$S \to AB \mid \epsilon$ — ①

$A \to aB$ — ②

$B \to Sb$ — ③

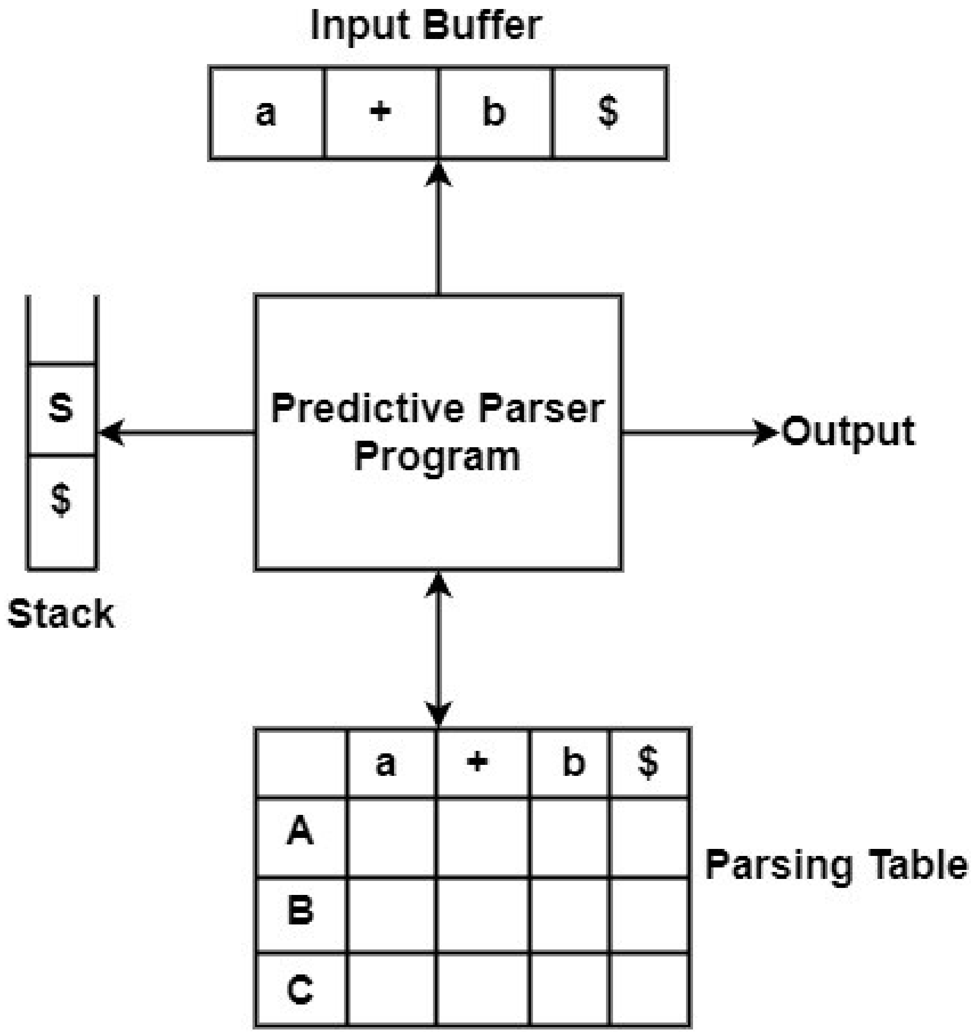
abb

left most Derivation

S	
AB	$S \to AB$
$\boxed{aB}\,B$	$A \to aB$
$a\boxed{Sb}\,B$	$B \to Sb$
$\to$ $ab\,B$	$S \to \epsilon$
$ab\boxed{Sb}$	$B \to Sb$
$\boxed{abb}$	$S \to c$

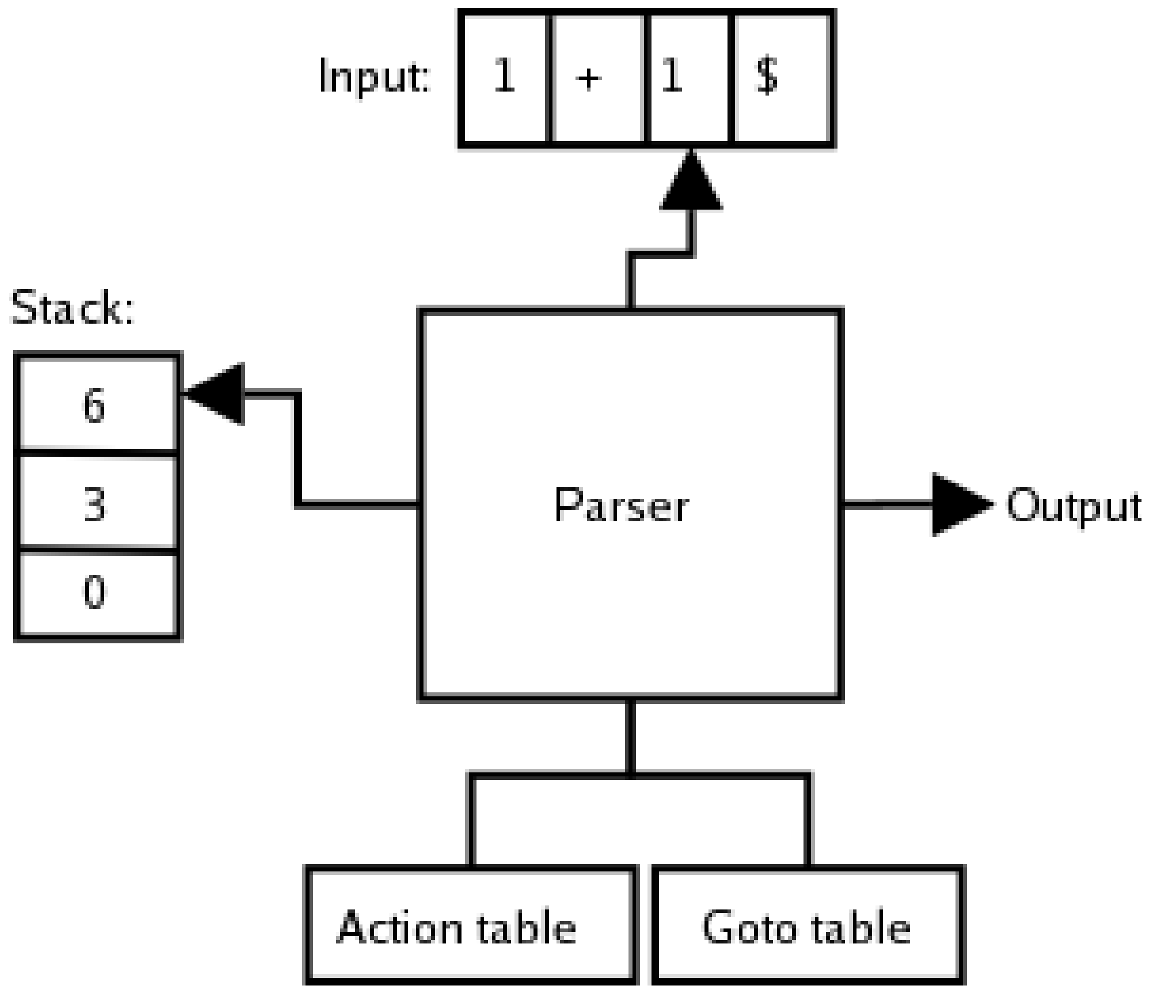
Right most Derivation

S	
AB	$S \to AB$
$A\boxed{Sb}$	$B \to Sb$
$A\boxed{\epsilon}b$	$S \to \epsilon$
$\boxed{aB}\,b$	$A \to aB$
$a\boxed{Sb}b$	$B \to SB$
$\boxed{abb}$	$S \to \epsilon$



Predictive Parser

	STACK	INPUT	ACTION	OUTPUT
1	$\$S$	$bbbbaab\$$	LL produce $[S \longrightarrow bBab]$	$[S \longrightarrow bAab]$
2	$\$baAb$	$bbbbaab\$$	LL shift b	
3	$\$baA$	$bbbaab\$$	— start the LR parser for A	
4	$\$baq_0$	$bbbaab\$$	LR shift b	
5	$\$baq_0bq_1$	$bbaab\$$	LR reduce on $[B \longrightarrow b]$	
6	$\$baq_0Bq_1$	$bbaab\$$	— stop the LR parser for A	$[A \longrightarrow BbA][B \longrightarrow b]$
7	$\$baAb$	$bbaab\$$	LL shift b	
8	$\$baA$	$baab\$$	— start the LR parser for A	
9	$\$baq_0$	$baab\$$	LR shift b	
10	$\$baq_0bq_1$	$aab\$$	— stop the LR parser for A	$[A \longrightarrow ba]$
11	$\$baa$	$aab\$$	LL shift a	
12	$\$ba$	$ab\$$	LL shift a	
13	$\$b$	$b\$$	LL shift b	
14	$\$$	$\$$	LL accpet	


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:

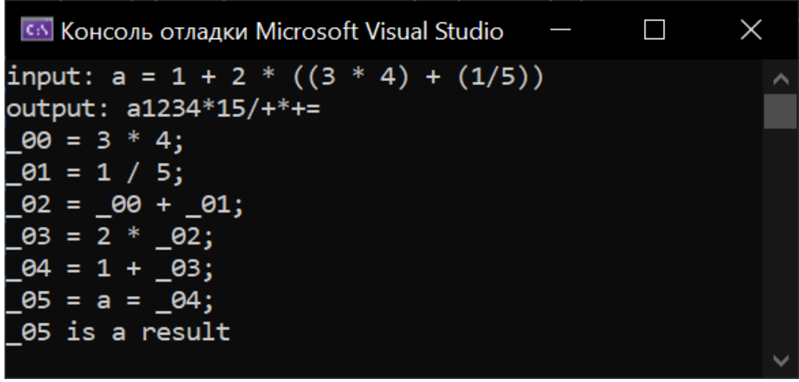
Крок	Елемент	Стек
1	12	12
2	2	2 12
3	3	3 2 12
4	4	4 3 2 12
5	*	**12** 2 12
6	10	10 12 2 12
7	5	5 10 12 2 12
8	/	**2** 12 2 12
9	+	**14** 2 12
10	*	**28** 12
11	+	**40**

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.

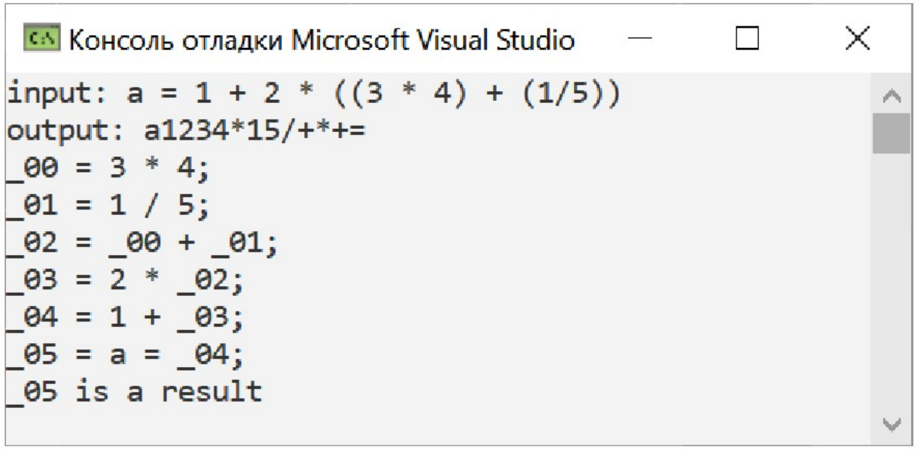


```
input: a = 1 + 2 * ((3 * 4) + (1/5))
output: a1234*15/+*+=
_00 = 3 * 4;
_01 = 1 / 5;
_02 = _00 + _01;
_03 = 2 * _02;
_04 = 1 + _03;
_05 = a = _04;
_05 is a result
```

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:



Консоль отладки Microsoft Visual Studio

```
input: a = 1 + 2 * ((3 * 4) + (1/5))
output: a1234*15/+*+=
_00 = 3 * 4;
_01 = 1 / 5;
_02 = _00 + _01;
_03 = 2 * _02;
_04 = 1 + _03;
_05 = a = _04;
_05 is a result
```

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**