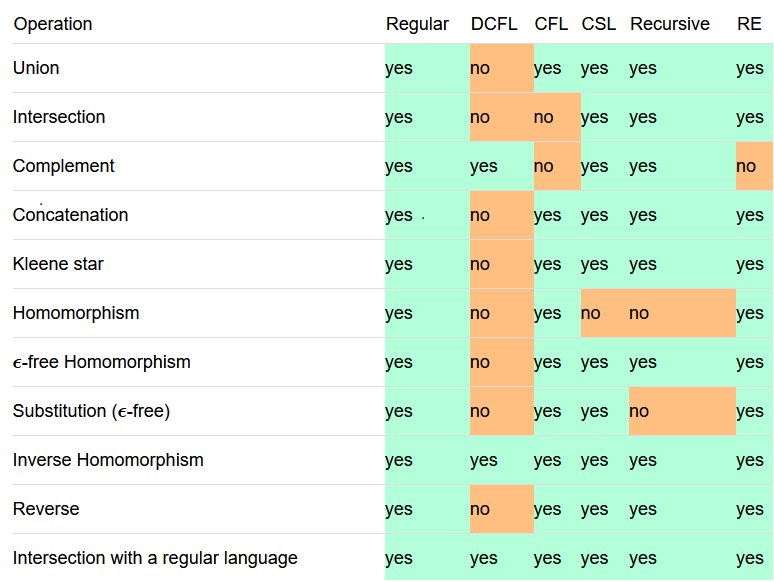
**TYPES OF GRAMMAR :-**









|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ***REGULAR LANGUAGE*** | ***CONTEXT FREE LANGUAGE*** | ***CONTEXT SENSITIVE LANG*** | ***RECURSIVE LANGUAGE*** | ***RECURSIVELY ENUMERABLE LANGUAGE*** |
| **UNION** | TRUE | TRUE | TRUE | TRUE | TRUE |
| **INTERSECTION** | TRUE | FALSE |  | TRUE | TRUE |
| **COMPLEMENT** | TRUE | FALSE |  | TRUE | FALSE |
| **CONCATENATION** | TRUE | TRUE |  | TRUE | TRUE |
| **KLEENE CLOSURE** | TRUE | TRUE |  | TRUE | TRUE |
| **SET DIFFERENCE** |  |  |  | TRUE | FALSE |
| **REVERSAL** |  |  | TRUE | TRUE | TRUE |
| **HOMOMORPHISM** | TRUE |  |  |  |  |
| **INVERSE HOMOMORPHISM** | TRUE |  |  |  |  |
| **QUOTIENT** | TRUE |  |  |  |  |
| **SUBSTITUITION** | TRUE |  |  |  |  |

Every regular language is a context-free language, but a context-free language may not be regular.

Every context-free language is a recursive language, but a recursive language may not be context-free.

Every recursive language is a recursively enumerable language, but a recursively enumerable language may not be recursive.

**Type 3 - Regular Languages**

Linear grammar is a context-free grammar that has at most one non terminal in the right hand side of each of its productions.

the left-linear or left regular grammars, in which all non terminals in right hand sides are at the left ends.

the right-linear or right regular grammars, in which all non terminals in right hand sides are at the right ends.

Right-linear (right-regular), with rules of the form

A --> aB or A --> a ,

where A and B are single non-terminal symbols, a is a terminal symbol

Parse trees with these grammars are right-branching.

Left-linear (left-regular), with rules of the form

A --> B a or A --> a

Parse trees with these grammars are left-branching.

Examples of regular languages are pattern matching languages (regular expressions).

**Type 2 - Context-Free Languages**

A Context-Free Grammar (CFG) is one whose production rules are of the form:

A --> a

where A is any single non-terminal, and a is any combination of terminals and non-terminals.

The minimal automaton that recognizes context-free languages is a push-down automaton. It uses stack when expanding the non-terminal symbols with the right-hand side of the corresponding grammar rule.

Examples of CFLs are some simple programming languages.

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.

**Type 1 - Context-Sensitive Languages**

Context-Sensitive grammars may have more than one symbol on the left-hand-side of their grammar rules,

provided that at least one of them is a non-terminal and the number of symbols on the left-hand-side

does not exceed the number of symbols on the right-hand-side.

Their rules have the form:

aAv --> aßv

where A is a single non-terminal symbol, and a ß --> are any combination of terminals and non-terminals.

The automaton which recognizes a context-sensitive language is called a linear-bounded automaton:

Examples of context-sensitive languages are most programming languages

**Type 0 - Unrestricted (Free) Languages**

Unrestricted grammars have no restrictions on their grammar rules, except that there must be at least one non-terminal on the left-hand-side. The rules have the form

a --> ß

where a and ß are arbitrary strings of terminal and non-terminal symbols and a ¹ e (the empty string)

The type of automata which can recognize such a language is a Turing machine, with an infinitely long memory.

Examples of unrestricted languages are almost all natural languages.

**NFA DFA STATES :-**

-If NFA has n states, then its DFA can have 2N states in worst case. That is at maximum 2N cases.

-Worst case complexity is O(2N).

-------------------------------------------------------------------------

-Let Total no.of states = n, Total no.of symbols = m then

//Total number of DFS is ( 2n) \* (nnm)(this may be wrong)

//Total number of NFA is (2n) \* (2n)^(nm)(this may be wrong)

**MEALY MACHINE**

A Mealy machine has outputs that depend on the state and input (thus, the FSM has the output written on edges)

**MOORE MACHINE**

A Moore machine has outputs that depend on state only (thus, the FSM has the output written in the state itself)

**PUMPING LEMMA**:-

-If the language is finite, it is regular, otherwise it might be non-regular.

-It is to prove that a language is not regular. It is important to note that pumping lemma is not used for proving whether a language is regular. It is rather used for proving if the language is not regular.

**LEMMA:-**

1)Select a string z in the language L.

2)Break the string z into x, y and z in accordance with the above conditions imposed by the pumping lemma.

3)Now check if there is any contradiction to the pumping lemma for any value of i.

**More Explanation** :-

Suppose that a language L is regular. Then there is a FA that accepts L.

Let n be the number of states of that FA. Then for any string x in L with |x| ≥ n, there are strings u, v and w which satisfy the following:

x = uvw

|uv| ≤ n

|v| > 0 is same as v ≠ ε

For every integer m ≥ 0, uvmw ∈ L.

If L is regular then for every x such that |x| ≥ n then there exists uvw such that x=uvw, v ≠ ε, |uv| ≤ n, and for which uviw is in L for every i.

(https://www.youtube.com/watch?v=2qrlwpjs4J8)

**Example:-**

1.Language L = {anbn for n>=0} is not regular.

2. The language containing strings of balanced parentheses is not regular.

3. Show that L2 = {0m1m | x ∈ {0, 1}\*} is not regular.

**4**. The language of palindromes over {0, 1} is not regular.

5. L2 = {xx | x ∈ {0, 1}\*} is not regular.

6. Prove that Language L = {0n: n is a perfect square} is irregular.

7. We prove that L3 = {1n2 | n ≥ 0} is not regular.

7. We prove that L = {02n | n ≥ 0} is regular. Because

L is described by the regular expression (00)∗ so it is regular.

8. Prove that L = {an: n is a prime number} is not regular.

9. L1 = {(ab)n ak | n > k, k ≥ 0} is not regular

10. L2 = {an bm | n ¹ m} is not regular

11. Extracting letters in the even-numbered positions is regular

12. Removing the first two leftmost symbol is regular

13. Intersection of Context free language with Regular language is Context free.

**Pumping Lemma for Context Free :-**

**Example**

1. L  = { *anbncn* | *n* > 0 } can be shown to be non-context-free by using the pumping lemma
2. L  = { *anbn* | *n* > 0 } is an deterministic context-free
3. L  = { wcw*R* | w∈ {0, 1}\*} is an deterministic context-free
4. L = {anbncm | n, m > 0} is a context free
5. L = {anbmcm | n, m > 0} is a context free

**Chomsky Normal Form:-**

A CF Grammar is in Chomsky Normal Form iff all productions are in the form

A → BC or A → a where A,B,C are in V and a is in T.

For Example :-

Valid grammars are below

S ->AS|a

A->SA |b

Invalid grammars are below

S->AS|AAS

A->SA|aa

**Greibach normal form** :-

A CF Grammar is in Greibach Normal Form if all productions are in the form A → ax where a ε T and x ε V\*

A context-free grammar is in Greibach normal form, if all production rules are of the form:

A -> a A1 A2 A3 ----An

or

A -> epsilon

**Ambiguous Grammar:-**

A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one LeftMost Derivation Tree (LMDT) or RightMost Derivation Tree (RMDT).

**PUSH DOWN AUTOMATA**

A pushdown automaton is a seven-tuple

M = (Q, Σ, Γ, δ, q0, Z0, F), where

• Q — finite set of states

• Σ — finite input alphabet

• Γ — finite alphabet of pushdown symbols

• δ — mapping functions / transition function

• q0 ∈ Q — starting/initial state

• Z0 ∈Γ— start symbol on the pushdown

• F ⊆ Q — set of final states

Example:

δ (q, a, a ) = ( p, aa) Meaning:

 When in state q,

 Reading in a from the tape

 With an a popped off the stack

 The machine will

 Go into state p

 Push the string “aa” onto the stack

**Recursive and Recursively Enumerable Set** :-

1. The recursive and recursively enumerable languages are closed under union, concatenation, closure, Kleene closure and intersection.
2. The recursive languages are also closed under set difference and complementation.
3. The recursively enumerable languages are not closed under complementation or set difference.
4. if a language and its complement are both r.e., then the language is recursive.
5. Complement of a regular language is also regular
6. regular languages are closed under union, intersection, complementation, Kleene closure and Homo morphism
7. The context-free languages are not closed under intersection and complement
8. The context-free languages are closed under union, concatenation and Kleene closure.
9. Every context sensitive language is recursive(For more details see the set diagram in the first page)

**Rules for operator grammar :-**

operator grammer does not contain

1) No nullable variable (ie, P → ε)

2) Two adjacent non-terminal on RHS of production (ie. A -> BC )

**Unit Productions:-**

A unit production is one of the form A -> B, where both A and B are single non-terminals.

**NOTES:-**

* The concatenation of the empty language with any other language always gives the empty language itself.

**Expressive power :-**

Expressing power of any machine can be defined as the maximum number of languages it can accept. if machine M1 can accept more languages then M2 then we can say that expressing power of M1 is greater than M2.

a)languages accepted by NFA, will also be accepted by DFA because we can make DFA corresponding to NFA.so their expressing power is same.

b)In this case languages accepted by NPDA is more than DPDA ,so expressing power of NPDA is more than DPDA. So expressive power is different.

c)both deterministic and non deterministic turing can accept same language.so there expressing power is same.

d) In turing machine if we increase the number of tape then also language accepted by that machine is same as single tape turing machine.so there expressing power is same.

**HALTING PROBLEM**

A problem is said to be a Decidable problem if there exists a corresponding Turing machine which halts on every input with an answer- yes or no. It is also important to know that these problems are termed as Turing Decidable since a Turing machine always halts on every input, accepting or rejecting it.

**List of Decidable problem:-**

1. Are two regular languages L and M equivalent?

We can easily check this by using Set Difference operation.

L-M =Null and M-L =Null.

Hence (L-M) U (M-L) = Null, then L,M are equivalent.

2. Membership of a CFL?

We can always find whether a string exists in a given CFL by using an algorithm based on dynamic programming.

3. Emptiness of a CFL

By checking the production rules of the CFL we can easily state whether the language generates any strings or not.

List of UnDecidable problem:-

---------------------------

1. Whether a CFG generates all the strings or not?

As a CFG generates infinite strings, we can’t ever reach up to the last string and hence it is Undecidable.

2. Whether two CFG L and M equal?

Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.

3. Ambiguity of CFG?

There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.

4. Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?

It is also an Undecidable Problem as there doesn’t exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.

5. Is a language Learning which is a CFL, regular?

This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

6. Membership problem of a Turing Machine?

7. Finiteness of a Turing Machine?

8. Emptiness of a Turing Machine?

9. Whether the language accepted by Turing Machine is regular or CFL?

**COMPILER DESIGN**

Data structures / Databases required:



***Source program*** – Original Source program, which is scanned by compiler as string of characters.

***Terminal Table*** – A permanent database that has entry for each terminal symbols such as arithmetic operators, keywords, punctuation characters such as ‘;’ , ‘,’etc Fields: Name of the symbol.

***Literal Table –*** This table is created during lexical analysis so as to describe all literals in the program.

Fields: Name of the literal.

***Identifier Table*** – Created during lexical analysis and describes all identifiers in the program.

Fields: Name of the identifier.

***Uniform Symbol Table*** – Created during lexical analysis to represent the program as a string of tokens, rather than of individual characters. Each uniform symbol contains the identification of the table to which it belongs.( IDN – Identifier table, LIT – Literal Table TRM – Terminal Symbol Table)and index within that table.

***Buffer –*** One buffer or two buffer schemes to load source program part by part to reduce disk i/o.

**Removal of Left Recursion :-**

The production A => Aα | β

is converted into following productions

A => βA'

A'=> αA' | ε

**Removal of Left Factoring:**

Left factoring is another useful grammar transformation used in parsing

For example, going from:

A -> α β | α γ

to:

A -> α A'

A' -> β | γ

Left factor example:

Let the given grammar: A-->ab1 | ab2 | ab3

A --> aA', A' --> b1 | b2| b3

Let the given grammar: S -> abS | aSb

S --> aS', S' -> bS | Sb

**Shift-reduce Parsing :-**

- Bottom-up Parsing

- A general form of shift-reduce parsing is LR (scanning from Left to right and using Right-most derivation in reverse)

- It suffers from both shift reduce conflict and reduce reduce conflict

**Shift-Reduce And Reduce-Reduce Conflict:-**

If it is legal to shift or reduce, there is a shift-reduce conflict

If it is legal to reduce by two different productions, there is a reduce-reduce conflict

**Source of Conflicts:-**

-Ambiguous grammars always cause conflicts

-But beware, so do many nonambiguous grammars

**LL parse**:-

1) directly corresponds with the Polish Notation

2) LL parsers produce a leftmost derivation

3) LL parsers build the tree from the top-down

4) LL parsers are often called “predictive parsers,”

**LR Parse:-**

1) LR corresponds to Reverse Polish Notation

2) LR parsers produce a reversed rightmost derivation

3) LR parsers build the tree from the bottom-up

4) LR parsers are often called “shift-reduce parsers

5) LR parsers are also known as LR(k) parsers,

6) L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

**LL(1) grammar :-**

In trying to turn a grammar into an LL(1) grammar you should try the following steps:

1-Remove left-recursion.

2-Expose first set clashes.

3-Left-factor the grammar.

4-Attempt to remove first/follow set clashes.

5-Return to step 2 after thinking about whether you want to continue!

6 – Left most derivation

**LL(1) GRAMMAR**

First L stands for scanning input from Left to Right.

Second L stands for Left Most Derivation.

1 stands for using one input symbol at each step.

**How to Check Grammar is LL(1) or not**

LL(1) grammar is grammar which is not

1.Ambiguous. 2. Left Recursive. 3. Left factored.

If a grammar have any of the above problem simply it can’t be LL(1).

**Rules for First Sets**

1. If X is a terminal then First(X) is just X!
2. If there is a Production X → ε then add ε to first(X)
3. If there is a Production X → Y1Y2..Yk then add first(Y1Y2..Yk) to first(X)
4. First(Y1Y2..Yk) is either
5. First(Y1) (if First(Y1) doesn't contain ε)
6. OR (if First(Y1) does contain ε) then First (Y1Y2..Yk) is everything in First(Y1) <except for ε > as well as everything in First(Y2..Yk)
7. If First(Y1) First(Y2)..First(Yk) all contain ε then add ε to First(Y1Y2..Yk) as well.

**Rules for Follow Sets**

1. First put $ (the end of input marker) in Follow(S) (S is the start symbol)
2. If there is a production A → aBb, (where a can be a whole string) then everything in FIRST(b) except for ε is placed in FOLLOW(B).
3. If there is a production A → aB, then everything in FOLLOW(A) is in FOLLOW(B)
4. If there is a production A → aBb, where FIRST(b) contains ε, then everything in FOLLOW(A) is in FOLLOW(B)

(http://www.comrevo.com/2015/08/how-to-find-first-and-follow-of-grammar-with-examples.html)

**Synthesized Attribute:-** An attribute that gets its values from the attributes attached to the children of its non-terminal.

**Inherited Attribute:-** An attribute that gets its values from the attributes attached to the parent (or siblings) of its non-terminal.

**AN ANNOTATED PARSE TREE:-** It is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called annotating or decorating the parse tree.

For Example shows the below picture.



**CODE OPTIMIZATION**



**Code motion(Loop Invariant code)** :-

If variables used in a computation within a loop are not altered within the loop, the calculation can be performed outside of the loop and the results used within the loop.

**Common subexpression elimination** :-

In common expressions, the same value is recalculated in a subsequent expression. The duplicate expression can be eliminated by using the previous value.

**Constant propagation:-**

Constants used in an expression are combined, and new ones are generated. Some implicit conversions between integers and floating-point types are done.

**Dead code elimination:-**

Eliminates code that cannot be reached or where the results are not subsequently used.

**Strength Reduction:-**

Replaces less efficient instructions with more efficient ones. For example, in array subscripting, an add instruction replaces a multiply instruction.

**Constant Folding:-**

If an expression such as 2+3\*4 is calculated and assigned 24.If a < b goto L1 else goto L2. If we know the a and b value, then we directly use the statement goto L1 or goto L2.

**LOOP JAMMING:-**

Loop jamming is a technique that merges the bodies of two loops if the two loops have the same number of iterations and they use the same indices. This eliminates the test of one loop. For example, consider the following loop:

{

for ( I = 0; I < 10; I ++)

for ( J = 0; J < 10; J ++) X [ I,J ] = 0;

for ( I = 0; I < 10; I ++) X [ I,I ] = 1;

}

Here, the bodies of the loops on I can be concatenated . The result of loop jamming will be:

{

for ( I = 0; I < 10; I ++){

for ( J = 0; J < 10; J ++) X [ I,J ] = 0; X [ I,I ] = 1; }}

**Loop Unrolling**

Loop overhead can be reduced by reducing the number of iterations and replicating the body of the loop.

**Example:**

In the code fragment below, the body of the loop can be replicated once and the number of iterations can be reduced from 100 to 50.

for (i = 0; i < 100; i++)

g ();

Below is the code fragment after loop unrolling.

for (i = 0; i < 100; i += 2)

{

g ();

g ();

}

**PEEPHOLE OPTIMIZATION:-**

Characteristics of peephole optimizations:

a) Redundant-instructions elimination

b) Flow-of-control optimizations

c) Algebraic simplifications

d) Use of machine idioms

e) Unreachable

**STATIC SINGLE ASSIGNMENT FORM:-**

It is a property of an intermediate representation (IR), which requires that each variable is assigned exactly once, and every variable is defined before it is used. Existing variables in the original IR are split into versions, new variables typically indicated by the original name with a subscript in textbooks, so that every definition gets its own version.

In Ordinary Form

y := 1

y := 2

x := y

In SSA form, both of these are immediate:

y1 := 1

y2 := 2

x1 := y2