Syntax Analysis

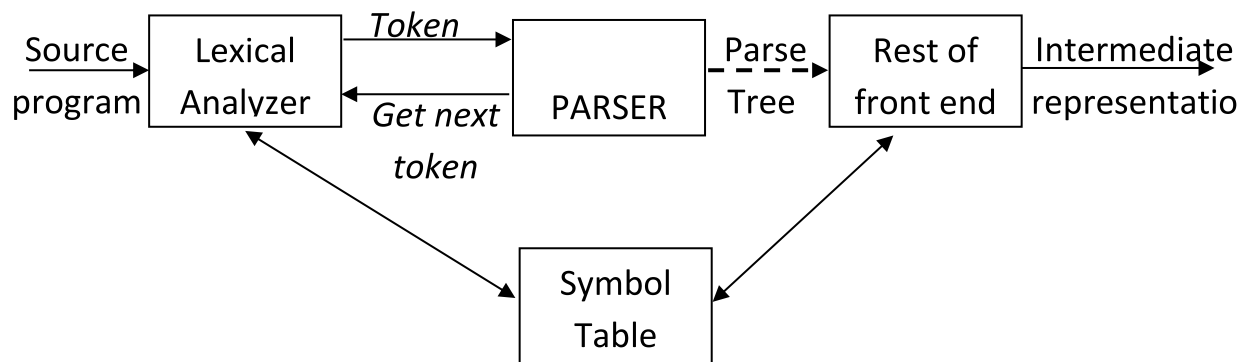
Introduction

Syntax analysis or parsing is the second phase of a compiler. In this topic, we shall learn the basic concepts used in the construction of a parser.

As shown in topics (2 and 3), a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses **context-free grammar *CFG***, which is recognized by pushdown automata.

Syntax Analyzers

A syntax analyzer (or parser) takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code *token stream* against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



Position of Parser in Compiler Model

This way, the parser accomplishes two tasks:

- Parsing the code, looking for errors and
- Generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later.

Example: if a source program contains the following expression:

A + /B

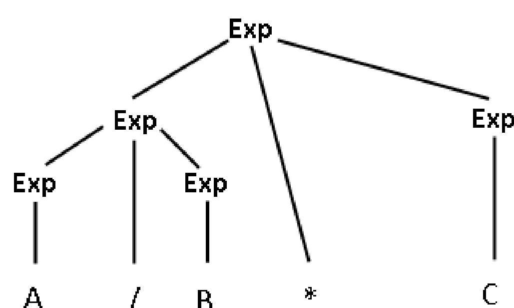
Then after **lexical analysis** this expression might appear to the syntax analyzer as the token sequence

id1 + / id2

On seeing the /, the syntax analyzer should detect an **error** situation, because the presence of these two adjacent operators violates the formation rules of a Pascal expression.

Example: identifying which parts of the token stream should be grouped together: **A/B*C**

Parse Tree:



Syntactic Errors:

Syntactic errors include ***misplaced semicolons*** or extra or ***missing braces***; that is, '((" or ")." As another example, in C++, the appearance of a ***case*** statement without an enclosing ***switch*** is a syntactic error. For examples:

1. x= z+*y;
2. for (i=1; i++; i<=10)
3. cin<<x;
4. and more….

Context-Free Grammars (CFG)

Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars. For example, we might have a conditional statement defined by a rule such as

If $\mathbf{S_1}$ and $\boldsymbol{S_2}$ are statements and E is an expression, then

"If E **then** S_1 **else** S_2" is a statement.

This form of conditional statement cannot be specified using the notation regular expressions.

Could also express as: ***stmt*** ⟶ **if *(expr) stmt* else *stmt***

Such a role is called syntactic variables, ***stmt*** to denote the class of statements and ***expr*** the class of expressions.

Components of Context-Free Grammars (CFG)

A context-free grammar has four components:

- A set of **non-terminals (variables** *V)*. Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** Σ. Terminals are the basic symbols from which strings are formed.
- A set of **productions** *P (also called **rules**)*. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, followed by an arrow, followed by a string of non-terminals and/or **terminals**, called the right side of the production.
- One of the non-terminals is designated as the start symbol ***S***; from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal *initially the start symbol* by the right side of a production, for that non-terminal.

Example: The grammar with the following productions defines simple arithmetic expressions.

expr → ***expr op expr***

expr → **(*expr*)**

expr → – ***expr***

expr → **id**

op → +

op → –

op → *

op → /

op → ↑

In this grammar, the terminal symbols are **id** + **-** * / ↑ **( )**

The nonterminal symbols are ***expr*** and ***op,*** and ***expr*** is the start symbol.

The above grammar can be rewriting by using **shorthands** as:

E → ***EAE*** | **(*E*)** | **-*E*** | **id**

A → + | **-** | * | / | ↑

where ***E*** and ***A*** are non-terminals, with ***E*** the start symbol. The remaining symbols are terminals.

Derivations and Parse Trees

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options:-

1. Left-most Derivation

If the input is scanned and replaced with production rules from left to right, it is known as left -most derivation.

2. Right-most Derivation

If the input is scanned and replaced with production rules from right to left, it is known as right-most derivation.

Example-1

Production rules: $E \rightarrow E + E|E * E\ |id \quad \ldots\ldots\ldots\ldots.(1.1)$

Input string: id + id * id

The ***left-most derivation*** is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

Notice that the left-most side non-terminal is always processed first.

The ***right-most derivation*** is:

E → E + E
E → E + E * E
E → E + E * id
E → E + id * id
E → id + id * id

Example-2: consider the following grammar for arithmetic expressions:

$E \longrightarrow E + E \,|E * E\,|\,(E)\,|-E\,|\mathbf{id}$ ……………… (**1**.2).

The nonterminal ***E*** is an abbreviation for expression. The production

$E \longrightarrow -E$ signifies that an expression preceded by minus sign is also an expression. In the simplest case can replace single ***E*** by -***E***. We can describe this action by writing:

$E \longrightarrow -E$ which is read as "***E*** derives -***E***"

We can take a single E and repeatedly apply productions in any order to obtain a sequence of replacements. For example,

$E \longrightarrow -E \longrightarrow -(E) \longrightarrow -(\mathbf{id})$

We call such a sequence of replacements a **derivation** of **- (id)** from ***E***. This derivation provides a proof that one particular instance of an expression is the string **- (id)**.

Example: The string **-(id + id)** is a sentence of grammar (1.2) because there is the derivation

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E)$$
$$\Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Each **interior node** of the parse tree is labeled by some **nonterminal** ***A***, and the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this ***A*** was replaced in the derivation.

The **leaves** of the parse tree are labeled by **nonterminal** or **terminals** and, read from left to right.

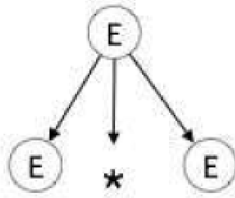
Example-1: Consider the grammar as in (1.1)

We take the left-most derivation of a + b * c

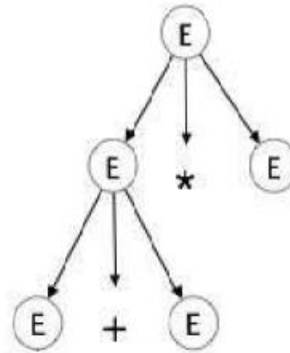
The left-most derivation is:

E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id

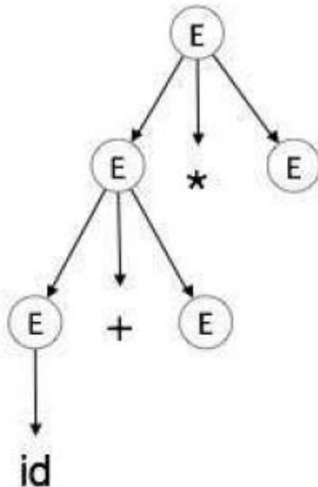
Step 1: E → E * E



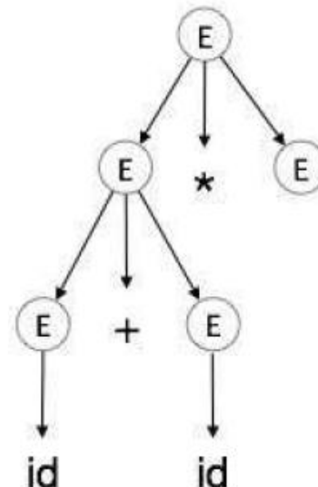
Step-2: E → E + E * E



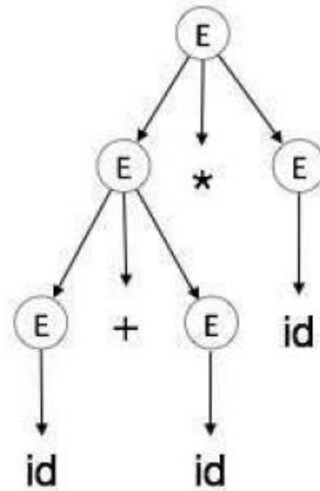
Step-3: E → id + E * E



Step-4: E → id + id * E



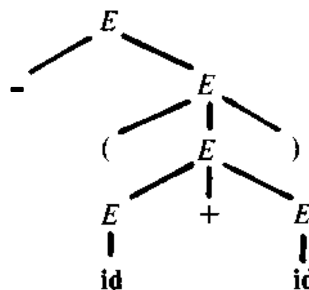
Step-5: E → id + id * id



In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

Example 2: Consider the grammar as in (1.2), then the parse tree for -(id+id) that implied by the derivation is:

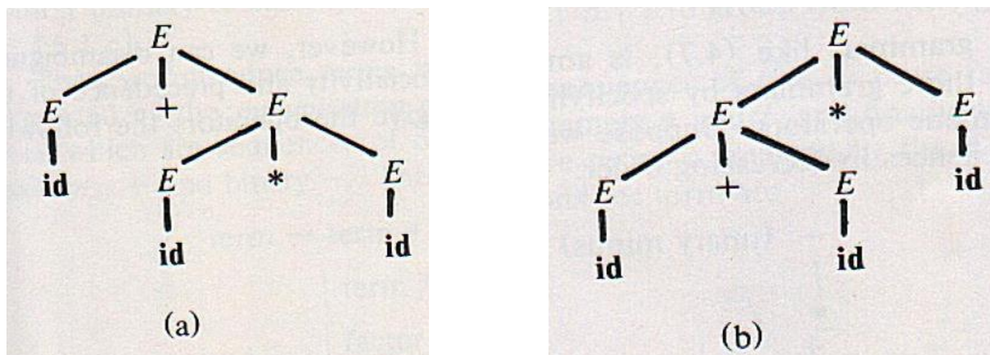


Example: Let us again consider the arithmetic expression grammar (1.2), with which we have been dealing. The sentence **id + id * id** has the two distinct leftmost derivations:

a) $E \rightarrow E + E \rightarrow id + E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id$

b) $E \rightarrow E * E \rightarrow E + E * E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id$

With the two corresponding parse tree shown in figure below:



Two parse trees for **id + id * id**

Ambiguity

A grammar G is said to be ambiguous if it has more than one parse tree ***left or right derivation*** for at least one string. In this type, we cannot uniquely determine which parse tree to select for a sentence. The grammar in the previous example is ambiguous.

Example: Consider the following grammar for arithmetic expressions involving +, -, *, /, and↑ (exponentiation)

$$E \longrightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid -E \mid \textbf{id}$$

This grammar is ambiguous.

Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following ***associativity*** and ***precedence*** constraints.

Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the ***associativity*** of those operators. If the operation is left-associative, then the operand will be taken by the left

operator or if the operation is right-associative, the right operator will take the operand.

Example: Operations such as **Addition**, **Multiplication**, **Subtraction**, and **Division** are left associative. If the expression contains: id op id op id
It will be evaluated as: (id op id) op id; for example: id+id+id
Operations like Exponentiation are right associative.

Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3*4 can have two different parse trees. By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically * *multiplication* has precedence over + *addition*, so the expression 2+3*4 will always be interpreted as: 2+(3*4)

These methods decrease the chances of ambiguity in a language or its grammar.

Generating of Context-Free Grammar

Every language that can be described by a regular expression or regular language can also be described by Context-Free Grammar.

General Rules:

- a   S-> a
- a + b   S-> a|b
- ab   S-> aA , A-> b
- a*   S-> aS|ϵ
- a^+ S-> aS | a
- $(a + b)^*$ S-> aS | bS | ϵ

- $(a + b)^+$ S-> aS | bS | a | b
- $(ab)^*$ S-> aA | ϵ ; A-> bS
- $(ab)^+$ S-> aA ; A-> bS | b

Example: Design CFG that accept the RE =a^* ; L={ϵ, a, aa, …}

$S \longrightarrow aS \mid \in$

Example: Design CFG that accept the RE = $(a+b)^*$; L={ ϵ, a, b, ab, …}

$S \longrightarrow aS \mid bS \mid \in$

Example: Design CFG that accept the RE = $(a+b)^*$ a $(a+b)^*$

Let X = $(a+b)^*$ and Y = a; So

$S \longrightarrow XYX$

$X \longrightarrow aX \mid bX \mid \in$

$Y \longrightarrow a$

Example: Design CFG that accept the RE = $a^* + a(a+b)^*$

Let X=a^*, and Y= $a(a+b)^*$ then

$S \longrightarrow X \mid Y$

$X \longrightarrow aX \mid \epsilon$

$Y \longrightarrow aZ$; where Z = $(a+b)^*$

$Z \longrightarrow aZ \mid bZ \mid \epsilon$

Example: Design CFG that accept the RE = ***a*** (***a***|***b***)****b***

$S \longrightarrow aAb$

$A \longrightarrow aA \mid bA \mid \in$

Example: Design CFG that accept the RE = (***a***|***b***)****abb***

$S \longrightarrow aS \mid bS \mid aX$; $X \longrightarrow bY$

$Y \longrightarrow bZ$; $Z \longrightarrow \in$

Example: Design CFG that accept the $a^n b^n$ where $n \geq 1$.

$S \longrightarrow aXb$; $X \longrightarrow aXb \mid \in$

Example: Design CFG for ***Singed Integer*** number.

$S \longrightarrow XD$

$X \longrightarrow + \mid -$

$D \longrightarrow 0D \mid 1D \mid 2D \mid 3D \mid 4D \mid 5D \mid 6D \mid 7D \mid 8D \mid 9D \mid \in$

H.W: Design CFG that accept the followings:-

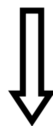
1. RE= $(a+b)^* (a+b) + (ab+ba)^* (a+b)^*$ * $B(a+b) +a$
2. L={$a^n b^m$; $n \geq 1, m \geq 0$ }
3. A CFG for all binary strings with an even number of 0's.
4. A CFG for the regular language corresponding to the RE $00^* 11^*$.
5. L={$a^n b^m$; $n \neq m$ and $n, m \geq 1$}
6. L={$a^n b^m$; (n+m) is even}
7. L={$a^n b^m$; $n \geq 3, m \geq 2$}
8. L={$a^n b^n$; $n \geq 0$ }
9. L={$a^n b^n c^m$; $n \geq 1, m \geq 0$ }
10. L={$a^n b^n c^m d^m$; $n, m \geq 0$ }
11. L={$a^n b^m c^m d^n$; $n, m \geq 1$ }
12. L={$a^{m+n} b^n c^m$; $n, m \geq 1$ }
13. L={$a^n b^m$; $n \neq m$ and $n, m \geq 1$}
14. RE= ab*ab
15. RE = $a^+ b^*$

Elimination of Left Recursion

A grammar is ***Left Recursive*** if it has a nonterminal A such that there is a derivation $A \xrightarrow{+} A\alpha$ for some string α. **Top-Down Parsing** ***methods cannot handle left recursive grammars***, so a transformation that eliminates left recursion is needed.

In the following example, we show how the left recursion pair of productions $A \longrightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$A \longrightarrow A\alpha \mid \beta$



$A \longrightarrow \beta A'$

$A' \longrightarrow \alpha A' \mid \in$

without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

Example1: Consider the following grammar for arithmetic expressions.

$E \longrightarrow E+T \mid T$

Solution: Let A=E, α=+T, β=T then

$E \longrightarrow T\,\overline{E}$

$\overline{E} \longrightarrow +T\,\overline{E} \mid \epsilon$

Example: Consider the following grammar for arithmetic expressions.

$E \longrightarrow E+T \mid T$

$T \longrightarrow T*F \mid F$

$F \longrightarrow (E) \mid \mathbf{id}$

Eliminating the immediate ***left recursion*** (productions of the form $A \longrightarrow A\alpha$) to the productions for E and then for T, we obtain

$E \longrightarrow T\overline{E}$

$\overline{E} \longrightarrow +T\overline{E} \mid \in$

$T \longrightarrow F\overline{T}$

$\overline{T} \longrightarrow *F\overline{T} \mid \in$

$F \longrightarrow (E) \mid \mathbf{id}$

Note: No matter how many *A*-productions there are, we can eliminate immediate left recursion from them by the following technique. First, we group the *A*-productions as

$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots\ldots \mid \beta_n$

where no **β**, begins with an *A*. Then, we replace the *A*-productions by:

$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots\ldots\ldots \mid \beta_n A'$

$A' \longrightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots\ldots\ldots \mid \alpha_m A' \mid \in$

The nonterminal *A* generates the same strings as before but is no longer left recursive.

Note: This procedure eliminates all immediate left recursion from the *A* and *A*' productions, but it does not eliminate left recursion involving derivations of **two or more** steps (i.e. ***indirect recursive***).

For **Example**, consider the grammar:

$S \longrightarrow A\mathbf{a} \mid \mathbf{b}$

$A \longrightarrow A\mathbf{c} \mid S\mathbf{d} \mid \in$

The nonterminal *S* is left- recursive because $S \longrightarrow A\mathbf{a} \longrightarrow S\mathbf{da}$, but is not immediately left recursive. **Algorithm** in below will systematically eliminate left recursion from grammar.

Algorithm: Eliminating left recursion.

Input: Grammar ***G*** with no cycles or ε -productions.

Output: An equivalent grammar with no left recursion.

Method: Apply the algorithm below to ***G***. Note that the resulting non left-recursive grammar may have ε -productions.

Arrange the nonterminals in some order $A_1, A_2, \ldots\ldots A_n$.

for i: = 1 **to** n **do**

for j : = 1 **to** i-1 **do begin**

Replace each production of the form $A_i \longrightarrow A_i\gamma$ by the productions $A_i \longrightarrow \delta_1\gamma \mid \delta_2\gamma \mid \delta_3\gamma \mid \ldots\ldots \mid \delta_k\gamma$,

Where $A_j \longrightarrow \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots\ldots \mid \delta_k$ are all the current A_j-productions;

Eliminate the immediate left recursion among A_j-productions

End

Let us apply this procedure to previous grammar. We substitute the ***S***-productions in $A \longrightarrow Sd$ to obtain the following ***A***-productions.

$A \longrightarrow Ac \mid Aad \mid bd \mid \in$

Eliminating the immediate left recursion among the ***A***-productions yields the following grammar.

$S \longrightarrow Aa \mid b$

$A \longrightarrow bd\bar{A} \mid \bar{A}$

$\bar{A} \longrightarrow c\,\bar{A} \mid ad\bar{A} \mid \in$

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for **predictive parsing**. The basic idea is that when it is not clear which of **two** alternative productions to use to expand a nonterminal *A*, we may be able to rewrite the, *A*-productions to defer the decision until we have seen enough of the input to make the right choice. For example, if we have the two productions

if $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two *A*-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand *A* to $\alpha\beta_1$or to $\alpha\beta_2$. However, we may defer the decision by expanding *A* to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or to β_2. That is left-factored.

$$A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \quad \Rightarrow \quad \begin{aligned} A &\longrightarrow \alpha A' \\ A' &\longrightarrow \beta_1 \,/\, \beta_2 \end{aligned}$$

Algorithm : Left factoring a grammar

Input: Grammar ***G***.

Output: An equivalent left-factored grammar.

Method: For each nonterminal *A* find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i.e., there is a nontrivial common prefix, replace all the *A* productions $A \longrightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots\mid \alpha\beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α by

$$A \longrightarrow \alpha A' \mid \gamma$$

$$A' \longrightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots\ldots \mid \beta_n$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example:

$S \longrightarrow iEtS \mid iEtSeS \mid a$

$E \longrightarrow b$

Left-factored, this grammar becomes:

$S \longrightarrow iEtS\bar{S} \mid a$

$\bar{S} \longrightarrow eS \mid \in$

$E \longrightarrow b$

Example:

$A \longrightarrow aA \mid bB \mid ab \mid a \mid bA$

Solution:

$A \longrightarrow a\bar{A} \mid b\bar{B}$

$\bar{A} \longrightarrow A \mid b \mid \in$

$\bar{B} \longrightarrow B \mid A$

H.W

1) $S \longrightarrow S0S1S \mid 01$
2) $S \longrightarrow aSSbS \mid aSaSb \mid abbb \mid b$
3) $S \longrightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$