Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

1. G= (V, T, P, S)

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= {wcw^R | w € (a, b)*}

Production rules:

1. S → aSa
2. S → bSb
3. S → c

Now check that abbcbba string can be derived from the given CFG.

1. S ⇒ aSa
2. S ⇒ abSba
3. S ⇒ abbSbba

4. S ⇒ abbcbba

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Capabilities of CFG

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficient parser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be replaced with production rule.

Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:

Production rules:

1. S = S + S

2. S = S - S
3. S = a | b |c

Input:

```
a - b + c
```

The left-most derivation is:

1. S = S + S
2. S = S - S + S
3. S = a - S + S
4. S = a - b + S
5. S = a - b + c

Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

```
a - b + c
```

The right-most derivation is:

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

Example:

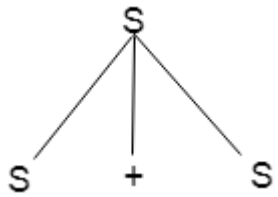
Production rules:

1. T= T + T | T * T
2. T = a|b|c

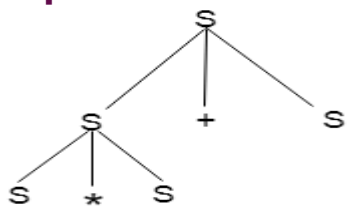
Input:

```
a * b + c
```

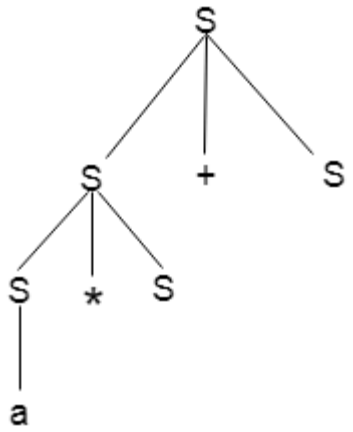
Step 1:



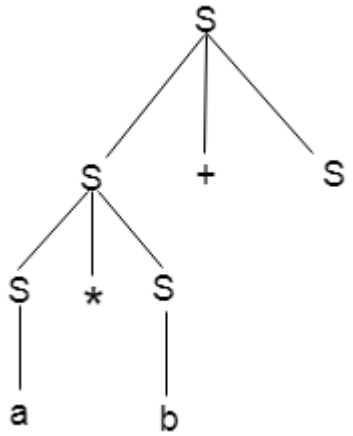
Step 2:



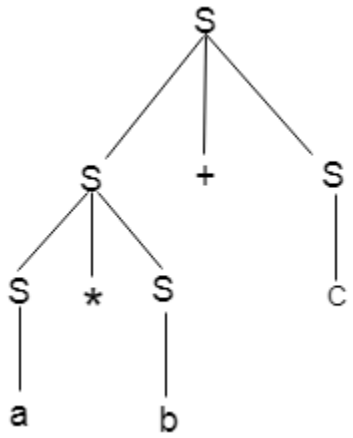
Step 3:



Step 4:



Step 5:



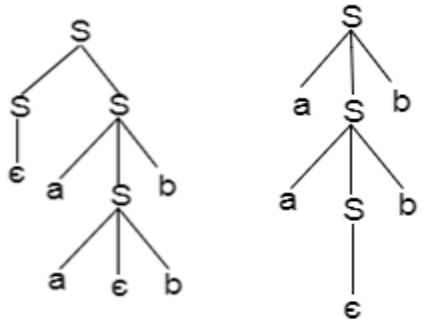
Ambiguity

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

1. S = aSb | SS
2. S = ∈

For the string aabb, the above grammar generates two parse trees:

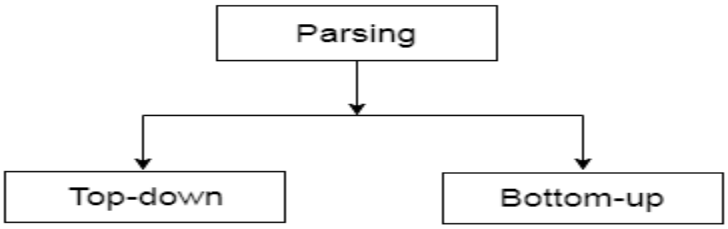


If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

Parser

Parser is a compiler that is used to break the data into smaller elements coming from lexical analysis phase.A parser takes input in the form of sequence of tokens and produces output in the form of parse tree.

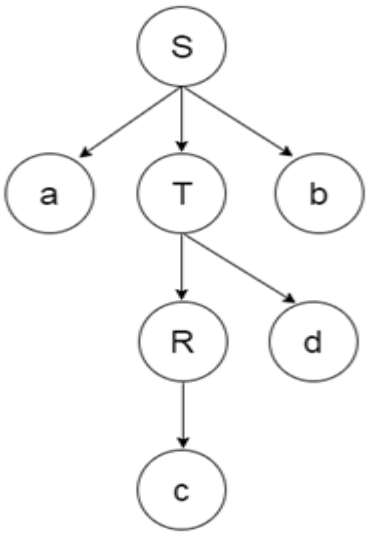
Parsing is of two types: top down parsing and bottom up parsing.



Top down paring

- The top down parsing is known as recursive parsing or predictive parsing.
- Bottom up parsing is used to construct a parse tree for an input string.
- In the top down parsing, the parsing starts from the start symbol and transform it into the input symbol.

Parse Tree representation of input string "acdb" is as follows:



Bottom up parsing

- Bottom up parsing is also known as shift-reduce parsing.
- Bottom up parsing is used to construct a parse tree for an input string.

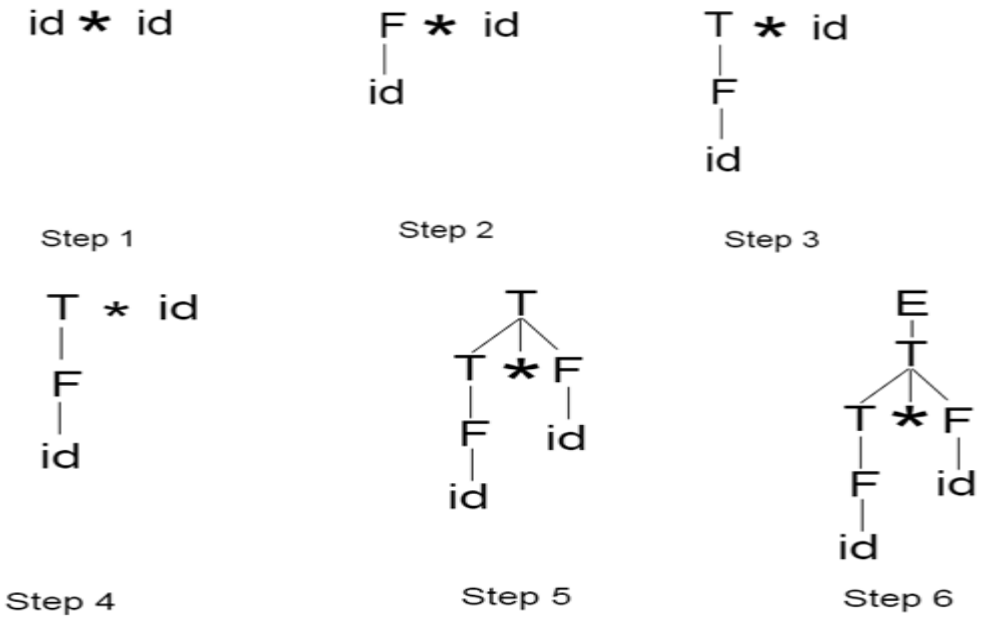
- In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse.

Example

Production

1. E → T
2. T → T * F
3. T → id
4. F → T
5. F → id

Parse Tree representation of input string "id * id" is as follows:



Bottom up parsing is classified in to various parsing. These are as follows:

1. Shift-Reduce Parsing
2. Operator Precedence Parsing
3. Table Driven LR Parsing

a. LR(1)

b. SLR(1)

c. CLR (1)

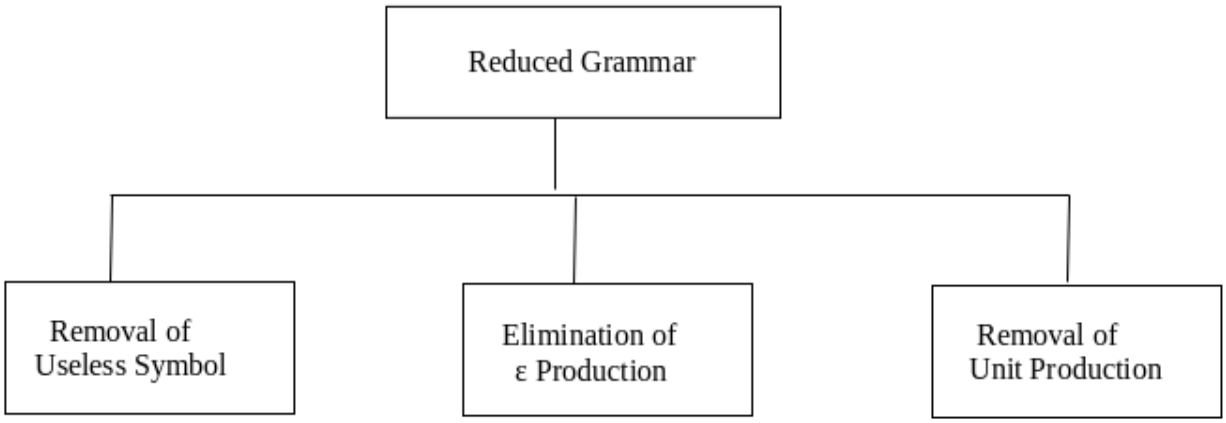
d. LALR(1)

Simplification of CFG

As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammar are not always optimized that means the grammar may consist of some extra symbols(non-terminal). Having extra symbols, unnecessary increase the length of grammar. Simplification of grammar means reduction of grammar by removing useless symbols. The properties of reduced grammar are given below:

1. Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
2. There should not be any production as X → Y where X and Y are non-terminal.
3. If ε is not in the language L then there need not to be the production X → ε.

Let us study the reduction process in detail./p>



Removal of Useless Symbols

A symbol can be useless if it does not appear on the right-hand side of the production rule and does not take part in the derivation of any string. That symbol is known as a

useless symbol. Similarly, a variable can be useless if it does not take part in the derivation of any string. That variable is known as a useless variable.

For Example:

1. T → aaB | abA | aaT
2. A → aA
3. B → ab | b
4. C → ad

In the above example, the variable 'C' will never occur in the derivation of any string, so the production C → ad is useless. So we will eliminate it, and the other productions are written in such a way that variable C can never reach from the starting variable 'T'.

Production A → aA is also useless because there is no way to terminate it. If it never terminates, then it can never produce a string. Hence this production can never take part in any derivation.

To remove this useless production A → aA, we will first find all the variables which will never lead to a terminal string such as variable 'A'. Then we will remove all the productions in which the variable 'B' occurs.

Elimination of ε Production

The productions of type S → ε are called ε productions. These type of productions can only be removed from those grammars that do not generate ε.

Step 1: First find out all nullable non-terminal variable which derives ε.

Step 2: For each production A → a, construct all production A → x, where x is obtained from a by removing one or more non-terminal from step 1.

Step 3: Now combine the result of step 2 with the original production and remove ε productions.

Example:

Remove the production from the following CFG by preserving the meaning of it.

1. S → XYX
2. X → 0X | ε

3. Y → 1Y | ε

Solution:

Now, while removing ε production, we are deleting the rule X → ε and Y → ε. To preserve the meaning of CFG we are actually placing ε at the right-hand side whenever X and Y have appeared.

Let us take

1. S → XYX

If the first X at right-hand side is ε. Then

1. S → YX

Similarly if the last X in R.H.S. = ε. Then

1. S → XY

If Y = ε then

1. S → XX

If Y and X are ε then,

1. S → X

If both X are replaced by ε

1. S → Y

Now,

1. S → XY | YX | XX | X | Y

Now let us consider

1. X → 0X

If we place ε at right-hand side for X then,

1. X → 0
2. X → 0X | 0

Similarly Y → 1Y | 1

Collectively we can rewrite the CFG with removed ε production as

1. S → XY | YX | XX | X | Y
2. X → 0X | 0
3. Y → 1Y | 1

Removing Unit Productions

The unit productions are the productions in which one non-terminal gives another non-terminal. Use the following steps to remove unit production:

Step 1: To remove X → Y, add production X → a to the grammar rule whenever Y → a occurs in the grammar.

Step 2: Now delete X → Y from the grammar.

Step 3: Repeat step 1 and step 2 until all unit productions are removed.

For example:

1. S → 0A | 1B | C
2. A → 0S | 00
3. B → 1 | A
4. C → 01

Solution:

S → C is a unit production. But while removing S → C we have to consider what C gives. So, we can add a rule to S.

1. S → 0A | 1B | 01

Similarly, B → A is also a unit production so we can modify it as

1. B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

1. S → 0A | 1B | 01
2. A → 0S | 00
3. B → 1 | 0S | 00
4. C → 01

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF.

However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S
2. S → a | aA | B
3. A → aBB | ε
4. B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:

1. S1 → S
2. S → a | aA | B
3. A → aBB
4. B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

1. S1 → S
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a

Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Chomsky's Normal Form (CNF)

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

For example:

1. G1 = {S → AB, S → c, A → a, B → b}
2. G2 = {S → aA, A → a, B → c}

The production rules of Grammar G1 satisfy the rules specified for CNF, so the grammar G1 is in CNF. However, the production rule of Grammar G2 does not satisfy the rules specified for CNF as S → aZ contains terminal followed by non-terminal. So the grammar G2 is not in CNF.

Steps for converting CFG into CNF

Step 1: Eliminate start symbol from the RHS. If the start symbol T is at the right-hand side of any production, create a new production as:

1. S1 → S

Where S1 is the new start symbol.

Step 2: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.

Step 3: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:

1. S → RA
2. R → a

Step 4: Eliminate RHS with more than two non-terminals. For example, S → ASB can be decomposed as:

1. S → RS
2. R → AS

Example:

Convert the given CFG to CNF. Consider the given grammar G1:

1. S → a | aA | B
2. A → aBB | ε
3. B → Aa | b

Solution:

Step 1: We will create a new production S1 → S, as the start symbol S appears on the RHS. The grammar will be:

1. S1 → S
2. S → a | aA | B
3. A → aBB | ε
4. B → Aa | b

Step 2: As grammar G1 contains A → ε null production, its removal from the grammar yields:

1. S1 → S
2. S → a | aA | B
3. A → aBB
4. B → Aa | b | a

Now, as grammar G1 contains Unit production S → B, its removal yield:

1. S1 → S
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Also remove the unit production S1 → S, its removal from the grammar yields:

1. S0 → a | aA | Aa | b
2. S → a | aA | Aa | b
3. A → aBB
4. B → Aa | b | a

Step 3: In the production rule S0 → aA | Aa, S → aA | Aa, A → aBB and B → Aa, terminal a exists on RHS with non-terminals. So we will replace terminal a with X:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → XBB
4. B → AX | b | a
5. X → a

Step 4: In the production rule A → XBB, RHS has more than two symbols, removing it from grammar yield:

1. S0 → a | XA | AX | b
2. S → a | XA | AX | b
3. A → RB
4. B → AX | b | a
5. X → a
6. R → XB

Hence, for the given grammar, this is the required CNF.

Greibach Normal Form (GNF)

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF(Greibach normal form) if all the production rules satisfy one of the following conditions:

- A start symbol generating ε. For example, S → ε.
- A non-terminal generating a terminal. For example, A → a.
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

For example:

1. G1 = {S → aAB | aB, A → aA| a, B → bB | b}
2. G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF. However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.

Steps for converting CFG into GNF

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

If any production rule in the grammar is not in GNF form, convert it.

Example:

1. S → XB | AA
2. A → a | SA
3. B → b
4. X → a

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

The production rule A → SA is not in GNF, so we substitute S → XB | AA in the production rule A → SA as:

1. S → XB | AA
2. A → a | XBA | AAA
3. B → b
4. X → a

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S → XB and B → XBA as:

1. S → aB | AA

2. A → a | aBA | AAA
3. B → b
4. X → a

Now we will remove left recursion (A → AAA), we get:

1. S → aB | AA
2. A → aC | aBAC
3. C → AAC | ε
4. B → b
5. X → a

Now we will remove null production C → ε, we get:

1. S → aB | AA
2. A → aC | aBAC | a | aBA
3. C → AAC | AA
4. B → b
5. X → a

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:

1. S → aB | aCA | aBACA | aA | aBAA
2. A → aC | aBAC | a | aBA
3. C → AAC
4. C → aCA | aBACA | aA | aBAA
5. B → b
6. X → a

The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:

1. S → aB | aCA | aBACA | aA | aBAA
2. A → aC | aBAC | a | aBA
3. C → aCAC | aBACAC | aAC | aBAAC
4. C → aCA | aBACA | aA | aBAA

5. B → b
6. X → a

Hence, this is the GNF form for the grammar G.

Pumping Lemma for CFG

Lemma

If **L** is a context-free language, there is a pumping length **p** such that any string **w ∈ L** of length **≥ p** can be written as **w = uvxyz**, where **vy ≠ ε**, **|vxy| ≤ p**, and for all **i ≥ 0, uv^ixy^iz ∈ L**.

Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

Problem

Find out whether the language **L = {$x^ny^nz^n$ | n ≥ 1}** is context free or not.

Solution

Let **L** is context free. Then, **L** must satisfy pumping lemma.

At first, choose a number **n** of the pumping lemma. Then, take z as $0^n1^n2^n$.

Break **z** into **uvwxy,** where

|vwx| ≤ n and vx ≠ ε.

Hence **vwx** cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

Case 1 − **vwx** has no 2s. Then **vx** has only 0s and 1s. Then **uwy**, which would have to be in **L**, has **n** 2s, but fewer than **n** 0s or 1s.

Case 2 − **vwx** has no 0s.

Here contradiction occurs.

Hence, **L** is not a context-free language.

CFL Closure Property

Context-free languages are **closed** under −

- Union
- Concatenation

- Kleene Star operation

Union

Let L_1 and L_2 be two context free languages. Then $L_1 \cup L_2$ is also context free.

Example

Let $L_1 = \{ a^n b^n , n > 0\}$. Corresponding grammar G_1 will have P: S1 → aAb|ab

Let $L_2 = \{ c^m d^m , m \geq 0\}$. Corresponding grammar G_2 will have P: S2 → cBb| ε

Union of L_1 and L_2, $L = L_1 \cup L_2 = \{ a^n b^n \} \cup \{ c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 | S2

Concatenation

If L_1 and L_2 are context free languages, then L_1L_2 is also context free.

Example

Union of the languages L_1 and L_2, $L = L_1L_2 = \{ a^n b^n c^m d^m \}$

The corresponding grammar G will have the additional production S → S1 S2

Kleene Star

If L is a context free language, then L* is also context free.

Example

Let $L = \{ a^n b^n , n \geq 0\}$. Corresponding grammar G will have P: S → aAb| ε

Kleene Star $L_1 = \{ a^n b^n \}^*$

The corresponding grammar G_1 will have additional productions S1 → SS_1 | ε

Context-free languages are **not closed** under −

- **Intersection** − If L1 and L2 are context free languages, then L1 ∩ L2 is not necessarily context free.
- **Intersection with Regular Language** − If L1 is a regular language and L2 is a context free language, then L1 ∩ L2 is a context free language.
- **Complement** − If L1 is a context free language, then L1' may not be context free.

Pushdown Automata(PDA)

- Pushdown automata is a way to implement a CFG in the same way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

- Pushdown automata is simply an NFA augmented with an "external stack memory". The addition of stack is used to provide a last-in-first-out memory management capability to Pushdown automata. Pushdown automata can store an unbounded amount of information on the stack. It can access a limited amount of information on the stack. A PDA can push an element onto the top of the stack and pop off an element from the top of the stack. To read an element into the stack, the top elements must be popped off and are lost.
- A PDA is more powerful than FA. Any language which can be acceptable by FA can also be acceptable by PDA. PDA also accepts a class of language which even cannot be accepted by FA. Thus PDA is much more superior to FA.

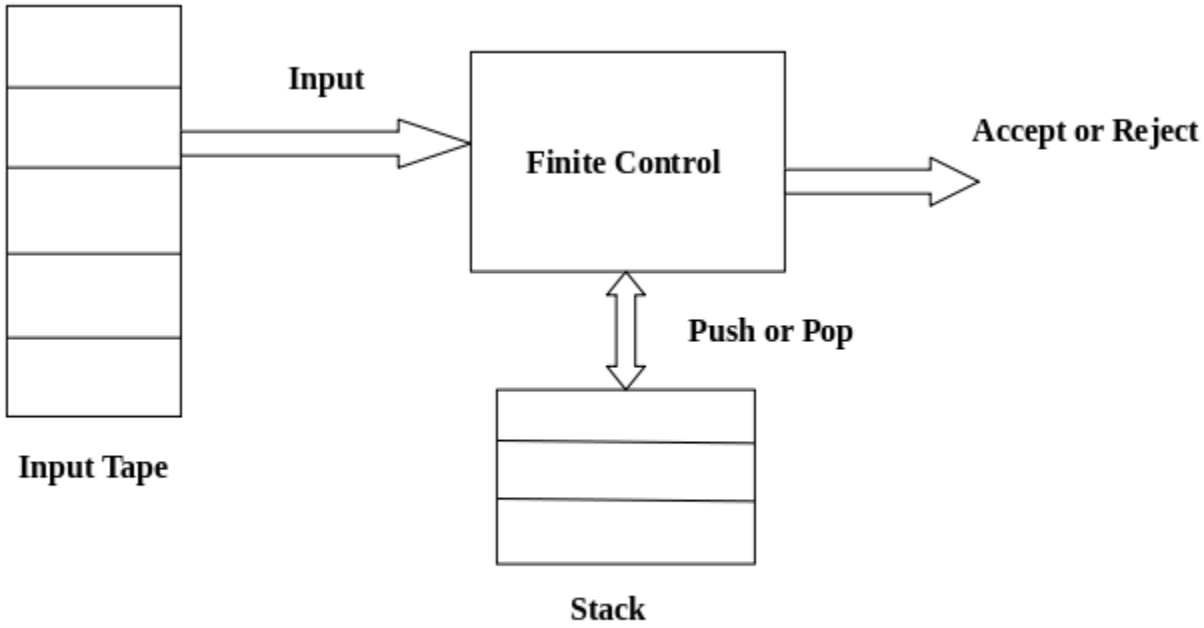


Fig: Pushdown Automata

PDA Components:

Input tape: The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time.

Finite control: The finite control has some pointer which points the current symbol which is to be read.

Stack: The stack is a structure in which we can push and remove the items from one end only. It has an infinite size. In PDA, the stack is used to store the items temporarily.

Formal definition of PDA:

The PDA can be defined as a collection of 7 components:

Q: the finite set of states

∑: the input set

Γ: a stack symbol which can be pushed and popped from the stack

q0: the initial state

Z: a start symbol which is in Γ.

F: a set of final states

δ: mapping function which is used for moving from current state to next state.

Instantaneous Description (ID)

ID is an informal notation of how a PDA computes an input string and make a decision that string is accepted or rejected.

An instantaneous description is a triple (q, w, α) where:

q describes the current state.

w describes the remaining input.

α describes the stack contents, top at the left.

Turnstile Notation:

⊢ sign describes the turnstile notation and represents one move.

⊢* sign describes a sequence of moves.

For example,

(p, b, T) ⊢ (q, w, α)

In the above example, while taking a transition from state p to q, the input symbol 'b' is consumed, and the top of the stack 'T' is represented by a new string α.

Example 1:

Design a PDA for accepting a language $\{a^n b^{2n} \mid n>=1\}$.

Solution: In this language, n number of a's should be followed by 2n number of b's. Hence, we will apply a very simple logic, and that is if we read single 'a', we will push two a's onto the stack. As soon as we read 'b' then for every single 'b' only one 'a' should get popped from the stack.

The ID can be constructed as follows:

1. δ(q0, a, Z) = (q0, aaZ)
2. δ(q0, a, a) = (q0, aaa)

Now when we read b, we will change the state from q0 to q1 and start popping corresponding 'a'. Hence,

1. δ(q0, b, a) = (q1, ε)

Thus this process of popping 'b' will be repeated unless all the symbols are read. Note that popping action occurs in state q1 only.

1. δ(q1, b, a) = (q1, ε)

After reading all b's, all the corresponding a's should get popped. Hence when we read ε as input symbol then there should be nothing in the stack. Hence the move will be:

1. δ(q1, ε, Z) = (q2, ε)

Where

PDA = ({q0, q1, q2}, {a, b}, {a, Z}, δ, q0, Z, {q2})

We can summarize the ID as:

1. δ(q0, a, Z) = (q0, aaZ)

2. δ(q0, a, a) = (q0, aaa)
3. δ(q0, b, a) = (q1, ε)
4. δ(q1, b, a) = (q1, ε)
5. δ(q1, ε, Z) = (q2, ε)

Now we will simulate this PDA for the input string "aaabbbbbb".

1. δ(q0, aaabbbbbb, Z) ⊢ δ(q0, aabbbbbb, aaZ)
2. ⊢ δ(q0, abbbbbb, aaaaZ)
3. ⊢ δ(q0, bbbbbb, aaaaaaZ)
4. ⊢ δ(q1, bbbbb, aaaaaZ)
5. ⊢ δ(q1, bbbb, aaaaZ)
6. ⊢ δ(q1, bbb, aaaZ)
7. ⊢ δ(q1, bb, aaZ)
8. ⊢ δ(q1, b, aZ)
9. ⊢ δ(q1, ε, Z)
10. ⊢ δ(q2, ε)
11. ACCEPT

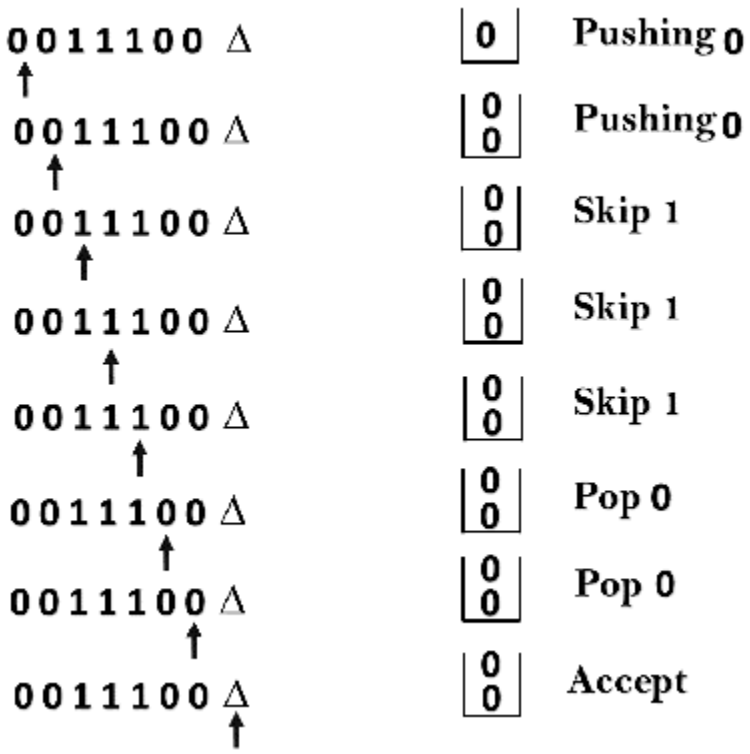
Example 2:

Design a PDA for accepting a language $\{0^n1^m0^n \mid m, n>=1\}$.

Solution: In this PDA, n number of 0's are followed by any number of 1's followed n number of 0's. Hence the logic for design of such PDA will be as follows:

Push all 0's onto the stack on encountering first 0's. Then if we read 1, just do nothing. Then read 0, and on each read of 0, pop one 0 from the stack.

For instance:



This scenario can be written in the ID form as:

1. δ(q0, 0, Z) = δ(q0, 0Z)
2. δ(q0, 0, 0) = δ(q0, 00)
3. δ(q0, 1, 0) = δ(q1, 0)
4. δ(q0, 1, 0) = δ(q1, 0)
5. δ(q1, 0, 0) = δ(q1, ε)
6. δ(q0, ε, Z) = δ(q2, Z) (ACCEPT state)

Now we will simulate this PDA for the input string "0011100".

1. δ(q0, 0011100, Z) ⊢ δ(q0, 011100, 0Z)
2. ⊢ δ(q0, 11100, 00Z)
3. ⊢ δ(q0, 1100, 00Z)
4. ⊢ δ(q1, 100, 00Z)
5. ⊢ δ(q1, 00, 00Z)
6. ⊢ δ(q1, 0, 0Z)
7. ⊢ δ(q1, ε, Z)
8. ⊢ δ(q2, Z)

9. ACCEPT

PDA Acceptance

A language can be accepted by Pushdown automata using two approaches:

1. Acceptance by Final State: The PDA is said to accept its input by the final state if it enters any final state in zero or more moves after reading the entire input.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by the final state can be defined as:

1. L(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ F}

2. Acceptance by Empty Stack: On reading the input string from the initial configuration for some PDA, the stack of PDA gets empty.

Let P =(Q, ∑, Γ, δ, q0, Z, F) be a PDA. The language acceptable by empty stack can be defined as:

1. N(PDA) = {w | (q0, w, Z) ⊢* (p, ε, ε), q ∈ Q}

Equivalence of Acceptance by Final State and Empty Stack

- If L = N(P1) for some PDA P1, then there is a PDA P2 such that L = L(P2). That means the language accepted by empty stack PDA will also be accepted by final state PDA.
- If there is a language L = L (P1) for some PDA P1 then there is a PDA P2 such that L = N(P2). That means language accepted by final state PDA is also acceptable by empty stack PDA.

Example:

Construct a PDA that accepts the language L over {0, 1} by empty stack which accepts all the string of 0's and 1's in which a number of 0's are twice of number of 1's.

Solution:

There are two parts for designing this PDA:

- If 1 comes before any 0's

- If 0 comes before any 1's.

We are going to design the first part i.e. 1 comes before 0's. The logic is that read single 1 and push two 1's onto the stack. Thereafter on reading two 0's, POP two 1's from the stack. The δ can be

1. δ(q0, 1, Z) = (q0, 11, Z) Here Z represents that stack is empty
2. δ(q0, 0, 1) = (q0, ε)

Now, consider the second part i.e. if 0 comes before 1's. The logic is that read first 0, push it onto the stack and change state from q0 to q1. [Note that state q1 indicates that first 0 is read and still second 0 has yet to read].

Being in q1, if 1 is encountered then POP 0. Being in q1, if 0 is read then simply read that second 0 and move ahead. The δ will be:

1. δ(q0, 0, Z) = (q1, 0Z)
2. δ(q1, 0, 0) = (q1, 0)
3. δ(q1, 0, Z) = (q0, ε) (indicate that one 0 and one 1 is already read, so simply read the second 0)
4. δ(q1, 1, 0) = (q1, ε)

Now, summarize the complete PDA for given L is:

1. δ(q0, 1, Z) = (q0, 11Z)
2. δ(q0, 0, 1) = (q1, ε)
3. δ(q0, 0, Z) = (q1, 0Z)
4. δ(q1, 0, 0) = (q1, 0)
5. δ(q1, 0, Z) = (q0, ε)
6. δ(q0, ε, Z) = (q0, ε) ACCEPT state

Non-deterministic Pushdown Automata

The non-deterministic pushdown automata is very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus NPDA is more powerful than DPDA.

Example:

Design PDA for Palindrome strips.

Solution:

Suppose the language consists of string L = {aba, aa, bb, bab, bbabb, aabaa, ...]. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

1. $\delta(q1, a, Z) = (q1, aZ)$
2. $\delta(q0, b, Z) = (q1, bZ)$
3. $\delta(q0, a, a) = (q1, aa)$
4. $\delta(q1, a, b) = (q1, ab)$
5. $\delta(q1, a, b) = (q1, ba)$
6. $\delta(q1, b, b) = (q1, bb)$

Pushing the symbols onto the stack

7. $\delta(q1, a, a) = (q2, \varepsilon)$
8. $\delta(q1, b, b) = (q2, \varepsilon)$
9. $\delta(q2, a, a) = (q2, \varepsilon)$
10. $\delta(q2, b, b) = (q2, \varepsilon)$
11. $\delta(q2, \varepsilon, Z) = (q2, \varepsilon)$

Popping the symbols on reading the same kind of symbol

Simulation of abaaba

1. δ(q1, abaaba, Z) Apply rule 1
2. ⊢ δ(q1, baaba, aZ) Apply rule 5
3. ⊢ δ(q1, aaba, baZ) Apply rule 4
4. ⊢ δ(q1, aba, abaZ) Apply rule 7
5. ⊢ δ(q2, ba, baZ) Apply rule 8
6. ⊢ δ(q2, a, aZ) Apply rule 7
7. ⊢ δ(q2, ε, Z) Apply rule 11
8. ⊢ δ(q2, ε) Accept

CFG to PDA Conversion

The first symbol on R.H.S. production must be a terminal symbol. The following steps are used to obtain PDA from CFG is:

Step 1: Convert the given productions of CFG into GNF.

Step 2: The PDA will only have one state {q}.

Step 3: The initial symbol of CFG will be the initial symbol in the PDA.

Step 4: For non-terminal symbol, add the following rule:

1. δ(q, ε, A) = (q, α)

Where the production rule is A → α

Step 5: For each terminal symbols, add the following rule:

1. δ(q, a, a) = (q, ε) **for** every terminal symbol

Example 1:

Convert the following grammar to a PDA that accepts the same language.

1. S → 0S1 | A
2. A → 1A0 | S | ε

Solution:

The CFG can be first simplified by eliminating unit productions:

1. S → 0S1 | 1S0 | ε

Now we will convert this CFG to GNF:

1. S → 0SX | 1SY | ε
2. X → 1
3. Y → 0

The PDA can be:

R1: δ(q, ε, S) = {(q, 0SX) | (q, 1SY) | (q, ε)}
R2: δ(q, ε, X) = {(q, 1)}
R3: δ(q, ε, Y) = {(q, 0)}
R4: δ(q, 0, 0) = {(q, ε)}
R5: δ(q, 1, 1) = {(q, ε)}

Example 2:

Construct PDA for the given CFG, and test whether 0104 is acceptable by this PDA.

1. S → 0BB
2. B → 0S | 1S | 0

Solution:

The PDA can be given as:

1. A = {(q), (0, 1), (S, B, 0, 1), δ, q, S, ?}

The production rule δ can be:

R1: δ(q, ε, S) = {(q, 0BB)}
R2: δ(q, ε, B) = {(q, 0S) | (q, 1S) | (q, 0)}
R3: δ(q, 0, 0) = {(q, ε)}
R4: δ(q, 1, 1) = {(q, ε)}

Testing 010^4 i.e. 010000 against PDA:

1. δ(q, 010000, S) ⊢ δ(q, 010000, 0BB)
2. ⊢ δ(q, 10000, BB) R1
3. ⊢ δ(q, 10000,1SB) R3
4. ⊢ δ(q, 0000, SB) R2
5. ⊢ δ(q, 0000, 0BBB) R1
6. ⊢ δ(q, 000, BBB) R3
7. ⊢ δ(q, 000, 0BB) R2
8. ⊢ δ(q, 00, BB) R3
9. ⊢ δ(q, 00, 0B) R2
10. ⊢ δ(q, 0, B) R3
11. ⊢ δ(q, 0, 0) R2
12. ⊢ δ(q, ε) R3
13. ACCEPT

Thus 010^4 is accepted by the PDA.

Example 3:

Draw a PDA for the CFG given below:

1. S → aSb
2. S → a | b | ε

Solution:

The PDA can be given as:

1. P = {(q), (a, b), (S, a, b, z0), δ, q, z0, q}

The mapping function δ will be:

R1: `δ(q, ε, S) = {(q, aSb)}`

R2: `δ(q, ε, S) = {(q, a) | (q, b) | (q, ε)}`

R3: `δ(q, a, a) = {(q, ε)}`

R4: `δ(q, b, b) = {(q, ε)}`

R5: `δ(q, ε, z0) = {(q, ε)}`

Simulation: Consider the string aaabb

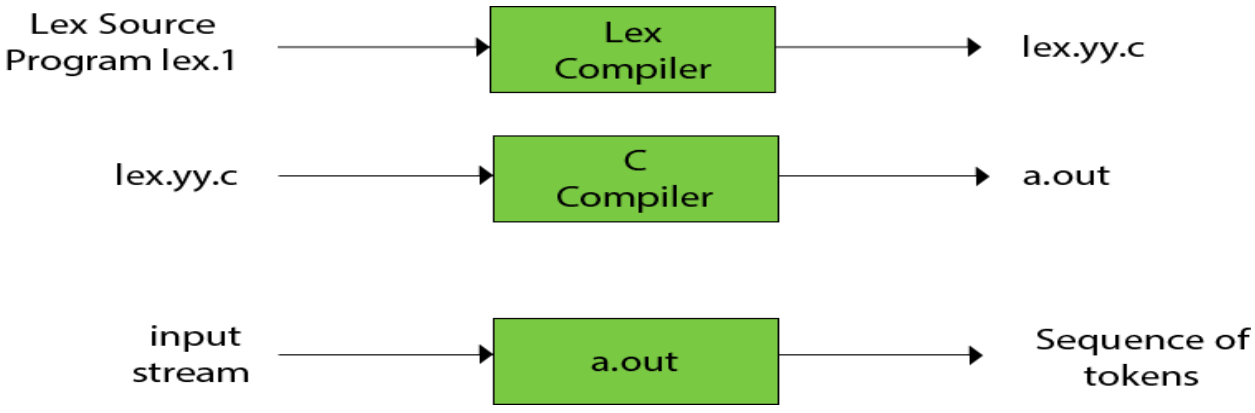
1. δ(q, εaaabb, S) ⊢ δ(q, aaabb, aSb) R3
2. ⊢ δ(q, εaabb, Sb) R1
3. ⊢ δ(q, aabb, aSbb) R3
4. ⊢ δ(q, εabb, Sbb) R2
5. ⊢ δ(q, abb, abb) R3
6. ⊢ δ(q, bb, bb) R4
7. ⊢ δ(q, b, b) R4
8. ⊢ δ(q, ε, z0) R5
9. ⊢ δ(q, ε)
10. ACCEPT

LEX

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines }

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

YACC

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

These are some points about YACC:

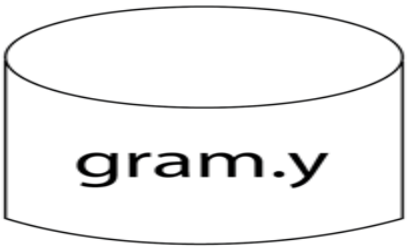
Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

- The output file "file.output" contains the parsing tables.

- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

The basic operational sequence is as follows:

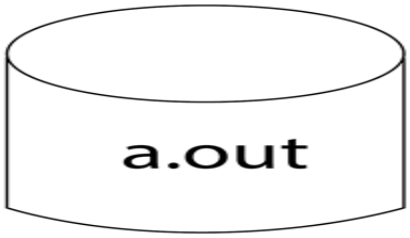


This file contains the desired grammar in YACC format.

It shows the YACC program.

It is the c source program created by YACC.

C Compiler



Executable file that will parse grammar given in gram.Y