

left most Derivation

$\begin{array}{c} S \\ | \\ AB \\ | \\ \boxed{B} B \\ | \\ a \boxed{sb} B \\ | \\ ab B \\ | \\ ab \boxed{sb} \\ | \\ abb \end{array}$

$S \rightarrow ABCE$ — ①
 $A \rightarrow aB$ — ②
 $B \rightarrow sb$
 "abb"

$S \rightarrow AB$

$A \rightarrow aB$

$B \rightarrow sb.$

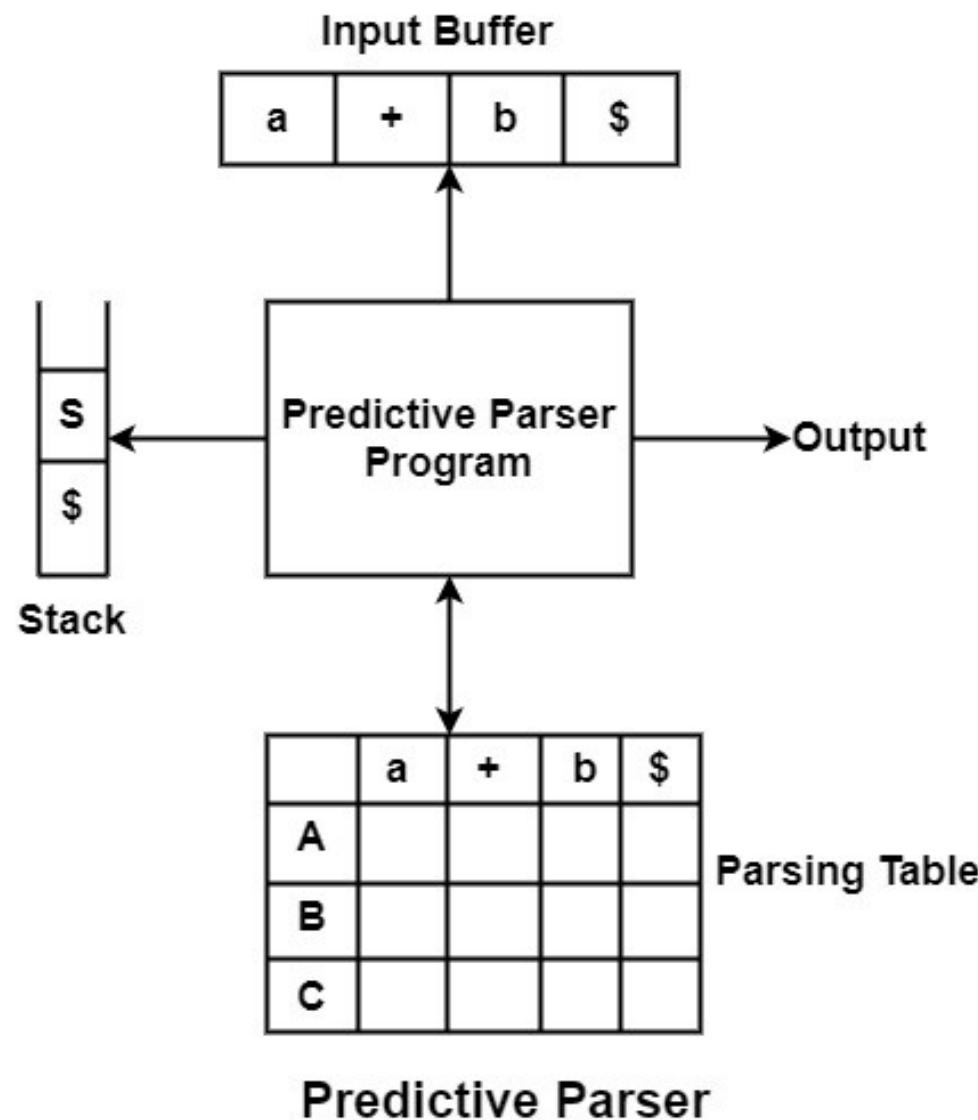
$S \rightarrow E$

$B \rightarrow sb$

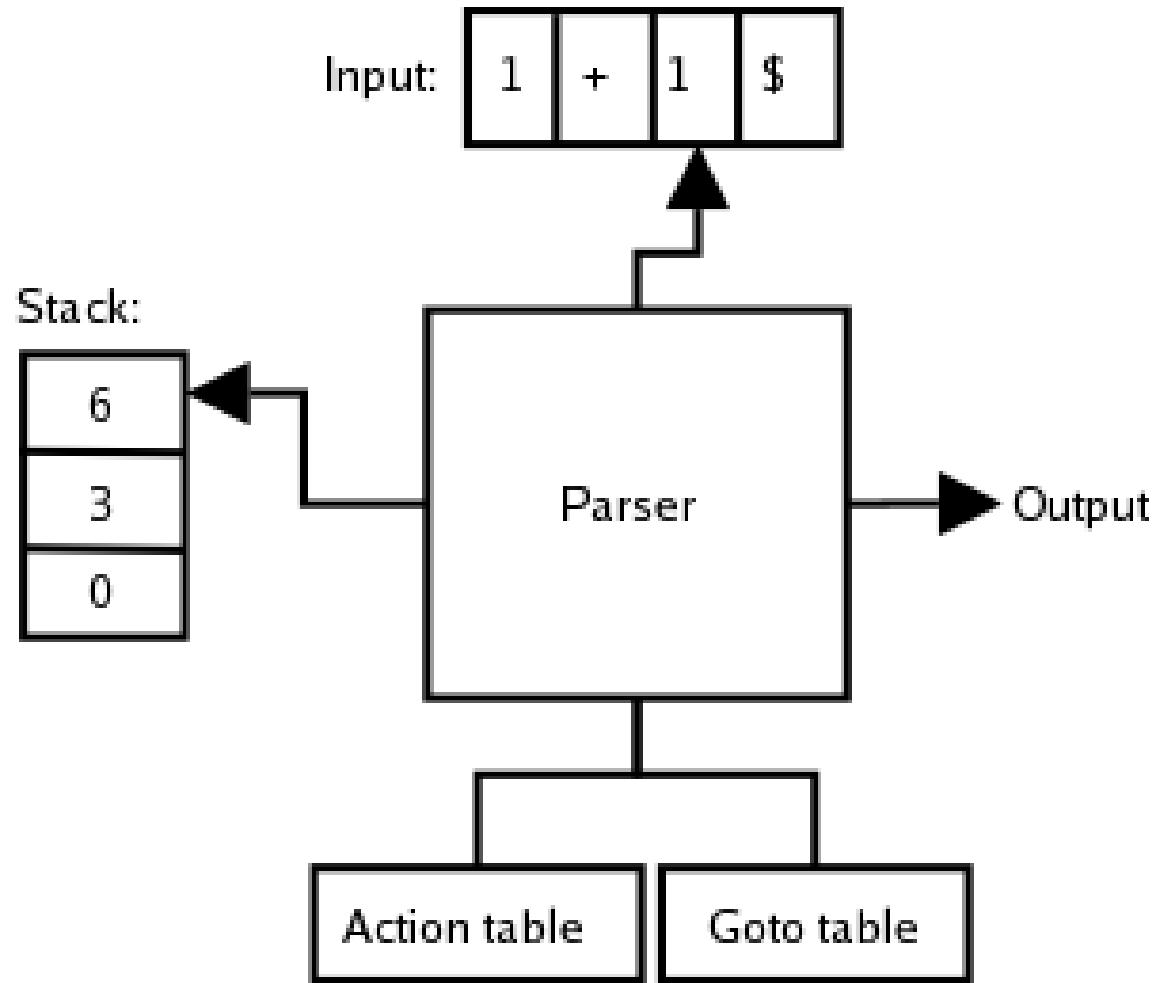
$S \rightarrow C$

Right most Derivation

$\begin{array}{c} S \\ \\ AB \\ \\ A \boxed{sb} \\ \\ A \boxed{\epsilon} b \\ \\ \boxed{AB} b \\ \\ a \boxed{sb} b \\ \\ \boxed{abb} \end{array}$	$S \rightarrow AB$
$B \rightarrow sb.$	$S \rightarrow E$
$A \rightarrow aB$	$S \rightarrow E$
$B \rightarrow SB.$	$S \rightarrow E$
$S \rightarrow E$	$S \rightarrow E$



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



Step	Parse Stack state [Symbol _{state}]*	Look Ahead	Unscanned	Parser Action	Grammar Rule		Next State
0	0	<i>id</i>	* 2 + 1	shift			9
1	0 <i>id</i> ₉	*	2 + 1	reduce	Value → <i>id</i>		7
2	0 Value ₇	*	2 + 1	reduce	Products → Value		4
3	0 Products ₄	*	2 + 1	shift			5
4	0 Products ₄ * ₅	<i>int</i>	+ 1	shift			8
5	0 Products ₄ * ₅ <i>int</i> ₈	+	1	reduce	Value → <i>int</i>		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	<i>int</i>	<i>eof</i>	shift			8
10	0 Sums ₁ + ₂ <i>int</i> ₈	<i>eof</i>		reduce	Value → <i>int</i>		7
11	0 Sums ₁ + ₂ Value ₇	<i>eof</i>		reduce	Products → Value		3
12	0 Sums ₁ + ₂ Products ₃	<i>eof</i>		reduce	Sums → Sums + Products		1
13	0 Sums ₁	<i>eof</i>		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:

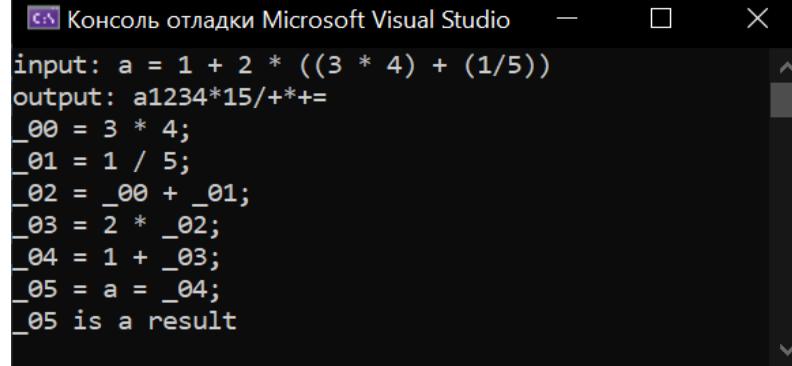
Крок	Елемент	Стек
1	12	12
2	2	2 12
3	3	3 2 12
4	4	4 3 2 12
5	*	12 2 12
6	10	10 12 2 12
7	5	5 10 12 2 12
8	/	2 12 2 12
9	+	14 2 12
10	*	28 12
11	+	40

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.



The screenshot shows the 'Консоль отладки Microsoft Visual Studio' (Debug Console) window. It displays the input C-like expression 'a = 1 + 2 * ((3 * 4) + (1/5))' and the generated SSA code. The SSA code consists of six lines, each starting with a label '_00', '_01', etc., followed by an assignment operation. The final line '_05 is a result' indicates the target variable for the assignment.

```
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/+*+=  
  
Static single-assignment form:  
_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	*	<u>12</u> 2 1
6	1	1 <u>12</u> 2 1
7	5	5 1 <u>12</u> 2 1
8	/	<u>0</u> <u>12</u> 2 1
9	+	<u>12</u> 2 1
10	*	<u>24</u> 1
11	+	<u>25</u>