Лабораторная работа №11
«Разработка парсеров на языке Java»

1. Цель работы

Овладение методом рекурсивного спуска для разработки парсеров по грамматике некоторого формального языка.

2. Исходные данные

2.1. Основные определения

Определение 1. Контекстно-свободной грамматикой (КС-грамматикой) называется четвёрка $\langle T,N, P, S\rangle$, в которой:

- T – конечное множество терминальных символов;
- N – конечное множество нетерминальных символов, причём $N \cap T = \emptyset$;
- P – конечное множество правил вида $X \rightarrow \langle a_1, a_2, \ldots, a_n\rangle$, где $X \in N$, $a_i \in T \cup N$;
- S – начальный символ (или аксиома), причём $S \in N$.

Объединение множества терминальных символов и множества нетерминальных символов называется объединённым алфавитом грамматики.

Определение 2. Назовём цепочкой кортеж, составленный из символов объединённого алфавита грамматики. Кортеж $\langle a_1, a_2, \ldots, a_n\rangle$ по сути является строкой, и его принято записывать как $a_1a_2 \ldots a_n$. При этом пустой кортеж обозначают буквой ε.

Таким образом, правило грамматики $X \rightarrow \langle a_1, a_2, \ldots, a_n\rangle$ на практике записывается как $X \rightarrow a_1a_2 \ldots a_n$ или, в частном случае, как $X \rightarrow \varepsilon$. При этом нетерминальный символ X слева от стрелки называется *левой частью* правила, а цепочка $a_1a_2 \ldots a_n$ – *правой частью* правила.

Мы будем говорить, что цепочка $u = a_1 \ldots a_{k-1}\mathbf{a_k}a_{k+1} \ldots a_n$ непосредственно порождает цепочку $v = a_1 \ldots a_{k-1}\ \mathbf{b_1 \ldots b_m}a_{k+1} \ldots a_n$ (или, что то же самое, v непосредственно выводится из u), если a_k – нетерминальный символ, и существует правило $a_k \rightarrow b_1 \ldots b_m$. При этом мы будем использовать запись $u \Rightarrow v$.

Определение 3. Последовательность порождений $u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_n$, называют выводом и обозначают как $u_1 \Rightarrow^* u_n$.

Если каждое порождение $u_i \Rightarrow u_i{+}1$ в составе вывода заключается в замене самого левого нетерминального символа в цепочке u_i правой частью соответствующего ему правила грамматики, то такой вывод называется левым выводом.

Если имеется вывод $u \Rightarrow^* w$, то говорят, что u порождает w, или, что то же самое, w выводится из u. Отметим, что для общности разрешаются выводы нулевой длины, то есть любая цепочка выводится из самой себя: $u \Rightarrow^* u$.

Определение 4. Предложение – это составленная только из терминальных символов цепочка, порождаемая цепочкой S, где S – аксиома грамматики.

Определение 5. Язык L, порождаемый КС-грамматикой $\langle T,N, P, S\rangle$, – это множество предложений, порождаемых аксиомой этой грамматики: $L = \{u \mid S \Rightarrow^* u\}$.

Определение 6. Задача синтаксического анализа – определить, является ли заданная цепочка терминальных символов предложением языка, и если является, то построить её вывод из аксиомы грамматики этого языка. Программа, решающая задачу синтаксического анализа, называется синтаксическим анализатором или парсером.

2.2 Бэкус-наурова форма

Для записи правил грамматики широко используется так называемая бэкус-наурова форма (сокращённо, БНФ), в которой левая и правая части каждого правила разделяются знаком «::=», и несколько правил:

<X> ::= u,
<X> ::= v,
. . .
<X> ::= w

с общей левой частью записываются в сокращённом виде как

<X> ::= u | v | ... | w.

Чтобы отличать нетерминальные символы от терминальных, нетерминальные символы часто заключают в угловые скобки. Аксиомой грамматики, записанной в БНФ, считается нетерминальный символ в левой части самого первого правила.

2.3 Метод рекурсивного спуска

Метод рекурсивного спуска – это формальный метод составления парсера, который позволяет перевести правила грамматики, записанной в БНФ, в программу на выбранном языке программирования.

Синтаксические анализаторы, составленные по методу рекурсивного спуска, работают за линейное время. Метод применим к ограниченному классу грамматик, называемых LL(1)-грамматиками. Определение LL(1)-грамматики выходит за рамки данной лабораторной работы и будет дано в курсе «Конструирование компиляторов».

Метод рекурсивного спуска заключается в том, что парсер строится как набор взаимнорекурсивных функций, по одной функции на каждый нетерминальный символ. Мы будем именовать эти функции как ParseX, где X – имя нетерминального символа.

Запуск парсера осуществляется путём вызова функции ParseS, соответствующей аксиоме грамматики.

В процессе работы парсер, составленный методом рекурсивного спуска, проходит по заданной цепочке терминальных символов от её начала до конца. При этом текущий рассматриваемый символ записывается в переменную Sym, доступную из любой функции парсера, а загрузка следующего символа в переменную Sym осуществляется путём вызова функции Next. В простейшем случае функция Next просто читает следующий символ Unicode из входной строки, однако в реальных грамматиках терминальными символами являются не символы Unicode, а составленные из них лексемы (например, идентификатор, число в десятичной записи, строковый литерал), поэтому функция Next обычно осуществляет лексический анализ входной строки и при каждом вызове записывает в переменную Sym следующую прочитанную из входной строки лексему.

Каждая функция ParseX в составе парсера предназначена для того, чтобы прочитать, начиная с текущей позиции во входной последовательности символов, цепочку, которая может быть порождена нетерминальным символом X. Предполагается, что при вызове функции ParseX переменная Sym содержит первый символ этой цепочки, а после завершения работы функции ParseX переменная Sym будет содержать символ, непосредственно следующий за этой цепочкой.

Чтобы разобраться, как по методу рекурсивного спуска составлять функцию ParseX для нетерминального символа X, рассмотрим сначала простейший случай, когда символ X стоит в левой части одного-единственного правила грамматики:

$<X> ::= a_1\ a_2\ ...\ a_N.$

В этом случае функция ParseX представляет собой последовательность из N участков кода, каждый из которых соответствует символу из правой части правила.

Если символ aI в правой части правила – терминальный, то ему соответствует следующий
участок кода:

```
if Sym != aI:
        panic «Ожидается символ aI»
Sym = Next
```

То есть сначала мы проверяем, что символ во входной последовательности равен ожидаемому на этом месте символу aI, и аварийно завершаем работу парсера, если это не так. А затем мы пропускаем этот символ, загружая в переменную Sym следующий символ из входной последовательности.

Если символ aI в правой части правила – нетерминальный, то мы просто вызываем соответствующую ему функцию, чтобы она прочитала из входной последовательности порождённую им цепочку символов:

```
ParseAI
```

Теперь рассмотрим общий случай, когда для нетерминального символа X в грамматике существуют сразу несколько правил:

$<X> ::= u_1 \mid u_2 \mid ... \mid u_N.$

Здесь функция ParseX должна определить, какое из правил было применено для порождения цепочки. При этом единственная информация, которой она обладает для ответа на этот вопрос, – это текущий терминальный символ, записанный в переменной Sym.

LL(1)-грамматика обладает важным свойством: цепочки, которые могут порождаться правыми частями правил, соответствующих одному нетерминальному символу, начинаются с различных терминальных символов. Этот факт позволяет по текущему символу Sym однозначно идентифицировать нужное правило грамматики.

Определение 7. Множество FIRST для цепочки u – это множество терминальных символов, с которых могут начинаться цепочки, выводимые из u, дополненное специальным признаком ε, если из цепочки u может выводиться пустая цепочка:

$$FIRST(u) = \{a \mid u \Rightarrow^* av\} \cup \{\varepsilon \mid u \Rightarrow^* \varepsilon\}.$$

Существует несложный алгоритм для вычисления множества FIRST для любой цепочки. Мы не будем его рассматривать, так как в случае простых грамматик прикинуть состав множества FIRST можно просто в уме.

Пусть множества FIRST для правых частей правил, соответствующих нетерминальному символу X, не пересекаются. При этом ни одно из множеств не содержит признака ε. Тогда тело функции ParseX должно иметь следующий вид:

```
if Sym∈FIRST(u1):
```

```
        разбор цепочки u1
else if Sym∈ FIRST(u2):
        разбор цепочки u2
        ...
else if Sym∈ FIRST(uN):
        разбор цепочки uN
else:
        panic «синтаксическая ошибка»
```

Здесь фраза «разбор цепочки uI» означает фрагмент кода, составленный способом, рассмотренным нами выше для правой части единственного правила, соответствующего нетерминальному символу X.

Если одно из множеств FIRST содержит признак ε, значит нетерминал X может породить пустую цепочку. Без потери общности будем считать, что признак ε содержит множество FIRST(uN). В этом случае функция ParseX должна принимать вид

```
if Sym∈ FIRST(u1):
        разбор цепочки u1
else if Sym∈ FIRST(u2):
        разбор цепочки u2
        ...
else:
        разбор цепочки uN
```

Действительно, если uN порождает пустую цепочку, то символ Sym не порождается нетерминальным символом X, так как он непосредственно следует за пустой цепочкой, порождаемой символом X. Поэтому проверять, принадлежит ли Sym множеству FIRST(uN) нельзя.

3. Задание

В ходе лабораторной работы нужно разработать программу, выполняющую синтаксический анализ текста по одной из LL(1)-грамматик, БНФ которых приведены в таблицах 1–6. Текст может содержать символы перевода строки.

В записи БНФ терминальные символы IDENT, NUMBER и STRING означают идентификаторы, числа и строки, соответственно. Идентификатор – это последовательность букв и цифр, начинающаяся с буквы. Число – это непустая последовательность десятичных цифр. Строка – это обрамлённая кавычками произвольная последовательность символов, не содержащая кавычек и символов перевода строки.

Программа должна выводить в стандартный поток вывода последовательность правил грамматики, применение которых даёт левый вывод введённого из стандартного потока ввода текста. Если вывод не может быть построен, программа должна выводить сообщение «syntax error at (line, col)», где line и col – координаты ошибки в тексте.

4. Замечание

Ниже приведены таблицы «Варианты грамматик с примерами предложений».

	Грамматика	Предложение	Студент	Группа
1	<List>::= <Item> <Tail> <Tail>::= , <List> |ε <Item>::= IDENT | { <List> }	alpha, {beta, gamma}, delta		
2	<Decl>::= <Type> <Ptr> <Type>::= int | float <Ptr>::= * <Ptr> | <Prim> <Dims> <Dims>::= [NUMBER] <Dims> | ε <Prim>::= IDENT | (<Ptr>)	int *(*a[10][20])[5]		

3	<Braces> ::= <Item> <Tail>
	<Tail>::= <Braces> | ε
	<Item>::= (<Braces>)
	| { <Braces> }
	| [<Braces>]
	| NUMBER
		10 20 {
		30 [40] (50 60)
		}		

4	<Bool>::= <Term> <Ors> <Ors>::= or <Term> <Ors> | ε <Term>::= <Factor> <Ands> <Ands>::= and <Factor> <Ands> | <Factor> ::= IDENT | true | false | (<Bool>)	A and (B or true)		
5	<Apply> ::= <Lambda> <Apply> | ε <Lambda> ::= IDENT <Apply> | \ IDENT . <Lambda> | (<Lambda>)	\f.(\x.f(x x)) (\x.f(x x))		
6	<Json>::= [<Array>] | { <Map> } | NUMBER | STRING | [ ] | { } <Array> ::= <Json> <JTail> <JTail> ::= , <Array> | ε <Map>::= <Pair> <MTail> <MTail> ::= , <Map> | ε <Pair>::= STRING : <Json>	{ "alpha": 1, "beta": [10, "10", {}], "gamma": { "x": [] } }		
7	<Expr>::= (<List>) <List>::= <Atom> <Tail> <Tail>::= <Atom> <Tail> | ε <Atom>::= NUMBER | IDENT | <Expr>	(A B (10))		
8	<Type>::= integer | real	array[1..10,1..3] of ^real		

| array [<Dim>] of <Type>
| ^ <Type>
<Dim>::= <Rng> <Tail>
<Tail>::= , <Dim> | ε
<Rng>::= NUMBER .. NUMBER

9 <Decl>::= <Enum> <VarsOpt> ;
<VarsOpt>::= <Vars> | ε
<Enum>::= enum IDENT { <List> }
<List>::= <Item> <LTail>
<LTail> ::= , <List> |
ε
<Item>::= IDENT <ITail>
<ITail> ::= = NUMBER |
ε
<Vars>::= IDENT <VTail>
<VTail> ::= , <Vars> | ε

```
enum Day {
SUN = 0, MON, TUE,
WED, THU, FRI, SAT
} first, last;
```

10 <Expr>::= <Term> <Adds>
<Adds>::= + <Expr> | - <Expr> | ε
<Term>::= IDENT | NUMBER
| (<Expr>)

```
a + 10 - (x - 1)
```

11 <Stmt>::= if (IDENT) <Stmt>
| IDENT = NUMBER ;
| { <Seq> }
<Seq>::= <Stmt> <Seq> | ε

```
if (x) {
x = 0;
if (y) z = 10;
}
```

12 <Expr>::= * <Expr> | <Prim>
<Tail>
| <Op> <Expr>
<Tail>::= ε | <Op> <Tail>
| [<Expr>] <Tail>
<Op>::= ++ | --
<Prim>::= IDENT | (<Expr>)

```
***ptr[x++][--y[i]]--
```

13 <Expr>::= (cons <List>) | nil
| NUMBER
<List>::= <Expr> <Tail>
<Tail>::= <List> | ε

```
(cons 10 (cons 1) nil)
```

14 <Type>::= <Base> <Tail>
<Tail>::= -> <Type> | ε
<Base>::= <Cort> <List>
<List>::= list <List> | ε
<Cort>::= <Factor> <CTail>
<CTail>::= * <Cort> | ε
<Factor>::= int | real | (<Type>)

```
int * (int list) ->
int * real list ->
real
```

15 <Lambda> ::= NUMBER

```
(-> -> 1): (-> 0: 0)
```

<Apply>
| -> <Lambda>
| (<Lambda>)
<Apply> ::= : <Lambda> <Apply>
| ε

16 <Dir>::= #define IDENT <Arg> <Seq>
<Arg>::= ε | (<List>)
<List>::= IDENT <LTail>
<LTail> ::= , <List> | ε
<Seq>::= <Item> <STail>
<STail> ::= <Seq> | ε
<Item>::= NUMBER | IDENT | STRING

#define A(x,y) "abc" 10

17 <Stmt>::= <Expr> = <Expr>
<Expr>::= <Atom> <Expr> | ε
<Atom>::= IDENT | STRING
| (<Expr>)

"abc" x ("a" "bd" y) =
(x y () ("qwerty"))

18 <Pal>::= IDENT <Pal> IDENT
| NUMBER <Pal> NUMBER
| STRING <Pal> STRING
| (<Mid>)
<Mid>::= <Pal> <Mid> | ε

z 10 x (a () a b () b) y 20 z

19 <Decl>::= struct { <Fs> } <Vars>
;
<Vars>::= IDENT <VTail>
<VTail>::= , <Vars> | ε
<Fs>::= <Field> <FsTail>
<FsTail> ::= <Fs> | ε
<Field> ::= int <Vars> ; | <Decl>

```
struct {
int x, y;
struct {
int a;
int b;
} pair;
} p;
```

20 <Chain> ::= IDENT <Tail>
<Tail>::= . IDENT (<Arg>)
<Tail>
|
ε
<Arg>::= NUMBER | STRING |
<Chain>

a.mul(5).sub(b.div(c))

21 <Gener> ::= IDENT <Tail>
<Tail>::= [<Args>] | ε
<Args>::= <Gener> <List>
<List>::= , <Args> | ε

Map[Int, List[String]]

22 <Eq>::= IDENT <List>
<List>::= : <Term> <Tail> | ε

x: "y" + (z: "a" + "b" + "c")

<Tail>::= + <Term> <Tail> | ε
<Term>::= STRING | (<Eq>)

23 <Pattern>::= <Term> <Tail> (1::x::Nil)::("a",*)::Nil
<Tail>::= :: <List> | ε
<List>::= <Term> :: <List> | Nil
<Term>::= NUMBER | STRING |
IDENT
| * | (<Tuple>)
<Tuple> ::= <Pattern> <TupTail>
<TupTail>::= , <Tuple> | ε

24 <Expr>::= <Range> <Tail> 1..5 in (0..4 + x)*y
<Tail>::= in <Range> | ε
<Range> ::= <Term> <RngTail>
<RngTail>::= + <Range> | ε
<Term>::= <Factor> <TmTail>
<TmTail> ::= * <Term> | ε
<Factor> ::= NUMBER ..
NUMBER
| IDENT | (<Range>)

25 <Type>::= int | float32 | <FType> func (x, y int,
<FType> ::= func (<Args>) f func(z float32) int
<Ret>)
<Ret>::= <Type> | ε
<Args>::= <List> <ATail> | ε
<ATail> ::= , <List> <ATail> | ε
<List>::= IDENT <LTail>
<LTail> ::= , <List> | <Type>

26 <Class> ::= class IDENT <Body> class X
<Body>::= <Method> <Body> | def m1
end end
<Method> ::= def IDENT <Args> def m2(a, b)
end end
<Args>::= (<List>) | ε end
<List>::= IDENT <Tail> | ε
<Tail>::= , IDENT <Tail> | ε

27 <Type>::= i32 | String | <Func> fn (x: i32) ->
<Func>::= fn (<Args>) <Ret> fn (f: fn(s: String)) ->
<Ret>::= -> <Type> | ε i32
<Args>::= <List> | ε
<List>::= IDENT : <Type> <Tail>
<Tail>::= , <List> | ε

28 <New>::= new <Type> { <Items> new int[][] {
} new int[] { 1, 2, 3 },
<Type>::= <Base> <Tail> new int[] { 4, 5, 6 }
<Base>::= int | String }

<Tail>::= [ ] <Tail> | ε
<Items> ::= <List> | ε
<List>::= <Val> <LTail>
<LTail> ::= , <List> | ε
<Val>::= IDENT | NUMBER |
STRING
| <New>

29 <Graph> ::= digraph <Block>
<Block> ::= { <Body> }
<Body>::= <List> |
ε
<List>::= <Stmt> <Tail>
<Tail>::= ; <List> |
ε
<Stmt>::= <Id> <Rest> <Attr>
| subgraph <Id> <Block>
<Rest>::= -> <Id> |
ε
<Attr>::= [label = STRING] |
<Id>::= IDENT | NUMBER

```
digraph {
A [label = "alpha"];
subgraph B {
1 [label = "one"];
2 [label = "two"];
1 -> 2
};
A -> B
}
```

30 <Ini>::= <Section> <IniTail>
<IntTail>::= <Ini> | ε
<Section>::= [IDENT] <Pairs>
<Pairs> ::= IDENT = <Value>
<Pairs>
| ε
<Value> ::= NUMBER | STRING

```
[base]
Name = "Вася"
Age = 18
Position = "студент"
[aux]
Description = "двоечник"
```

31 <Tm>::= template < <Args> >
<Args>::= <Arg> <Tail>
<Tail>::= , <Args> | ε
<Arg>::= typename <Name>
| int <Name>
| <Tm> <Class>
<Name>::= IDENT | ε
<Class> ::= class IDENT | ε

```
template <
template <typename, int>
class C,
typename T, int N
>
```

32 <Expr>::= <Tuple> | <Basic> |
<Let>
<Tuple> ::= (<Items>)
<Items> ::= <Expr> <Tail> | ε
<Tail>::= , <Expr> <Tail> | ε
<Basic> ::= NUMBER | STRING
| IDENT <Args>
<Args>::= <Tuple> | ε
<Let>::= let <List> in <Expr> end
<List>::= <Expr> = <Expr>
<LTail>
<LTail> ::= , <List> | ε

```
let (x, y) = f(10),
z = g(y)
in (x, z, "a")
end
```

33 <Decl>::= type IDENT = <Type> type IntArray =array[1..100] of
<Type>::= <Range> ^integer;
| array [<Range>]
of <Type>
| ^ <Type>
| string
<Range> ::= integer | char
| NUMBER .. NUMBER

34 <Call>::= IDENT (<Args>) f(g(), h(10))
| IDENT ( )
| NUMBER
<Args>::= <Call> <Tail>
<Tail>::= , <Args> | ε

35 <Parcel> ::= <Call> <Tail> (f 4):(e (g 6))
<Tail>::= : <Parcel> | ε
<Call>::= (IDENT <Arg>)
<Arg>::= NUMBER | STRING |
<Parcel>