$S \rightarrow AB \mid \epsilon$ ①

$A \rightarrow aB$ ②

$B \rightarrow Sb$ ③

abb

Left most Derivation

S

AB

aBB

aSbB

abB

abSb

abb

$S \rightarrow AB$

$A \rightarrow aB$

$B \rightarrow Sb$

$S \rightarrow \epsilon$

$B \rightarrow Sb$

$S \rightarrow \epsilon$

Right most Derivation

S

AB $\quad S \rightarrow AB$

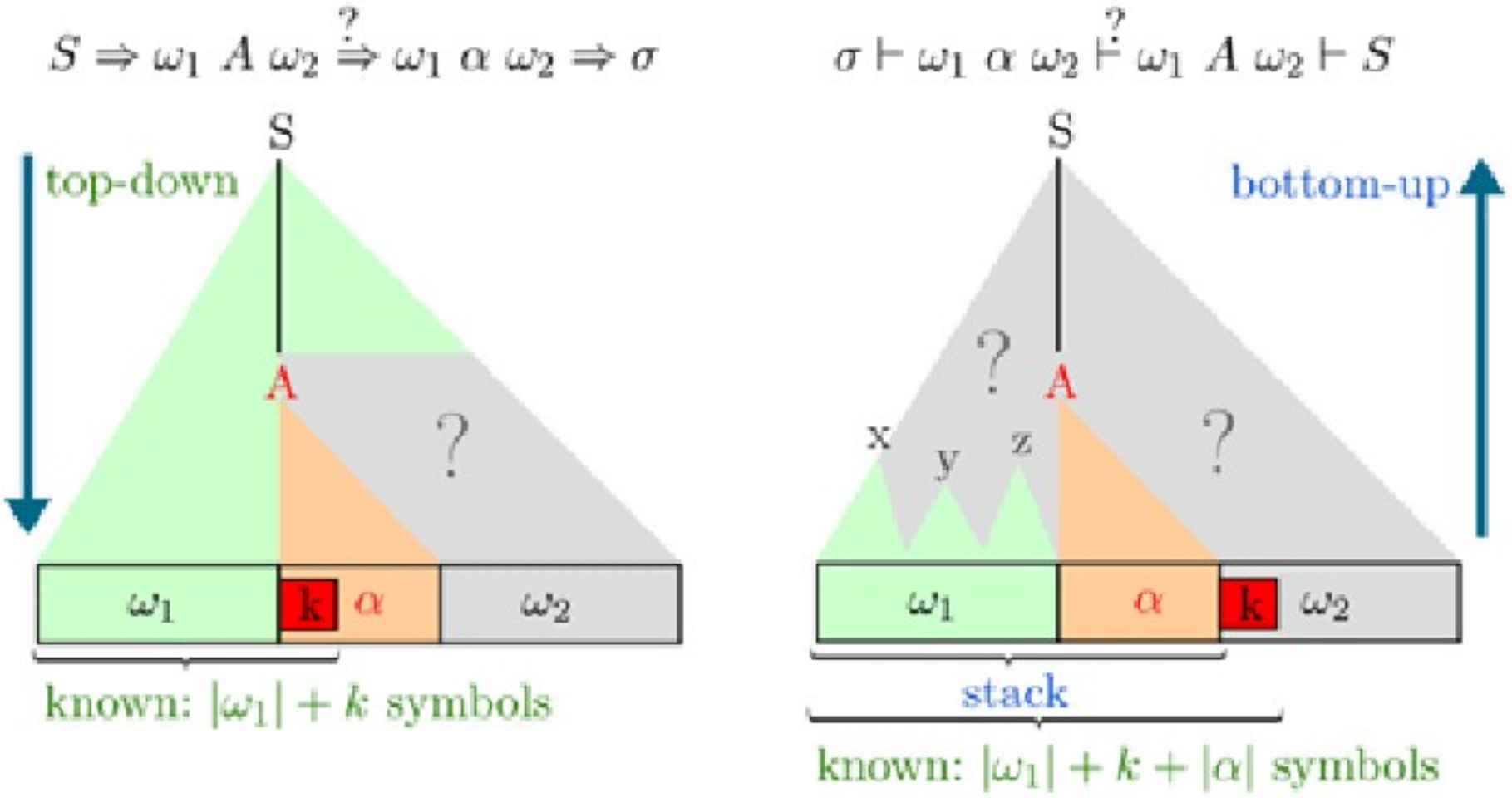
ASb $\quad B \rightarrow Sb$

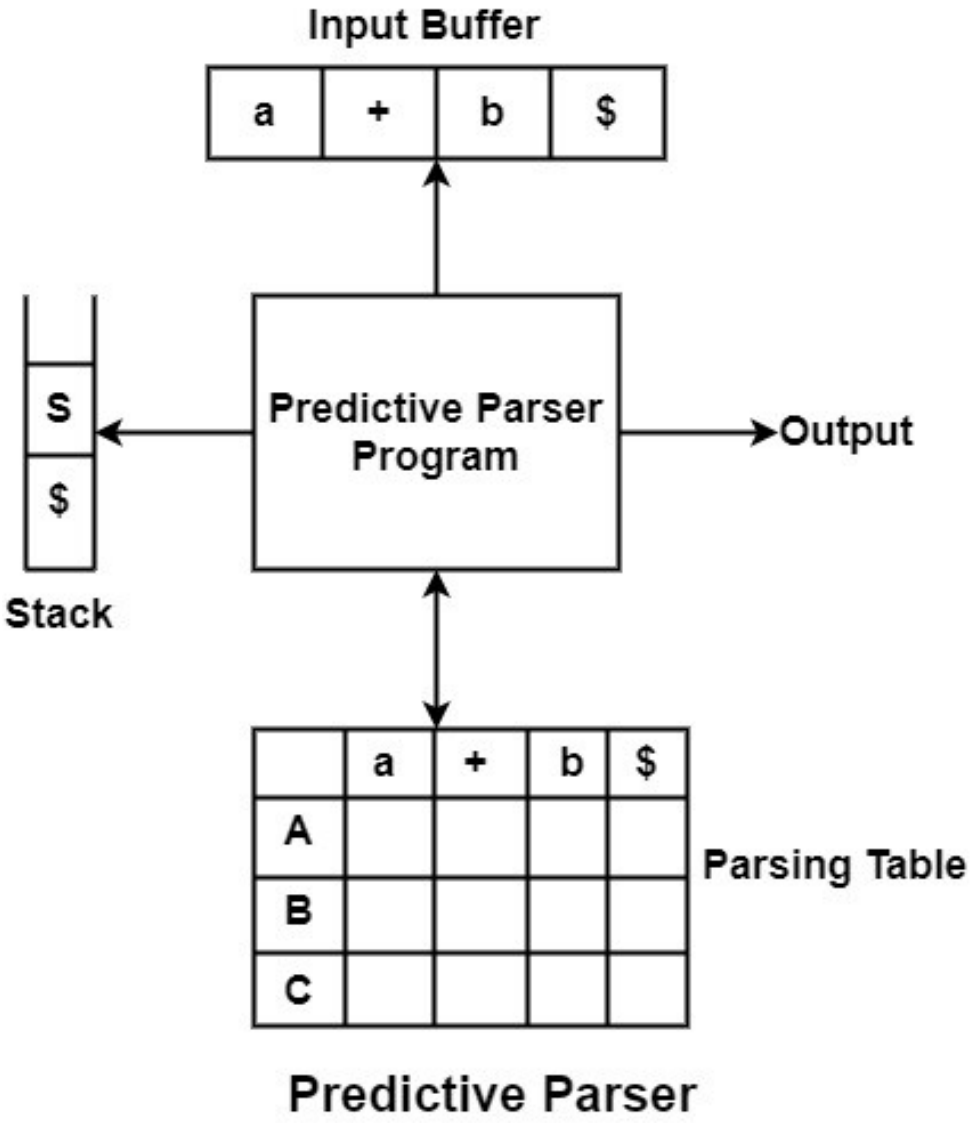
Aϵb $\quad S \rightarrow \epsilon$

aBb $\quad A \rightarrow aB$

aSbb $\quad B \rightarrow Sb$

abb $\quad S \rightarrow \epsilon$


$S \Rightarrow \omega_1\ A\ \omega_2 \overset{?}{\Rightarrow} \omega_1\ \alpha\ \omega_2 \Rightarrow \sigma$
S
top-down
A
?
ω_1
k
α
ω_2
known: $|\omega_1| + k$ symbols
$\sigma \vdash \omega_1\ \alpha\ \omega_2 \overset{?}{\vdash} \omega_1\ A\ \omega_2 \vdash S$
S
bottom-up
?
A
x
y
z
?
ω_1
α
k
ω_2
stack
known: $|\omega_1| + k + |\alpha|$ symbols



Predictive Parser

* Table Driven predictive parsee is LL(1)

1) Remove left recursion, Ambiguous grammar not allowed in LL(1).
2) compute FIRST & FOLLOW set.
3) construct predictive parsing table using Algorithm
4) parse string using parsee.

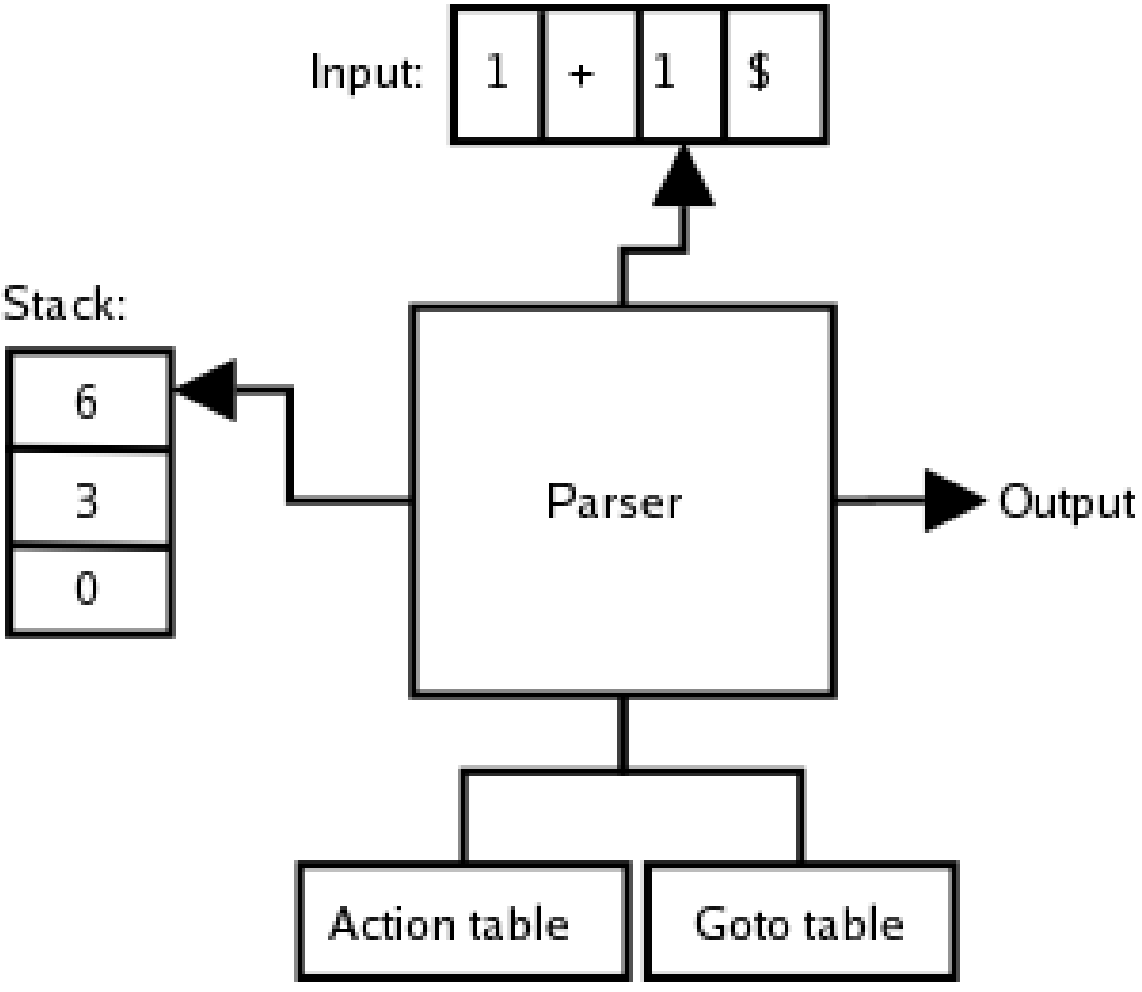
	FIRST	FOLLOW
$E \to TE'$	$E \to \{(, id\}$	$E \to \{\$,)\}$
$E' \to +TE' \mid \epsilon$	$E' \to \{+, \epsilon\}$	$E' \to \{\$,)\}$
$T \to FT'$	$T \to \{(, id\}$	$T \to \{+, \$,)\}$
$T' \to *FT' \mid \epsilon$	$T' \to \{*, \epsilon\}$	$T' \to \{+, \$,)\}$
$F \to (E) \mid id$	$F \to \{(, id\}$	$F \to \{*, +, \$,)\}$

LL(1) parsing Table

	V \ T	id	+	*	(	)	$
1	E	$E \to TE'$			$E \to TE'$		
2	E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
3	T	$T \to FT'$			$T \to FT'$		
4	T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
5	F	$F \to id$			$F \to (E)$		

id+id*id$

stack	Input	Action
$	id+id*id$	$E \to TE'$
$ E'T'	id+id*id$	$T \to FT'$
$ E'T'F	id+id*id$	$F \to id$
$ E'T'id	id+id*id$	$T' \to \epsilon$
$ E'T'	+id*id$	$E' \to +TE'$
$ E'	+id*id$	$T \to FT'$
$ E'T+	id*id$	$F \to id$
$ E'T'F	id*id$	
$ E'T'id	id*id$	$T' \to *FT'$
$ E'T'	*id$	$F \to id$
$ E'T'F*	*id$	$T' \to \epsilon$
$ E'T'id	id$	$E' \to \epsilon$
$ E'	$	Accept
$	$	


Input:
1
+
1
$
Stack:
6
3
0
Parser
Output
Action table
Goto table

Step	Parse Stack $_{state}$ [Symbol$_{state}$]*	Look Ahead	Unscanned	Parser Action	Grammar Rule	Next State
0	$_0$	*id*	* 2 + 1	shift		9
1	$_0$ *id*$_9$	*	2 + 1	reduce	Value → *id*	7
2	$_0$ Value$_7$	*	2 + 1	reduce	Products → Value	4
3	$_0$ Products$_4$	*	2 + 1	shift		5
4	$_0$ Products$_4$ *$_5$	*int*	+ 1	shift		8
5	$_0$ Products$_4$ *$_5$ *int*$_8$	+	1	reduce	Value → *int*	6
6	$_0$ Products$_4$ *$_5$ Value$_6$	+	1	reduce	Products → Products * Value	4
7	$_0$ Products$_4$	+	1	reduce	Sums → Products	1
8	$_0$ Sums$_1$	+	1	shift		2
9	$_0$ Sums$_1$ +$_2$	*int*	*eof*	shift		8
10	$_0$ Sums$_1$ +$_2$ *int*$_8$	*eof*		reduce	Value → *int*	7
11	$_0$ Sums$_1$ +$_2$ Value$_7$	*eof*		reduce	Products → Value	3
12	$_0$ Sums$_1$ +$_2$ Products$_3$	*eof*		reduce	Sums → Sums + Products	1
13	$_0$ Sums$_1$	*eof*		accept		

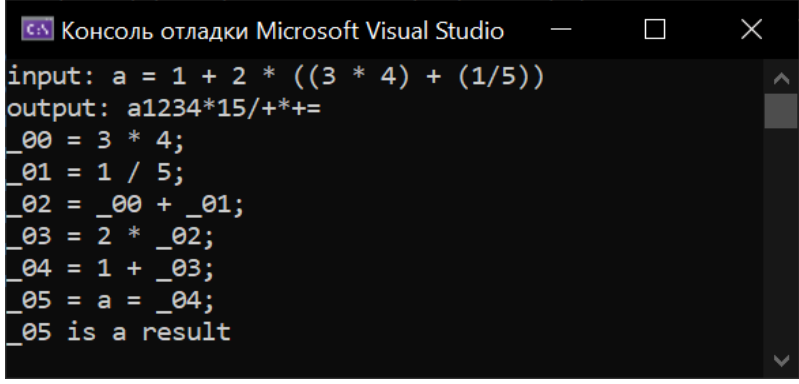
№	Токен	Стек	Вихідний рядок
1	`12`		`12`
2	`+`	`+`	`12`
3	`2`	`+`	`12 2`
4	`*`	`+ *`	`12 2`
5	`(`	`+ * (`	`12 2`
6	`(`	`+ * ( (`	`12 2`
7	`3`	`+ * ( (`	`12 2 3`
8	`*`	`+ * ( (*`	`12 2 3`
9	`4`	`+ * ( (*`	`12 2 3 4`
10	`)`	`+ * (`	`12 2 3 4 *`
11	`+`	`+ * (+`	`12 2 3 4 *`
12	`(`	`+ * (+ (`	`12 2 3 4 *`
13	`10`	`+ * (+ (`	`12 2 3 4 * 10`
14	`/`	`+ * (+ ( /`	`12 2 3 4 * 10`
15	`5`	`+ * (+ ( /`	`12 2 3 4 * 10 5`
16	`)`	`+ * (+`	`12 2 3 4 * 10 5 /`
17	`)`	`+ *`	`12 2 3 4 * 10 5 / +`
		`+`	`12 2 3 4 * 10 5 / +`
			`12 2 3 4 * 10 5 / + * +`

Маємо вираз: `12 + 2 * ((3 * 4) + (10 / 5))`

Вираз у польському інверсному записі: `12 2 3 4 * 10 5 / + * +`

Порядок дій над ним буде такий:

Крок	Елемент	Стек
1	12	12
2	2	2 12
3	3	3 2 12
4	4	4 3 2 12
5	*	**12** 2 12
6	10	10 12 2 12
7	5	5 10 12 2 12
8	/	**2** 12 2 12
9	+	**14** 2 12
10	*	**28** 12
11	+	**40**



In compiler design, **static single assignment form** (often abbreviated as **SSA form** or simply **SSA**) is a type of intermediate representation (IR) where each variable is assigned exactly once. SSA is used in most high-quality optimizing compilers for imperative languages, including LLVM, the GNU Compiler Collection, and many commercial compilers.

There are efficient algorithms for converting programs into SSA form. To convert to SSA, existing variables in the original IR are split into versions, new variables typically indicated by the original name with a subscript, so that every definition gets its own version. Additional statements that assign to new versions of variables may also need to be introduced at the join point of two control flow paths. Converting from SSA form to machine code is also efficient.

SSA makes numerous analyses needed for optimizations easier to perform, such as determining use-define chains, because when looking at a use of a variable there is only one place where that variable may have received a value. Most optimizations can be adapted to preserve SSA form, so that one optimization can be performed after another with no additional analysis. The SSA based optimizations are usually more efficient and more powerful than their non-SSA form prior equivalents.

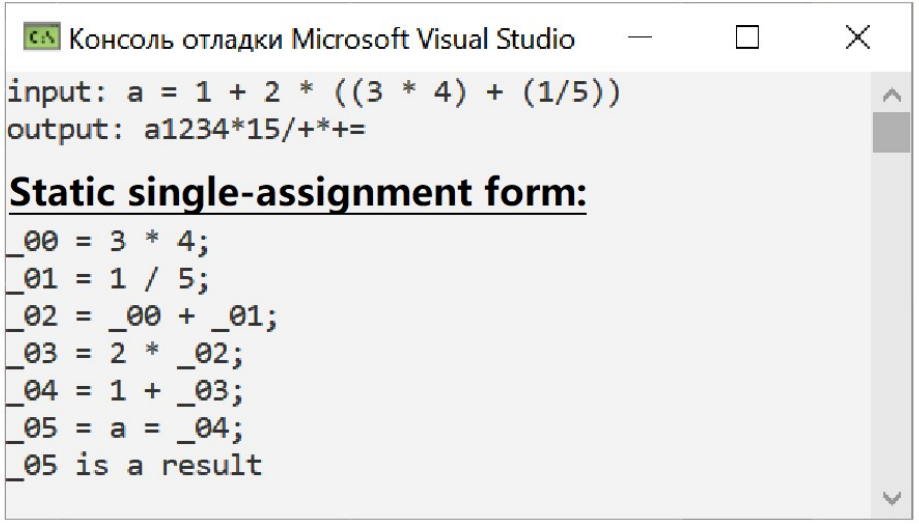
In functional language compilers, such as those for Scheme and ML, continuation-passing style (CPS) is generally used. SSA is formally equivalent to a well-behaved subset of CPS excluding non-local control flow, so optimizations and transformations formulated in terms of one generally apply to the other. Using CPS as the intermediate representation is more natural for higher-order functions and interprocedural analysis. CPS also easily encodes call/cc, whereas SSA does not.

Маємо вираз: `1 + 2 * ((3 * 4) + (1 / 5))`

Вираз у польському інверсному записі: `1 2 3 4 * 1 5 / + * +`

Порядок дій над ним буде такий:

Крок	Елемент	Стек
1	1	1
2	2	2 1
3	3	3 2 1
4	4	4 3 2 1
5	*	**12** 2 1
6	1	1 12 2 1
7	5	5 1 12 2 1
8	/	**0** 12 2 1
9	+	12 2 1
10	*	**24** 1
11	+	**25**



Контрольне завдання

Створіть поетапно зворотний польський запис та SSA-форму обчислень для виразу, що заданий в лабораторній роботі №1.

1.	$X=A_2+C_1-D_2/2+K$	1254021
2.	$X=A_4+C_2+D_1*4-K$	202
3.	$X=K-B_2*8+C_2-E_1$	37788663
4.	$X=A_4+C_1-D_4/8+K$	45694
5.	$X=B_4-A_2*2-E_2+K$	505
6.	$X=K+B_2/4-D_2*4-E_1$	6DD02316
7.	$X=A_4/2-4*(D_1+E_2-K)$	717
8.	$X=A_4-B_2+K-D_2/2+8*B_2$	88
9.	$X=4*B_2-C_2+D_4/4$	29
10.	$X=A_4-B_4/2+K+E_2*4$	2310
11.	$X=(A_4-B_2-K)*2+E_4/4$	311
12.	$X=K+B_4/2-4*F_2-E_1$	7055E0AC
13.	$X=A_2/2+8*(D_1+E_2-K)$	2513
14.	$X=A_4-B_1-K-D_2/2+4*B_1$	614
15.	$X=A_4+(4*C_2)-D_4/2+K$	4569600F

15.	$X=A_4+(4*C_2)-D_4/2+K$	4569600F
16.	$X=A_4/4+C_2-D_1*2+K$	616
17.	$X=A_4-K+C_4/2-E_1*8$	1017
18.	$X=4*(B_2-C_1)+D_2/4+K$	56987018
19.	$X=A_2*4+C_1-D_4/2+K$	4019
20.	$X=K+B_4/4-D_1*2-E_2$	18932020
21.	$X=A_4/8+2*(D_2-E_1+K)$	21
22.	$X=K-B_1-C_1-D_2/2+4*B_1$	45781022
23.	$X=A_2*8-C_1+D_4/2+K$	7AA02023
24.	$X=K-B_2/2+D_4+E_2*4$	74569024
25.	$X=(K-B_2-C_1)*2+E_4/4$	2B05025
26.	$X=A_2+K+C_2/2-E_1*8$	6C26
27.	$X=A_2*4+(K-E_1*4)$	A77627
28.	$X=K+B_4/2+D_2-E_2/4$	3FF28
29.	$X=K-B_1*4+D_2-F_2/2$	12A0C029
30.	$X=K+B_4-D_2/2+E_1*4$	25630