaration = 'package', PackageId, ';';

EntryPointFunName = 'main';

EntryPoint = 'fun', EntryPointFunName, '(', [VarDeclarationsList], ')', Block;

Grammar = PackageDeclaration, EntryPoint;
```


PackageDeclaration
PackageId
Id
package
Letter
Letter
Digit
.
;
EntryPoint
VarDeclarationsList
VarDeclaration
VarTypeSpecifier
EntryPointFunName
fun
main
(
Id
:
Type
,
)
{
Block
StatementsList
Statement
AssignStatement
AssignOp
var
VarDeclaration
=
Expression
ArithmExpression
Sign
ArithmExpression
+
-
Term
Factor
Term
*
/
^
Id
Literal
IntLiteral
UnsignedInt
Sign
Digit
RealLiteral
UnsignedReal
Sign
.
UnsignedInt
UnsignedInt
.
UnsignedInt
BoolLiteral
(
ArithmExpression
)
BoolExpression
Expression
RelOps
Expression
BoolLiteral
InStatement
var
VarDeclaration
AssignOp
readln
(
)
OutStatement
print
(
Expression
)
WhileStatement
while
(
Expression
)
Block
IfStatement
if
(
Expression
)
Block
else
IfStatement
Block
;
}