UNIT I

Finite Automata and Regular Expressions

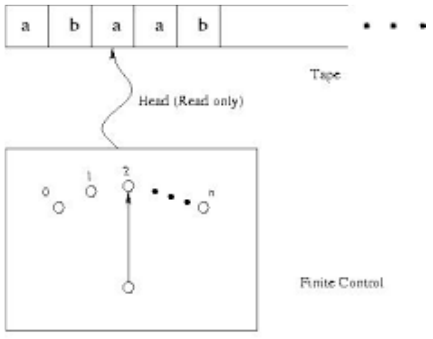
❖ Finite Automata:

A Finite Automata is the mathematical model of a digital computer. Finite Automata areused as string or language acceptors. They are mainly used in pattern matching tools likeLEX and Text editors.

- The Finite State System represents a mathematical model of a system with certain input.
- The model finally gives a certain output. The output given to the machine is processed by various states. These states are called intermediate states.
- A good example of finite state systems is the control mechanism of an elevator. Thismechanism only remembers the current floor number pressed, it does not remember all the previously pressed numbers.
- The finite state systems are useful in design of text editors, lexical analyzers and naturallanguage processing. The word "automaton" is singular and "automata" is plural.
- An automaton in which the output depends only on the input is called an automaton withoutmemory.
- An automaton in which the output depends on the input and state is called as automationwith memory.

Finite Automation Model:

- Informally, a FA – Finite Automata is a simple machine that reads an input string – one symbol at a time -- and then, after the input has been completely read, decides whether to accept or reject the input. As the symbols are read from the tape, the automaton can changeits state, to reflect how it reacts to what it has seen so far.
- The Finite Automata can be represented as,



Finite Automaton

i) Input Tape: Input tape is a linear tape having some cells which can hold an input symbol from $\sum$.

ii) Finite Control: It indicates the current state and decides the next state on receiving a particular input from the input tape. The tape reader reads the cells one by one from left to right and at any instance only one input symbol is read. The reading head examines read symbol and the head moves to the right side with or without changing the state. When the entire string is read and if finite control is in final state then the string is accepted otherwise rejected. The finite automaton can be represented by a transition diagram in which the vertices represent the states and the edges represent transitions.

- A Finite Automaton (FA) consists of a finite set of states and set of transitions among statesin response to inputs.
  - Always associated with a FA is a transition diagram, which is nothing but a'directed graph'.
  - The vertices of the graph correspond to the states of the FA.
  - The FA accepts a string of symbols from∑, x if the sequence of transitions corresponding to symbols in x leads from the state to an accepting state.
- Finite Automata can be classified into two type:
  1. FA without output or Language Recognizers (e.g. DFA and NFA)
  2. FA with output or Transducers (e.g. Moore and Mealy machines)

Finite Automata Definition:

Finite Automaton (FA), a collection of states in which we make transitions based upon input symbols.

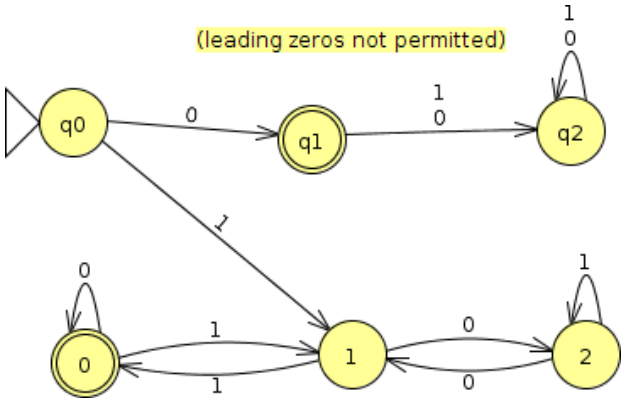
Definition: A Finite Automaton

A *finite automaton* (FA) is a 5-tuple (Q,Σ,q_0,A,δ) where

- Q is a finite set of *states*;
- Σ is a finite *input alphabet*;
- $q_0 \in Q$ is the *initial state*;
- $A \subseteq Q$ is the set of *accepting* states; and
- $\delta: Q \times \Sigma \rightarrow Q$is the transition function.

For any element q of Q and any symbol $\sigma \in \Sigma$, we interpret $\delta(q,\sigma)$ as the state to which the FA moves, if it is in state q and receives the input σ.

For example, for the FA shown here, we would say that:



- Q={q0,q1,q2,0,1,2}
  - These are simply labels for states, so we can use any symbol that is convenient.
- Σ={0,1}
  - These are the characters that we can supply as input.
- q0 is, well, q0 because we chose to use that matching label for the state.
- A={q1,0}
  - The accepting states
- δ={((q0,0),q1),((q0,1),1),((q1,0),q2),((q1,1),q2,(q2,0),q2),((q2,1),q2), ((0,0),0),((0,1),1), ((1,0),2),((1,1),0),((2,0),1),((2,1),2)}
  - Functions are, underneath it all, sets of pairs. The first element in each pair is the input to the function and the second element is the result. Hence when you see ((q0,0),q1) it means that, "if the input is (q0,0) the result is q1

    The input is itself a pair because δ was defined as a function of the form $Q\times\Sigma\rightarrow Q$so the input has the form $Q\times\Sigma$ the set of all pairs in which the first element is taken from set Q and the second element from set Σ.

  - Of course, there may be easier ways to visualize δ. In particular, we could do it via a table with the input state on one axis and the input character on another:

	Starting State					
Input	**q0**	**q1**	**q2**	**0**	**1**	**2**
0	q1	q2	q2	0	2	1
1	1	q2	q2	1	0	2

- The table representation is particularly useful because it suggests an efficient implementation. If we numbered our states instead of using arbitrary labels:

	Starting State					
Input	**0**	**1**	**2**	**4**	**5**	**6**
0	1	2	2	3	5	4
1	4	2	2	4	3	5

❖ Accepting the Union:

- L1={ab,aab,abab,abb,…….}
- L1={ab,aab,abab,abb,…….}

Here,

- L1= starts with a and end with b
- L2= starts with b and ends with a

Therefore,

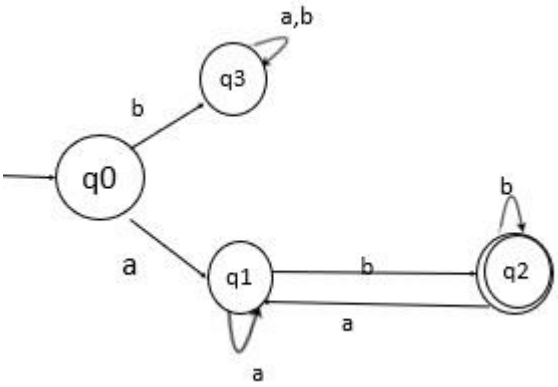
L=L1 U L2

Or

L=L1+L2

State transition diagram for L1

- The state transition diagram for the language L1 is given below −



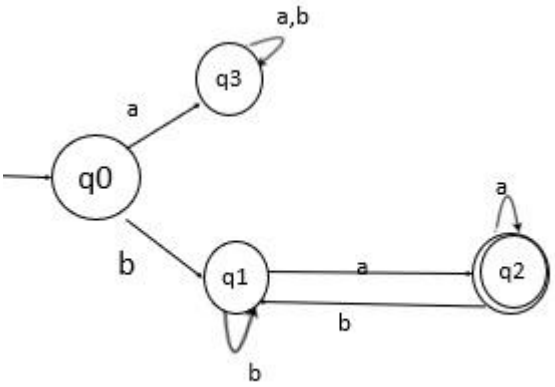
The above diagram accepts all strings starting with a and ending with b.Here,

- q0 is the initial state.
- q1 is an intermediate state.

- q2 is the final state.
- q3 is the dead state.

State transition diagram for L2

- The state transition diagram for language L2 is as follows −

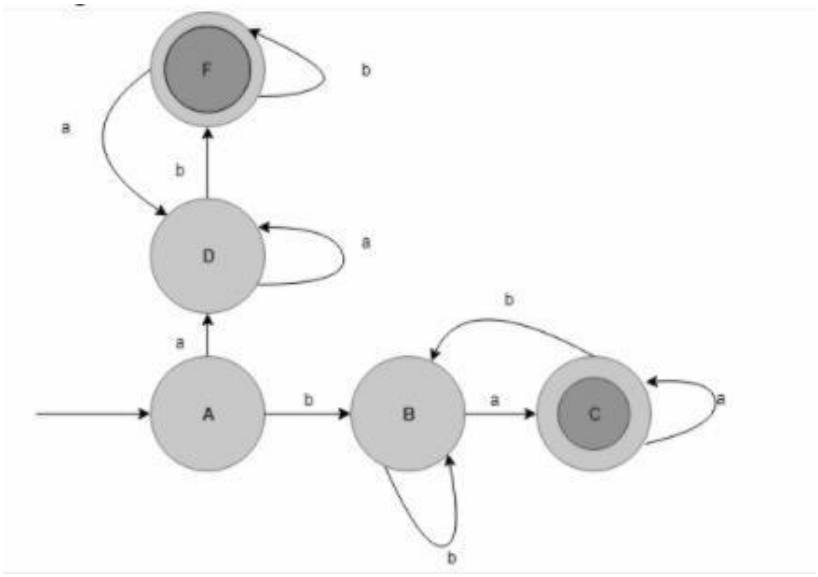


The above diagram accepts all strings starting with b and ending with a.Here,

- q0: Initial state.
- q1: Intermediate state.
- q2: Final state.
- q3: Dead state.

Now the union of L1 and L2 gives the final result of language which starts and ends with different elements.

The state transition diagram of L1 U L2 is as follows −



❖ Accepting Intersection:

Let's understand the intersection of two DFA with an example.

Designing a DFA for the set of string over {0, 1} such that it ends with 01 and has evennumber 0f 1's.

There two desired language will be formed:

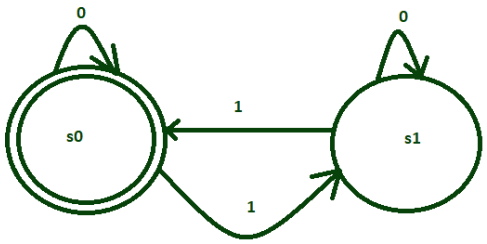
L_1= {01, 001, 101, 0101, 1001, 1101, ...}

L_2= {11, 011, 101, 110, 0011, 1100, ...}

L = L_1 and L_2 = $L_1 \cap L_2$

State Transition Diagram for the language **L_1**

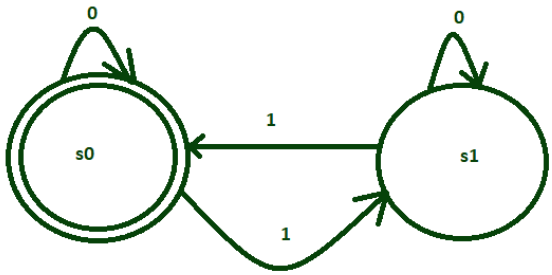
This is a FA for language L_1



It accepts all the string that accept 01 at end.

State Transition Diagram for the language **L_2**

This is a FA for language L_2

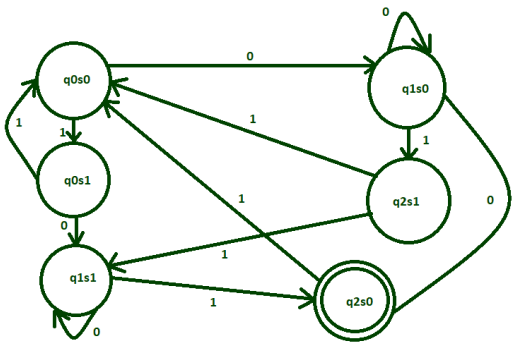


It accepts all the string that accept with even number of 1's.

State Transition Diagram For$L_1 \cap L_2$

State Intersection of L_1 and L_2 can be explained by language that a string over {0, 1} accept such that it ends with 01 and has even number of 1's.

L = $L_1 \cap L_2$
= {1001, 0101, 01001, 10001, ...}



Thus as we see that L_1 and L_2 have been combined through intersection process and this final FA accept all the language that has even number of 1's and is ending with 01.

❖ Regular Language:

The set of regular languages over an alphabet Σ is defined recursively as below. Any language belonging to this set is a **regular language** over Σ.

Definition of set of Regular Languages:

Basis Clause: ,Ø{ Λ } and {a} for any symbol a € Σ are regular languages.

Inductive Clause: If L_r and L_s are regular languages, then $L_r \cup L_s$, L_rL_s and L_r^* are regular languages.

Nothing is a regular language unless it is obtained from the above two clauses.

For example, let Σ = {a, b}. Then since {a} and {b} are regular languages, {a, b} (= {a} $\cup$ {b}) and {ab} (= {a}{b}) are regular languages. Also since {a} is regular, {a}* is a regular language which is the set of strings consisting of a's such as Λ , a, aa, aaa, aaaa etc. Note also that Σ^*, which is the set of strings consisting of a's and b's, is a regular language because {a, b} is regular.

Regular Expression:

Regular expressions are used to denote regular languages. They can represent regular languages and operations on them succinctly.

The set of regular expressions over an alphabet Σ is defined recursively as below. Any element of that set is a **regular expression**.

Basis Clause: Ø Λ and **a** are regular expressions corresponding to languages ,Ø{ Λ } and {a}, respectively,

where a is an element of Σ.

Inductive Clause: If **r** and **s** are regular expressions corresponding to languages L_r and L_s ,then **(r + s),(rs)** and **(r*)** are regular expressions corresponding to languages $L_r \cup L_s$, $L_r L_s$ and L_r^* , respectively.

Nothing is a regular expression unless it is obtained from the above two clauses.

FA is characterized into two types:

1.Deterministic Finite Automata (DFA):

DFA consists of 5 tuples {Q, Σ, q, F, δ}.

Q : set of all states.

Σ : set of input symbols. (Symbols which machine takes as input)

q : Initial state. (Starting state of a machine)

F : set of final state.

δ : Transition Function, defined as δ : Q X Σ --> Q.

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

For example, below DFA with Σ = {0, 1} accepts all strings ending with 0.

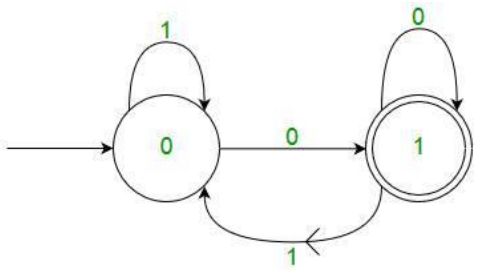


Figure: *DFA with Σ = {0, 1}*

One important thing to note is, ***there can be many possible DFAs for a pattern***. A DFA with a minimum number of states is generally preferred.

2.Nondeterministic Finite Automata(NFA):

NFA is similar to DFA except following additional features:

1. Null (or ε) move is allowed i.e., it can move forward without reading symbols.
2. Ability to transmit to any number of states for a particular input.

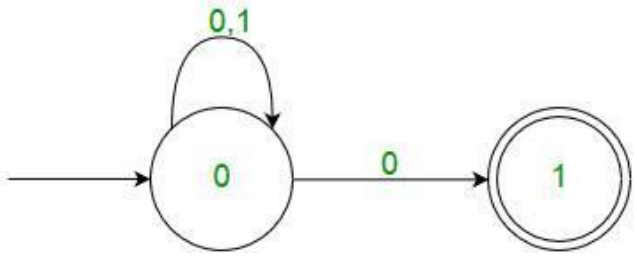
However, these above features don't add any power to NFA. If we compare both in terms of power, both are equivalent.

Due to the above additional features, NFA has a different transition function, the rest is the same as DFA.

δ: Transition Function

δ: Q X (Σ U ε) --> 2 ^ Q.

As you can see in the transition function is for any input including null (or ε), NFA can go to any state number of states. For example, below is an NFA for the above problem.



NFA

One important thing to note is, ***in NFA, if any path for an input string leads to a final state, then the input string* is *accepted***. For example, in the above NFA, there are multiple paths for the input string "00". Since one of the paths leads to a final state, "00" is accepted by the above NFA.

Algorithm for the conversion of Regular Expression to NFA

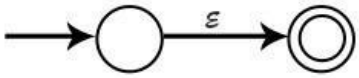
A Regular Expression is a representation of Tokens. But, to recognize a token, it can need a token Recognizer, which is nothing but a Finite Automata (NFA). So, it can convert Regular Expression into NFA.

Input − A Regular Expression R

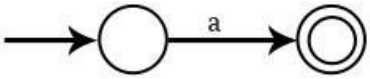
Output − NFA accepting language denoted by R

Method

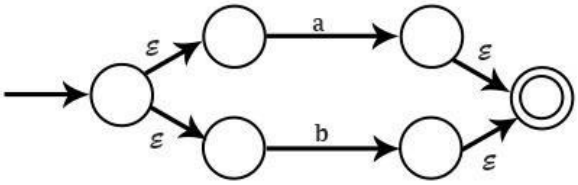
For ε, NFA is



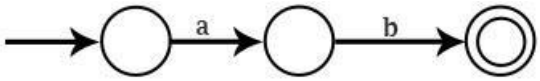
For a NFA is



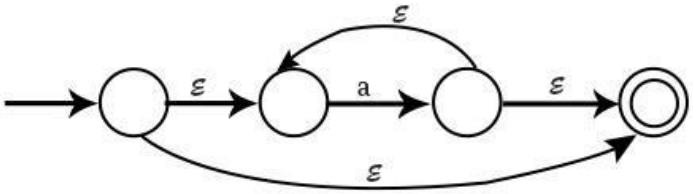
For a + b, or a | b NFA is



For ab, NFA is

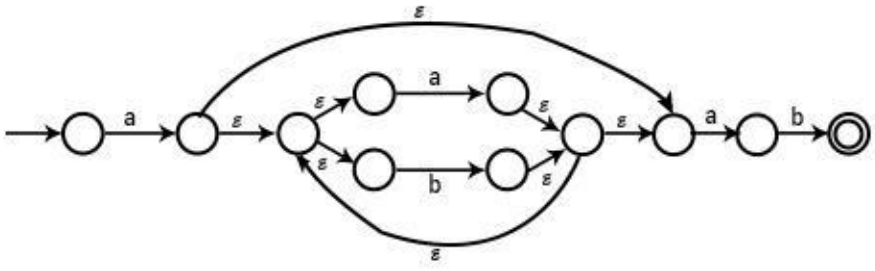


For a*, NFA is



Example1 − Draw NFA for the Regular Expression a(a+b)*ab

Solution



ε−closure (s) − It is the set of states that can be reached form state s on ε−transitions alone.

- If s, t, u states. Initially, ε−closure (s)={s}.
- If s→t, then ε−closure (s)={s,t}.
- If s→t→u, then ε−closure (s)={s,t,u}

It will be repeated until all states are covered.

Algorithm: ε−closure (T)

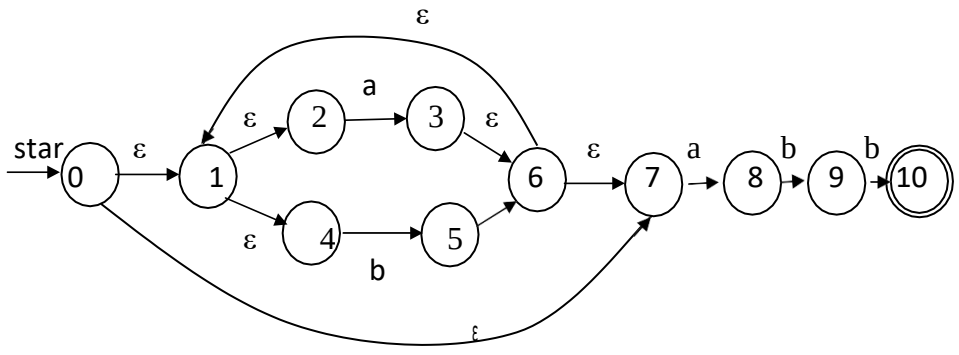
T is a set of states whose ε−closure (s) is to be found.

Push All states in T on the stack

ε −closure (T)=T

```
While (stack not empty) {
  Pop s, the top element of Stack
  for each state t, with edge s→t {
    ift is not present in ε−closure (T) {
      ε−closure (T)=ε−closure (T)∪{t}
      Push t on Stack
    }
  }
}
```

Example: Convert (a|b)*abb. To NFA and DFA



Start the Conversion

1. Begin with the start state 0 and calculate ε-closure(0). a. the set of states reachable by ε-transitions which includes 0 itself is { 0,1,2,4,7}. This defines a new state A in the DFA A = {0,1,2,4,7}
2. We must now find the states that A connects to. There are two symbols in the language (a, b) so in the DFA we expect only two edges: from A on a and from A on b. Call these states B and C:

We find B and C in the following way:

Find the state B that has an edge on a from A

a. start with A{0,1,2,4,7}. Find which states in A have states reachable by a transitions. This set is called move(A,a) The set is {3,8}: move(A,a) = {3,8}

b. now do an ε-closure on move(A,a). Find all the states in move(A,a) which are reachable with ε-transitions. We have 3 and 8 to consider. Starting with 3 we can get to 3 and 6 and from 6 to 1 and 7, and from 1 to 2 and 4. Starting with 8 we can get to 8 only. So the complete set is {1,2,3,4,6,7,8}. So ε-closure(move(A,a)) = B = {1,2,3,4,6,7,8}

This defines the new state B that has an edge on a from A

Find the state C that has an edge on b from A

c. start with A{0,1,2,4,7}. Find which states in A have states reachable by b transitions. This set is called move(A,b) The set is {5}: move(A,b) = {5}

d. now do an ε-closure on move(A,b). Find all the states in move(A,b) which are reachable with ε-transitions. We have only state 5 to consider. From 5 we can get to 5, 6, 7, 1, 2, 4. So the complete set is {1,2,4,5,6,7}. So

    a. ε-closure(move(A,a)) = C = {1,2,4,5,6,7}

This defines the new state C that has an edge on b from A

A={0,1,2,4,7} B={1,2,3,4,6,7,8} C={1,2,4,5,6,7}

Now that we have B and C we can move on to find the states that have a and b transitions from B and C.

Find the state that has an edge on a from B

e. start with B{1,2,3,4,6,7,8}. Find which states in B have states reachable by a transitions. This set is called move(B,a) The set is {3,8}: move(B,a) = {3,8}

f. now do an ε-closure on move(B,a). Find all the states in move(B,a) which are reachable with ε-transitions. We have 3 and 8 to consider. Starting with 3 we can get to 3 and 6 and from 6 to 1 and 7, and from 1 to 2 and 4. Starting with 8 we can get to 8 only. So the complete set is {1,2,3,4,6,7,8}. So ε-closure(move(A,a)) = {1,2,3,4,6,7,8}

which is the same as the state B itself. In other words, we have a repeating edge to B:

A={0,1,2,4,7} B={1,2,3,4,6,7,8} C={1,2,4,5,6,7}

Find the state D that has an edge on b from B

g. start with B{1,2,3,4,6,7,8}. Find which states in B have states reachable by b transitions. This set is called move(B,b) The set is {5,9}: move(B,b) = {5,9}

h. now do an ε-closure on move(B,b). Find all the states in move(B,b) which are reachable with ε-transitions. From 5 we can get to 5, 6, 7, 1, 2, 4. From 9 we get to 9 itself. So the complete set is {1,2,4,5,6,7,9}}. So

ε-closure(move(B,a)) = D = {1,2,4,5,6,7,9} This defines the new state D that has an edge on b from B

A={0,1,2,4,7}, B={1,2,3,4,6,7,8}, C={1,2,4,5,6,7}, D{1,2,4,5,6,7,9}

Find the state that has an edge on a from D

i. start with D{1,2,4,5,6,7,9}. Find which states in D have states reachable by a transitions. This set is called move(D,a) The set is {3,8}: move(D,a) = {3,8}

j. now do an ε-closure on move(D,a). Find all the states in move(B,a) which are reachable with ε-transitions. We have 3 and 8 to consider. Starting with 3 we can get to 3 and 6 and from 6 to 1 and 7, and from 1 to 2 and 4. Starting with 8 we can get to 8 only. So the complete set is {1,2,3,4,6,7,8}. So ε-closure(move(D,a)) = {1,2,3,4,6,7,8} =B

This is a return edge to B:

A={0,1,2,4,7}, B={1,2,3,4,6,7,8}, C={1,2,4,5,6,7}, D{1,2,4,5,6,7,9}

Find the state E that has an edge on b from D

k. start with D{1,2,4,5,6,7,9}. Find which states in D have states reachable by b transitions. This set is called move(B,b) The set is {5,10}: move(D,b) = {5,10}

l. now do an ε-closure on move(D,b). Find all the states in move(D,b) which are reachable with ε-transitions. From 5 we can get to 5, 6, 7, 1, 2, 4. From 10 we get to 10 itself. So the complete set is {1,2,4,5,6,7,10}}. So

ε-closure(move(D,b) = E = {1,2,4,5,6,7,10}

This defines the new state E that has an edge on b from D. Since it contains an accepting state, it is also an accepting state.

A={0,1,2,4,7}, B={1,2,3,4,6,7,8}, C={1,2,4,5,6,7}, D={1,2,4,5,6,7,9}, E={1,2,4,5,6,7,10}

We should now examine state C

Find the state that has an edge on a from C

m. start with C{1,2,4,5,6,7}. Find which states in C have states reachable by a transitions. This set is called move(C,a) The set is {3,8}:

move(C,a) = {3,8}

we have seen this before. It's the state B

A={0,1,2,4,7}, B={1,2,3,4,6,7,8}, C={1,2,4,5,6,7}, D={1,2,4,5,6,7,9}, E={1,2,4,5,6,7,10}

Find the state that has an edge on b from C

n. start with C{1,2,4,5,6,7}. Find which states in C have states reachable by b transitions. This set is called move(C,b) The set is {5}:

o. move(C,b) = {5}

p. now do an ε-closure on move(C,b). Find all the states in move(C,b) which are reachable with ε-transitions. From 5 we can get to 5,6,7,1,2,4. which is C itself So

ε-closure(move(C,b)) = C

This defines a loop on C

Finally we need to look at E. Although this is an accepting state, the regular expression allows us to repeat adding in more a's and b's as long as we return to the accepting E state finally. So

Find the state that has an edge on a from E

q. start with E{1,2,4,5,6,7,10}. Find which states in E have states reachable by a transitions. This set is called move(E,a) The set is {3,8}:

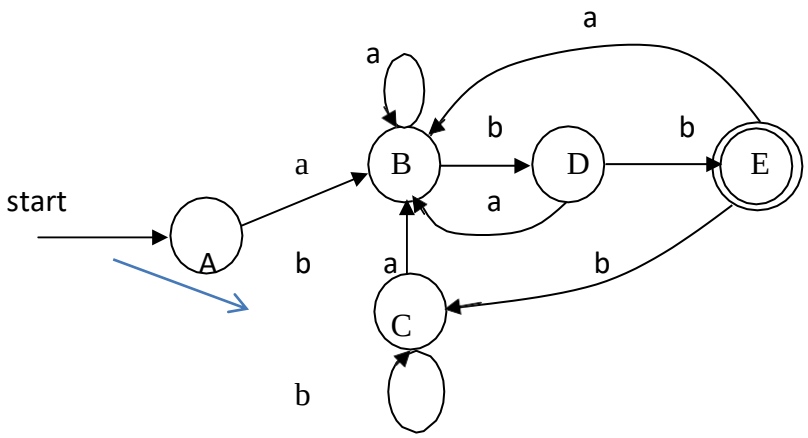
move(E,a) = {3,8} We saw this before, it's B

Find the state that has an edge on b from E

r. start with E{1,2,4,5,6,7,10}. Find which states in E have states reachable by b transitions. This set is called move(E,b) The set is {5}:

move(A,b) = {5}

We've seen this before. It's C. Finally



That's it ! There is only one edge from each state for a given input character. It's a DFA. Disregard the fact that each of these states is actually a group of NFA states. We can regard them as single states in the DFA. In fact it also requires other as an edge beyond E leading to the ultimate accepting state. Also the DFA is not yet optimized (there can be less states).

However, we can make the transition table so far. Here it is:

DFA:

ε – closure (0) = {0,1,2,4,7} —- Let A

Move(A,a) = {3,8}

ε – closure (Move(A,a)) = {1,2,3,4,6,7,8}—- Let B

Move(A,b) = {5}

ε – closure (Move(A,b)) = {1,2,4,5,6,7}—- Let C

Move(B,a) = {3,8}

ε – closure (Move(B,a)) = {1,2,3,4,6,7,8}—- B

Move(B,b) = {5,9}

ε – closure (Move(B,b)) = {1,2,4,5,6,7,9}—- Let D

Move(C,a) = {3,8}

ε – closure (Move(C,a)) = {1,2,3,4,6,7,8}—- B

Move(C,b) = {5}

ε – closure (Move(C,b)) = {1,2,4,5,6,7}—- C

Move(D,a) = {3,8}

ε – closure (Move(D,a)) = {1,2,3,4,6,7,8}—- B

Move(D,b) = {5,10}

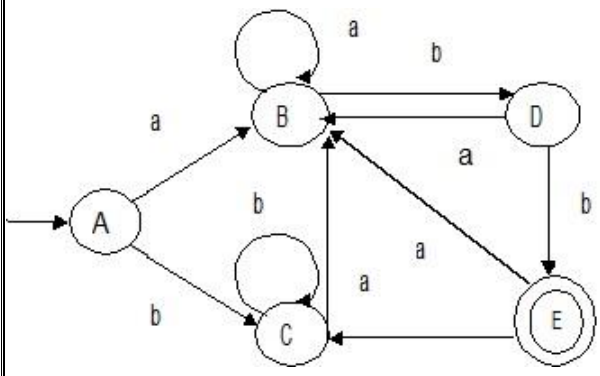
ε – closure (Move(D,b)) = {1,2,4,5,6,7,10}—- Let E

Move(E,a) = {3,8}

ε – closure (Move(E,a)) = {1,2,3,4,6,7,8}—- B

Move(E,b) = {5}

ε – closure (Move(E,b)) = {1,2,4,5,6,7}—- C



State	Input a	Input b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

❖ **Context free grammar**

Derivations trees:

- Derivation tree is a graphical representation for the derivation of the given production rules of the context free grammar (CFG).
- It is a way to show how the derivation can be done to obtain some string from a given set of production rules. It is also called as the Parse tree.
- The Parse tree follows the precedence of operators.
- The deepest subtree is traversed first. So, the operator in the parent node has less precedence over the operator in the subtree.

Properties

The properties of the derivation tree are given below −

- The root node is always a node indicating the start symbols.
- The derivation is read from left to right.
- The leaf node is always the terminal node.
- The interior nodes are always the non-terminal nodes.

Example

The production rules for the derivation tree are as follows −

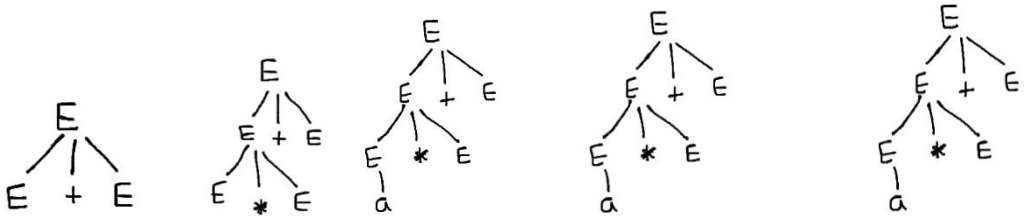
E=E+E

E=E*E

E=a|b|c

Here, let the input be a*b+c

Step 1: Step 2: Step 3: Step 4: Step 5



Ambiguity Grammar:

If a context free grammar **G** has more than one derivation tree for some string **w ∈L(G)**, it is called an **ambiguous grammar**. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Problem

Check whether the grammar G with production rules −

X → X+X | X*X |X| a

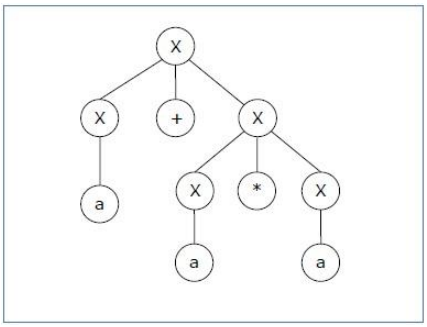
is ambiguous or not.

Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

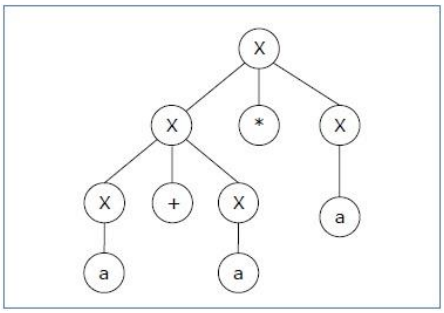
Derivation 1 − X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 −



Derivation 2 − X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 −



Since there are two parse trees for a single string "a+a*a", the grammar **G** is ambiguous.

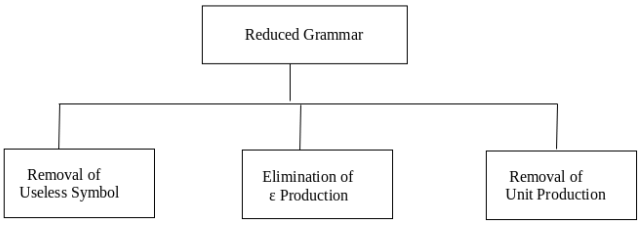
- **Simplified Forms:**

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 reduction process



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

S → XYX

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

S → YX

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

S → XY

If Y = ε then

S → XX

If Y and X are ε then,

$S \rightarrow X$

If both X are replaced by ε

$S \rightarrow Y$ Now,

$S \rightarrow XY \mid YX \mid XX \mid X \mid Y$

Now let us consider

$X \rightarrow 0X$

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

$X \rightarrow 0$

$X \rightarrow 0X \mid 0$

Similarly $Y \rightarrow 1Y \mid 1$

Collectively we can rewrite the CFG with removed ε production as

$S \rightarrow XY \mid YX \mid XX \mid X \mid Y$

$X \rightarrow 0X \mid 0$

$Y \rightarrow 1Y \mid 1$

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 $X \rightarrow Y$, add production $X \rightarrow a$ to the grammar rule whenever $Y \rightarrow a$ occurs in the grammar.

Step 2: Now delete $X \rightarrow Y$ from the grammar.

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

For example:

S → 0A | 1B | C

A → 0S | 00

B → 1 | A

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.

S → 0A | 1B | 01

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

B → 1 | 0S | 00

Thus finally we can write CFG without unit production as

S → 0A | 1B | 01

A → 0S | 00

B → 1 | 0S | 00

C → 01

- ❖ **Normal Forms**

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:
G1 = {S → AB, S → c, A → a, B → b}
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:

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:

S → RA
R → a

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

S → RS
R → AS

Example:

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

S → a | aA | B
A → aBB | ε
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:

S1 → S
S → a | aA | B
A → aBB | ε
B → Aa | b

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

S1 → S
S → a | aA | B
A → aBB
B → Aa | b | a

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

S1 → S
S → a | aA | Aa | b
A → aBB
B → Aa | b | a

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

S0 → a | aA | Aa | b
S → a | aA | Aa | b
A → aBB
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:

S0 → a | XA | AX | b
S → a | XA | AX | b
A → XBB
B → AX | b | a
X → a

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

S0 → a | XA | AX | b
S → a | XA | AX | b
A → RB
B → AX | b | a
X → a
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:

G1 = {S → aAB | aB, A → aA| a, B → bB | b}
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:

S → XB | AA
A → a | SA
B → b
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:

S → XB | AA
A → a | XBA | AAA
B → b
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:

S → aB | AA
A → a | aBA | AAA
B → b
X → a

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

S → aB | AA
A → aC | aBAC
C → AAC | ε
B → b
X → a

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

S → aB | AA
A → aC | aBAC | a | aBA
C → AAC | AA
B → b
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:

S → aB | aCA | aBACA | aA | aBAA
A → aC | aBAC | a | aBA
C → AAC
C → aCA | aBACA | aA | aBAA
B → b
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:

S → aB | aCA | aBACA | aA | aBAA
A → aC | aBAC | a | aBA
C → aCAC | aBACAC | aAC | aBAAC
C → aCA | aBACA | aA | aBAA
B → b
X → a

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

UNIT II

Introduction to Compiler

❖ INTRODUCTION TO LANGUAGE PROCESSING

- As Computers became inevitable and indigenous part of human life, and several languages withdifferent and more advanced features are evolved into this stream to satisfy or comfort the user in communicating with the machine , the development of the translators or mediator Software‘s have become essential to fill the huge gap between the human and machine understanding.

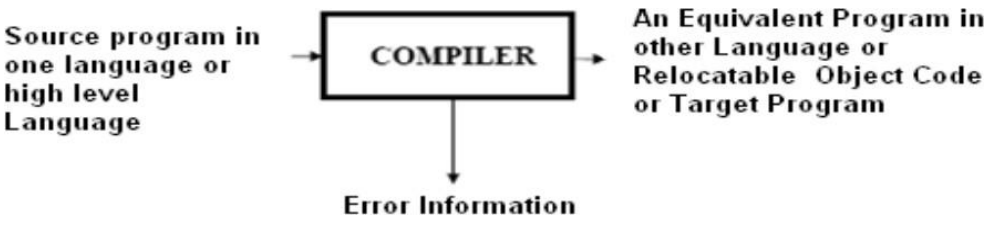
- This process is called Language Processing to reflect the goal and intent of the process. On the way to this process to understand it in a better way, we have to be familiar with some key terms and concepts explained in following lines.

❖ LANGUAGE TRANSLATORS

Is a computer program which translates a program written in one (Source) language to its equivalent program in other [Target]language. The Source program is a high level language where as the Target language can be any thing from the machine language of a target machine (between Microprocessor to Supercomputer) to another high level language program.

Two commonly Used Translators are Compiler and Interpreter

1. **Compiler :** Compiler is a program, reads program in one language called Source Language and translates in to its equivalent program in another Language called Target Language, in addition to this its presents the error information to the User.



If the target program is an executable machine-language program, it can then be called by the users to process inputs and produce outputs.

2. **Interpreter:** An interpreter is another commonly used language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

❖ LANGUAGE PROCESSING SYSTEM:

Based on the input the translator takes and the output it produces, a language translator can be called as any one of the following.

Preprocessor: A preprocessor takes the skeletal source program as input and produces an extended version of it, which is the resultant of expanding the Macros, manifest constants if any, and including header files etc in the source file. For example, the C preprocessor is a macro processor that is used automatically by the C compiler to transform our source before actual compilation. Over and above a preprocessor performs the following activities:

- Collects all the modules, files in case if the source program is divided into different modules stored at different files.
- Expands short hands / macros into source language statements.

Compiler: Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. In addition to above the compileralso

- Reports to its user the presence of errors in the source program.
- Facilitates the user in rectifying the errors, and execute the code.

Assembler: Is a program that takes as input an assembly language program and converts it into itsequivalent machine language code.

Loader / Linker: This is a program that takes as input a relocatable code and collects the libraryfunctions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

- **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.

- **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.

In addition to these translators, programs like interpreters, text formatters etc., may be used in language processing system. To translate a program in a high level language program to an executable one, the Compiler performs by default the compile and linking functions.

Normally the steps in a language processing system includes Preprocessing the skeletal Source program which produces an extended or expanded source program or a ready to compile unit of the source program, followed by compiling the resultant, then linking / loading , and finally its equivalent executable code is produced. As I said earlier not all these steps are mandatory. In some cases, the Compiler only performs this linking and loading functions implicitly.

❖ TYPES OF COMPILERS:

Based on the specific input it takes and the output it produces, the Compilers can be classifiedinto the following types;

Traditional Compilers(C, C++, Pascal): These Compilers convert a source program in a HLLinto its equivalent in native machine code or object code.

Interpreters(LISP, SNOBOL, Java1.0): These Compilers first convert Source code into intermediate code, and then interprets (emulates) it to its equivalent machine code.

Cross-Compilers: These are the compilers that run on one machine and produce code foranother machine.

Incremental Compilers: These compilers separate the source into user defined–steps;Compiling/recompiling step- by- step; interpreting steps in a given order

Converters (e.g. COBOL to C++): These Programs will be compiling from one high level language to another.

Just-In-Time (JIT) Compilers (Java, Micosoft.NET): These are the runtime compilers from intermediate language (byte code, MSIL) to executable code or native machine code. These perform type –based verification which makes the executable code more trustworthy

Ahead-of-Time (AOT) Compilers (e.g., .NET ngen): These are the pre-compilers to the native code for Java and .NET

Binary Compilation: These compilers will be compiling object code of one platform into objectcode of another platform**.**

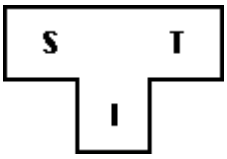
❖ BOOTSTRAPING

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler tocompile everything else as well as future versions of itself.

A compiler can be characterized by three languages:

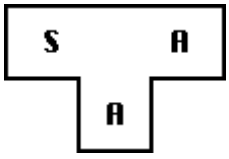
1. Source Language
2. Target Language
3. Implementation Language

The T- diagram shows a compiler ${}^{S}C_{I}{}^{T}$ for Source S, Target T, implemented in I.

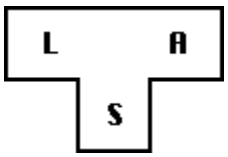


Follow some steps to produce a new language **L** for machine **A**:

1. Create a compiler ${}^{S}C_{A}{}^{A}$ for subset, S of the desired language, L using language "A" and that compiler runs on machine A.

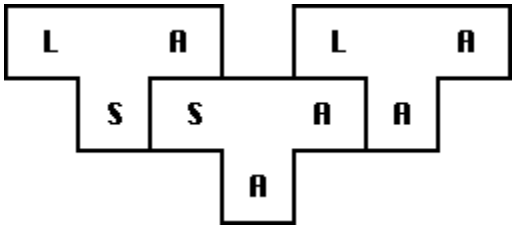


2. Create a compiler ${}^{L}C_{S}{}^{A}$ for language L written in a subset of L



3. Compile $^{L}C^{A}_{S}$ using the compiler $^{S}C^{A}_{A}$ to obtain $^{L}C^{A}_{A}$. $^{L}C^{A}_{A}$ is a compiler for language L, which runs on machine A and produces code for machine A.

$$^{L}C^{A}_{S} \rightarrow {}^{S}C^{A}_{A} \rightarrow {}^{L}C^{A}_{A}$$



The process described by the T-diagrams is called bootstrapping.

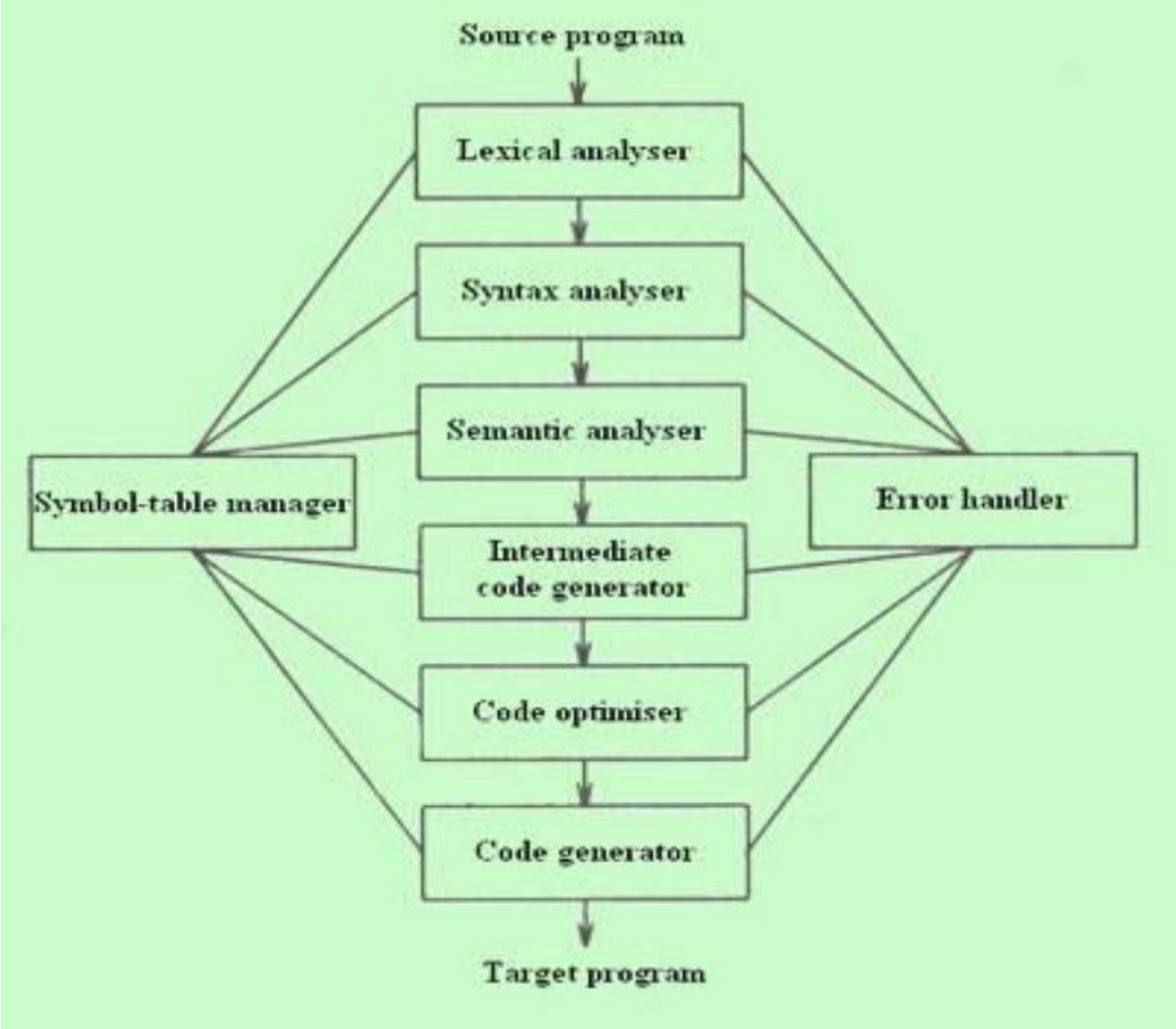
❖ PHASES OF A COMPILER:

- Due to the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces. Generally an interface contains a Data structure (e.g., tree), Set of exported functions. Each phase works on an abstract **intermediate representation** of the source program, not the source program text itself (except the first phase)
- Compiler Phases are the individual modules which are chronologically executed to perform their respective Sub-activities, and finally integrate the solutions to give target code.
- It is desirable to have relatively few phases, since it takes time to read and write immediate files. Following diagram (Figure1.4) depicts the phases of a compiler through which it goes during the compilation. There fore a typical Compiler is having the following Phases:

1. Lexical Analyzer (Scanner),
2. Syntax Analyzer (Parser),
3. Semantic Analyzer,
4. Intermediate Code Generator(ICG),
5. Code Optimizer(CO) , and
6. Code Generator(CG)

In addition to these, it also has **Symbol table management**, and **Error handler** phases. Not all the phases are mandatory in every Compiler. e.g, Code Optimizer phase is optional in some cases. The description is given in next section.

The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.



PHASE, PASSES OF A COMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely deferent representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

THE FRONT-END & BACK-END OF A COMPILER

- All of these phases of a general Compiler are conceptually divided into **The Front-end**, and **The Back-end**. This division is due to their dependence on either the Source Language or the Target machine. This model is called an Analysis & Synthesis model of a compiler.
- The **Front-end** of the compiler consists of phases that depend primarily on the Source language and are largely independent on the target machine. For example, front-end of the compiler includes Scanner, Parser, Creation of Symbol table, Semantic Analyzer, and the Intermediate Code Generator.
- The **Back-end** of the compiler consists of phases that depend on the target machine, and those portions don't dependent on the Source language, just the Intermediate language. In this we have different aspects of Code Optimization phase, code generation along with the necessary Error handling, and Symbol table operations.

LEXICAL ANALYZER (SCANNER): The Scanner is the first phase that works as interface between the compiler and the Source language program and performs the following functions:

- Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as an identifier , a Keyword , a punctuation mark, a multi character operator like := .
- The character sequence forming a token is called a **lexeme** of the token.
- The Scanner generates a token-id, and also enters that identifiers name in the Symboltable if it doesn't exist.
- Also removes the Comments, and unnecessary spaces.

The format of the token is **< Token name, Attribute value>**

SYNTAX ANALYZER (PARSER): The Parser interacts with the Scanner, and its subsequent phase Semantic Analyzer and performs the following functions:

- Groups the above received, and recorded token stream into syntactic structures, usually into a structure called **Parse Tree** whose leaves are tokens.

- The interior node of this tree represents the stream of tokens that logically belongstogether.
- It means it checks the syntax of program elements.

SEMANTIC ANALYZER: This phase receives the syntax tree as input, and checks the semantically correctness of the program. Though the tokens are valid and syntactically correct, it may happen that they are not correct semantically. Therefore the semantic analyzer checks the semantics (meaning) of the statements formed.

- The Syntactically and Semantically correct structures are produced here in the form of aSyntax tree or DAG or some other sequential representation like matrix.

INTERMEDIATE CODE GENERATOR(ICG): This phase takes the syntactically and semantically correct structure as input, and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:

- It should be easy to produce, and Easy to translate into the target program. Exampleintermediate code forms are:
- Three address codes,
- Polish notations, etc.

CODE OPTIMIZER: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:

- Attempts to improve the IC so as to have a faster machine code. Typical functions include –Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductions etc.
- Sometimes the data structures used in representing the intermediate forms may also bechanged.

CODE GENERATOR: This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machine code.

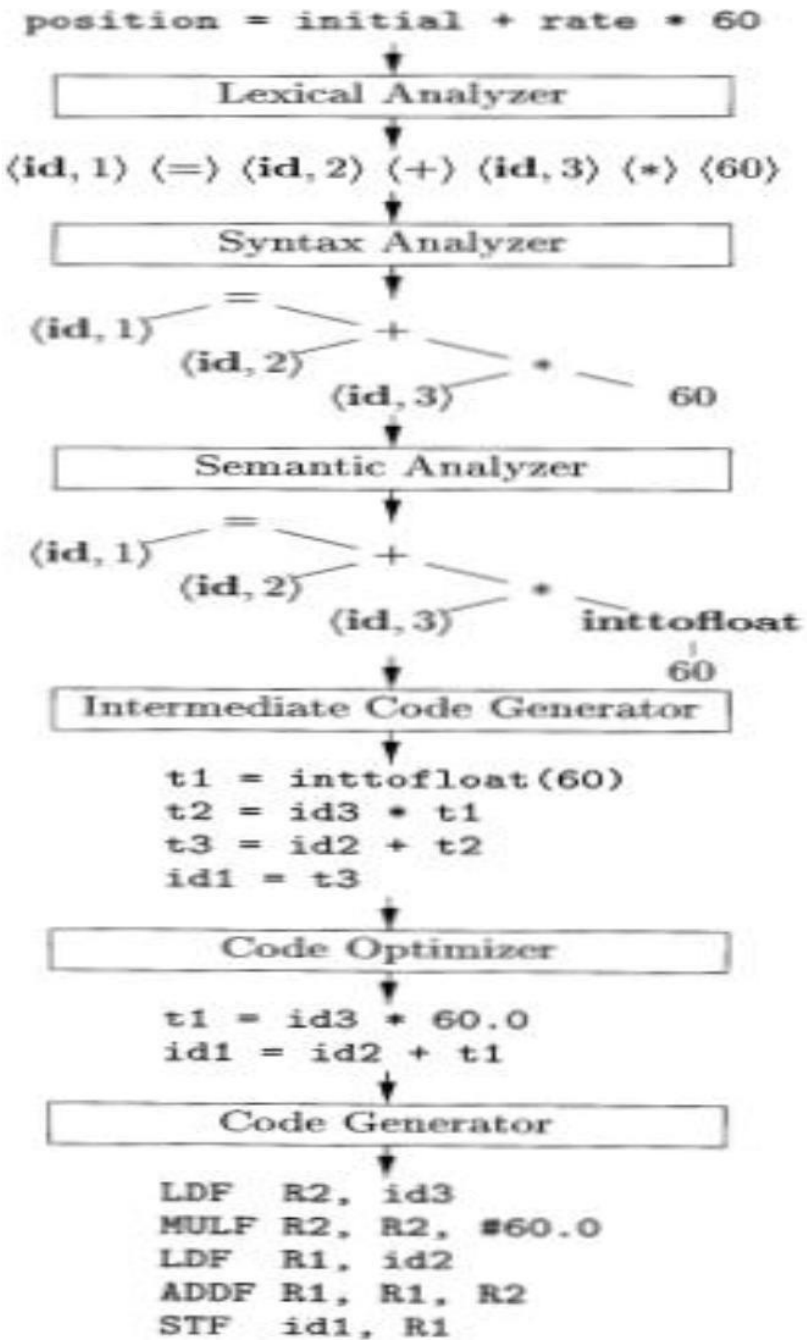
- Memory locations are selected for each variable used, and assignment of variables toregisters is done.
- Intermediate instructions are translated into a sequence of machine instructions.

The Compiler also performs the **Symbol table management** and **Error handling** throughout thecompilation process. Symbol table is nothing but a data structure that stores different source

language constructs, and tokens generated during the compilation. These two interact with allphases of the Compiler.

For example the source program is an assignment statement; the following figure shows howthephases of compiler will process the program.

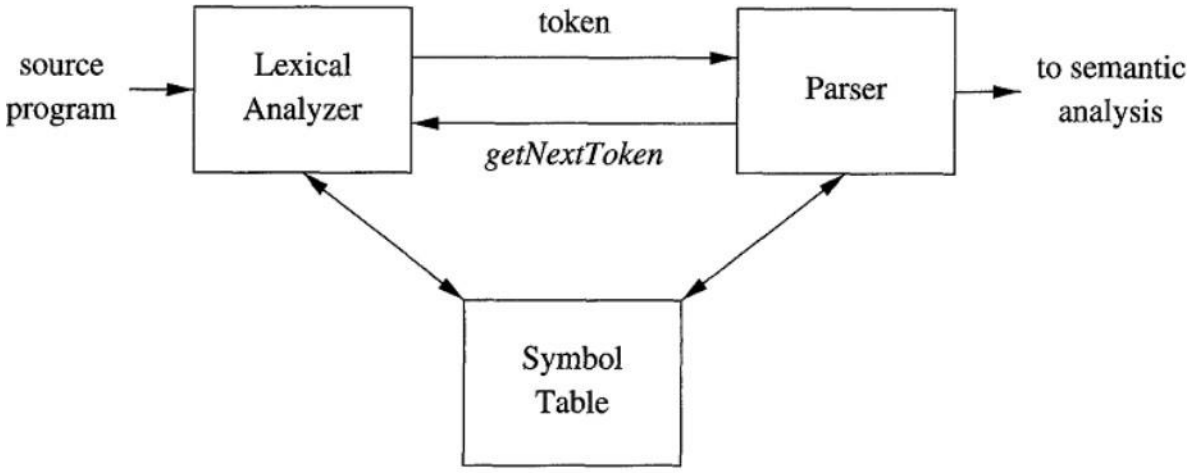
The input source program is **Position=initial+rate*60**



LEXICAL ANALYSIS:

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output tokens for each lexeme in the source program. This stream of tokens is sent to the parser for syntax analysis. It is common for the lexical analyzer to interact with the symbol table as well.

When the lexical analyzer discovers a lexeme constituting an identifier, it needs to enter that lexeme into the symbol table. This process is shown in the following figure.



When lexical analyzer identifies the first token it will send it to the parser, the parser receives the token and calls the lexical analyzer to send next token by issuing the **getNextToken()** command. This Process continues until the lexical analyzer identifies all the tokens. During this process the lexical analyzer will neglect or discard the white spaces and comment lines.

TOKENS, PATTERNS AND LEXEMES:

A token is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

A pattern is a description of the form that the lexemes of a token may take [or match].

In the case of a keyword as a token, the pattern is just the sequence of characters that form the

keyword. For identifiers and some other tokens, the pattern is a more complex structure that ismatched by many strings.

A lexeme is a sequence of characters in the source program that matches the patternfor atoken and is identified by the lexical analyzer as an instance of that token.

Example: In the following C language

statement ,printf ("Total = %d\n‖,
score) ;

both **printf** and **score** are lexemes matching the **pattern** for token **id**, and **"Total = %d\n‖**is a lexeme matching **literal [or string]**.

LEXICAL ANALYSIS Vs PARSING

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

There are a number of reasons why the analysis portion of a compiler is normally separated intolexical analysis and parsing (syntax analysis) phases.

1. **Simplicity of design is the most important consideration.** The separation of Lexical and Syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

2. **Compiler efficiency is improved**. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.

3. **Compiler portability is enhanced**: Input-device-specific peculiarities can be restricted to the lexical analyzer.

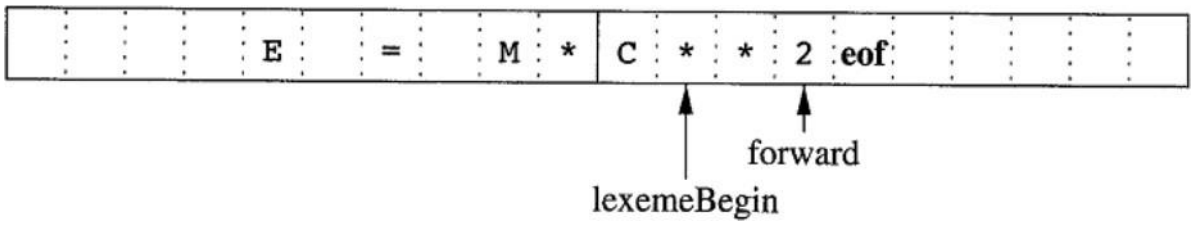
❖ INPUT BUFFERING:

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id. In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of

characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.



Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read N characters in to a buffer, rather than using one system call per character. If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file and is different from any possible character of the source program. Two pointers to the input are maintained:

1. The Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extentwe are attempting to determine.

2. Pointer **forward** scans ahead until a pattern match is found; the exact strategywhereby this determination is made will be covered in the balance of this chapter.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, we see forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

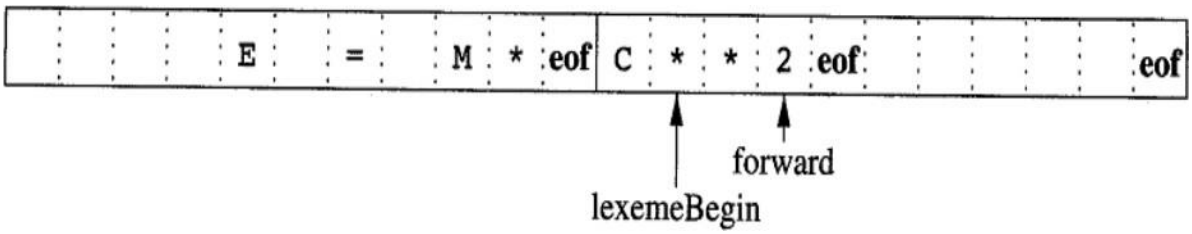
Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.

As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greaterthan N, we shall never overwrite the lexeme in its buffer before determining it.

Sentinels To Improve Scanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and oneto determine what character is read (the latter may be a multi way branch).

We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure shows the same arrangement as Figure above but with the sentinels added. Note that eof retains its use as amarker for the end of the entire input.



Any eof that appears other than at the end of a buffer means that the input is at an end. Below Figure summarizes the algorithm for advancing forward. Notice how the first test, which can be part of a multiway branch based on the character pointed to by forward, is

the only test we make, except

in the case where we actually are at the end of a buffer or the end of the

```
 input.switch ( *forward++  )
 {
 case eof: if (forward is at end of first buffer ){
                    reload second buffer;
                    forward = beginning of second buffer;}
                  else if (forward is at end of second buffer
                  ) {

                    reload first buffer;
                    forward = beginning of first buffer;
                  }
                    else  /* eof within a buffer marks the end of
                     input */terminate lexical analysis;
                 break;
                 }
```

❖ SPECIFICATION OF TOKENS:

Let us understand how the language theory undertakes the following terms:

1. Alphabets
2. Strings
3. Special symbols
4. Language
5. Longest match rule
6. Operations
7. Notations
8. Representing valid tokens of a language in regular expression
9. Finite automata

1. Alphabets: Any finite set of symbols

- {0,1} is a set of binary alphabets,
- {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets,
- {a-z, A-Z} is a set of English language alphabets.

2. Strings: Any finite sequence of alphabets is called a string.

3. Special symbols: A typical high-level language contains the following symbols:

Arithmetic Symbols	Addition(+), Subtraction(-), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.)
Assignment	=
Special assignment	+=, -=, *=, /=
Comparison	==, !=. <. <=. >, >=
Preprocessor	#

4. Language: A language is considered as a finite set of strings over some finite set of alphabets.

5. Longest match rule: When the lexical analyzer read the source-code, it scans the code letterby letter and when it encounters a whitespace, operator symbol, or special symbols it decides that a word is completed.

6. Operations: The various operations on languages are:

- Union of two languages L and M is written as, L U M = {s | s is in L or s is in M}
- Concatenation of two languages L and M is written as, LM = {st | s is in L and t is in M}
- The Kleene Closure of a language L is written as, L* = Zero or more occurrence of language L.

7. Notations: If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union** : L(r)UL(s)
- **Concatenation** : L(r)L(s)
- **Kleene closure** : (L(r))*

8. Representing valid tokens of a language in regular expression: If x is a regular expression, then :

- x* means zero or more occurrence of x.
- x+ means one or more occurrence of x.

9. Finite automata: Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly.If the input string is successfully processed and the automatareaches its final state, it is accepted.The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔ Q

❖ RECOGNITION OF TOKENS

Starting point is the language grammar to understand the

tokens:stmt -> if expr then stmt | if expr then stmt

else stmt | Ɛ expr -> term relop term | term

term -> id | number

The next step is to formalize the

patterns:digit -> [0-9]

Digits -> digit+

number -> digit(.digits)? (E[+-]?

Digit)?letter -> [A-Za-z_]

id -> letter

(letter|digit)*If -> if

Then ->

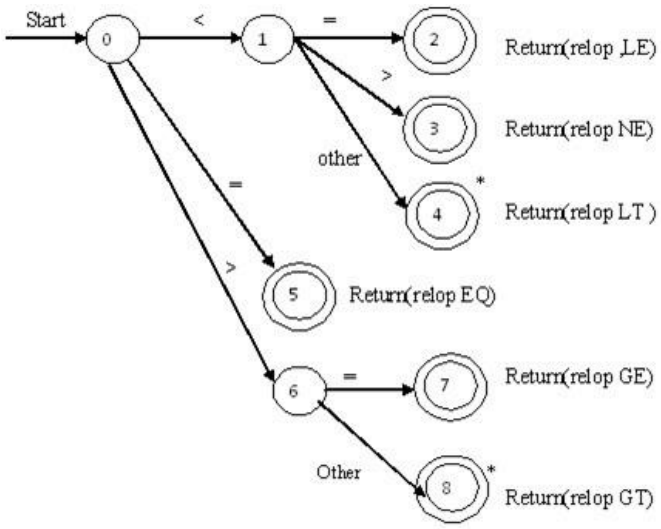
thenElse ->

else

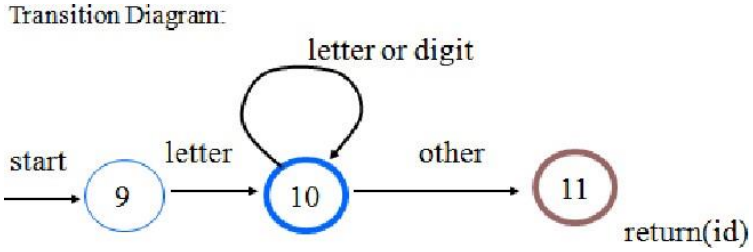
Relop -> < | > | <= | >= | = | <>

We also need to handle whitespaces: ws -> (blank | tab | newline)+

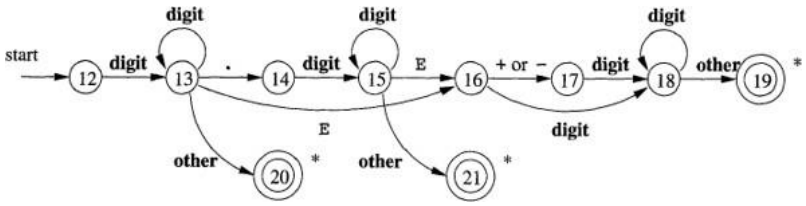
Transition diagram for relop



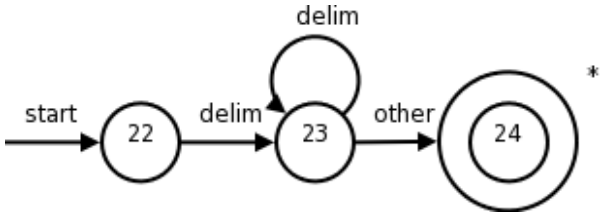
Transition diagram for reserved words and identifiers



Transition diagram for unsigned numbers



Transition diagram for whitespace



❖ A LANGUAGE FOR SPECIFYING LEXICAL ANALYZERS

There is a wide range of tools for constructing lexical analyzers.

Lex

YACC

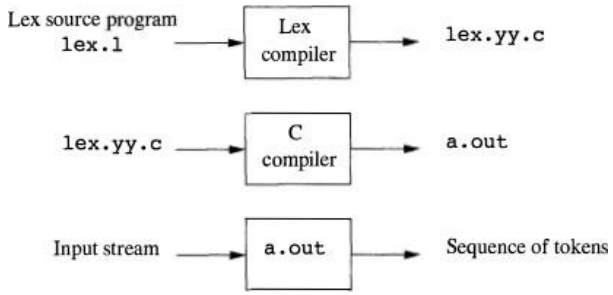
Lex is a computer program that generates lexical analyzers. Lex is commonly used withthe yacc parser generator.

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 oftokens.
- It reads the input stream and produces the source code as output through implementingthe 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 Specification

A Lex program consists of three parts:

{ definitions }

%%

{ rules }

%%

{ user subroutines }

Definitions include declarations of variables, constants, and regular definitions

Rules are statements of the

form p_1 {$action_1$}

p_2 {$action_2$}

…

p_n {$action_n$}

where p_i is regular expression and $action_i$ describes what action the lexical analyzer should takewhen pattern p_i matches a lexeme. Actions are written in C code.

User subroutines are auxiliary procedures needed by the actions. These can becompiled separately and loaded with the lexical analyzer.

YACC- YET ANOTHER COMPILER-COMPILER

- 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.

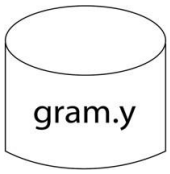
These are some points about YACC:

Input: A CFG- file.y

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

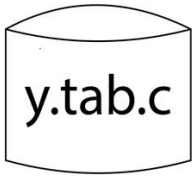
- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

The basic operational sequence is as follows:

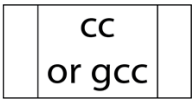


This file contains the desired grammar in YACC format.

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

❖ DESIGN OF LEXICAL ANALYSER:

A *lexical analyzer generator* systematically translates regular expressions to *NFA* which is then translated to an efficient *DFA*.

1 The Structure of the Generated Analyzer

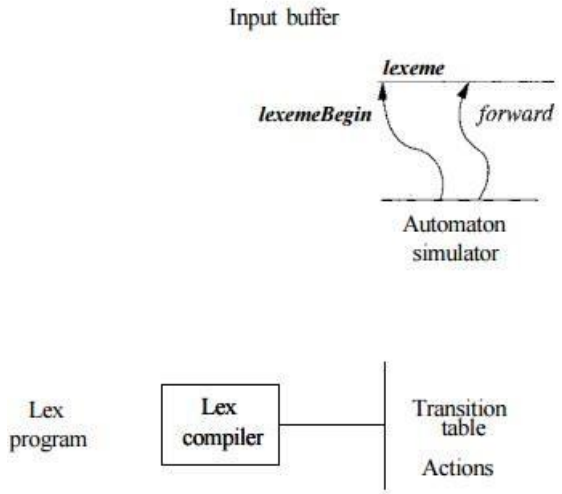
2 Pattern Matching Based on NFA's

3 DFA's for Lexical Analyzers

4 Implementing the Lookahead Operator

1. The Structure of the Generated Analyzer

Below Figure Overviews the architecture of a lexical analyzer generated by Lex. The program that serves as the lexical analyzer includes a fixed program that simulates an automaton; at this point we leave open whether that automaton is deterministic or nondeterministic. The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself.



In above diagram explains a Lex program is turned into a transition table and actions, which are used by a finite-automaton simulator

These components are:

- A transition table for the automaton.
- Those functions that are passed directly through Lex to the output
- The actions from the input program, which appear as fragments of code to beinvoked at the appropriate time by the automaton simulator.

To construct the automaton, we begin by taking each regular-expression pattern in the Lex program and converting it to an NFA. We need a single automaton that will recognize lexemes matching any of the patterns in the program, so we combine all the NFA's into one by introducing a new start state with e-transitions to each of the start states of the NFA's *N{* for pattern *pi.* This construction is shown in Fig. 3.50.

Example: We shall illustrate the ideas of this section with the following simple, abstract example:

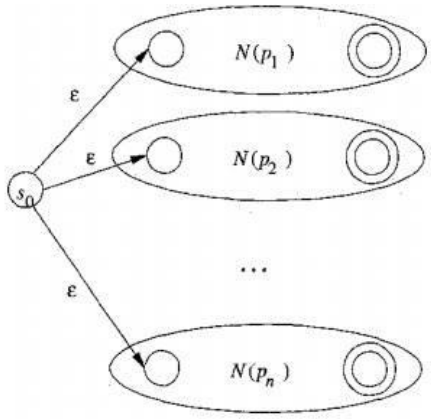


Figure 3.50: An NFA constructed from a **Lex** program

```
a       { action A1 for pattern p1 }
abb     { action A2 for pattern p2 }
a*b+    { action A3 for pattern p3 }
```

Note that these three patterns present some conflicts of the type discussed in earlier. In particular, string abb matches both the second and third patterns, but we shall consider it a lexeme for pattern p2, since that pattern is listed first in the above Lex program. Then, input strings such as aabbb • • • have many prefixes that match the third pattern. The Lex rule is to take the longest, so we continue reading 6's, until another a is met, whereupon we report the lexeme to be the initial a's followed by as many 6's as there are.

Three NFA's that recognize the three patterns. The third is a simplification of what would come out of Algorithm. Then, Fig. 3.52 shows these three NFA's combined into a single NFA by the addition of start state 0 and three e-transitions. •

2. Pattern Matching Based on NFA's

If the lexical analyzer simulates an NFA such as that of Fig. 3.52, then it must read input beginning at the point on its input which we have referred to as *lexemeBegin.* As it moves the pointer called *forward* ahead in the input, it calculates the set of states it is in at each point, following Algorithm.

Eventually, the NFA simulation reaches a point on the input where there are no next states. At that point, there is no hope that any longer prefix of the input would ever get the NFA to an accepting state; rather, the set of states will always be empty. Thus,we are ready to decide on the longest prefix that is a lexeme matching some pattern.

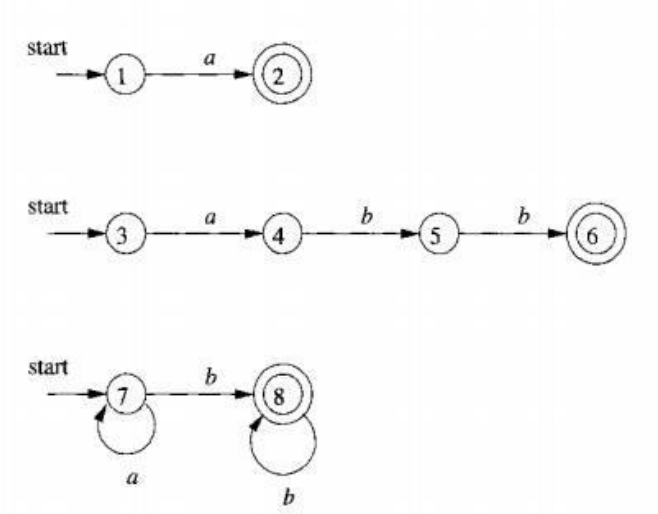


Figure 3.51: NFA's for **a**, **abb**, and **a*b+**

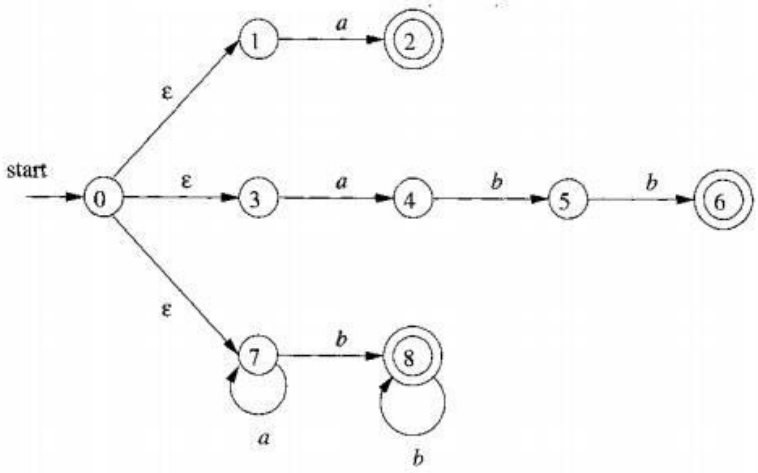


Figure 3.52: Combined NFA

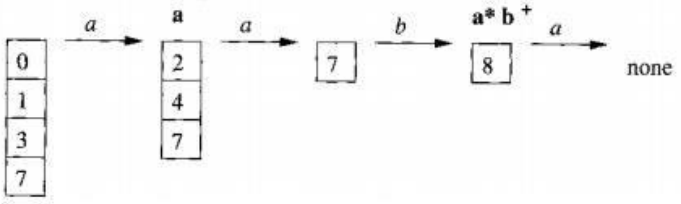


Figure 3.53: Sequence of sets of states entered when processing input *aaba*

We look backwards in the sequence of sets of states, until we find a set that includes one or more accepting states. If there are several accepting states in that set, pick the one associated with the earliest pattern *pi* in the list from the **Lex** program. Move the *forward* pointer back to the end of the lexeme, and perform the action *Ai* associated with pattern *pi.*

Example: Suppose we have the patterns of the input begins *aaba.* Figure 3.53 shows the sets of states of the NFA of Fig. 3.52 that we enter, starting with *e-closure* of the initial state 0, which is {0,1,3,7}, and proceeding from there. After reading the fourth input symbol, we are in an empty set of states, since in Fig. 3.52, there are no transitions out ofstate 8 on input *a.*

Thus, we need to back up, looking for a set of states that includes an accepting state. Notice that, as indicated in Fig. 3.53, after reading a we are in a set that includes state 2 and therefore indicates that the pattern a has been matched. However, after reading aab, we are in state 8, which indicates that a * b + has been matched; prefix aab is the longest prefix that gets us to an accepting state. We therefore select aab as the lexeme, and execute action A3, which should include a return to the parser indicating that the token whose pattern is p3 = a * b + has been found. •

3. DFA's for Lexical Analyzers

Another architecture, resembling the output of Lex, is to convert the NFA for all the patterns into an equivalent DFA, using the subset construction of Algorithm 3.20. Within each DFA state, if there are one or more accepting NFA states, determine the first pattern whose accepting state is represented, and make that pattern the output of the DFA state.

Example : Figure 3.54 shows a transition diagram based on the DFA that is constructed by the subset construction from the NFA in Fig. 3.52. The accepting states are labeled by the pattern that is identified by that state. For instance, the state {6,8} has two accepting states, corresponding to patterns **abb** and **a * b** $^{+}$ **.** Since the former is listed first, that is the pattern associated with state {6,8} . •

We use the DFA in a lexical analyzer much as we did the NFA. We simulate the DFA until at some point there is no next state (or strictly speaking, the next state is 0, the *dead state* corresponding to the empty set of NFA states). At that point, we back up through the sequence of states we entered and, as soon as we meet an accepting DFA state, we perform the action associated with the pattern for that state.

Example : Suppose the DFA of Fig. 3.54 is given input *abba.* The se-quence of states entered is 0137,247,58,68, and at the final *a* there is no tran-sition out of state 68. Thus, we consider the sequence from the end, and in this case, 68 itself is an accepting state that reports pattern p_2 = **abb . •**

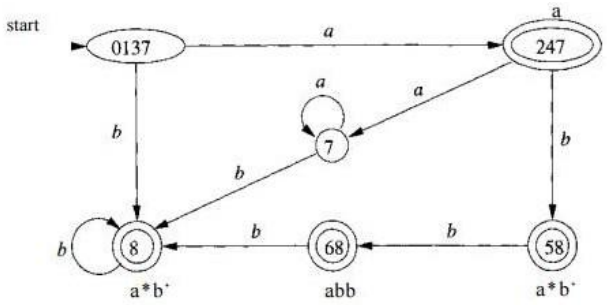


Figure 3.54: Transition graph for DFA handling the patterns `a`, `abb`, and `a*b*`

4. Implementing the Lookahead Operator

Lex lookahead operator / in a **Lex** pattern rVr_2 is sometimes necessary, because the pattern r*i for a particular token may need to describe some trailing context r_2 inorder to correctly identify the actual lexeme. When converting the pattern rVr_2 to an NFA, we treat the / as if it were e, so we do not actually look for a / on the input. However, if the NFA recognizes a prefix *xy* of the input buffer as matching this regular expression, the end of the lexeme is not where the NFA entered its accepting state. Rather the end occurs when the NFA enters a state *s* such that

1. *s* has an e-transition on the (imaginary) /,

2. There is a path from the start state of the NFA to state *s* that spells out *x.*

3. There is a path from state *s* to the accepting state that spells out *y.*

4. *x* is as long as possible for any *xy* satisfying conditions **1-3.**

If there is only one e-transition state on the imaginary / in the NFA, then the end of the lexeme occurs when this state is entered for the last time as the following example illustrates. If the NFA has more than one e-transition state on the imaginary /, then the general problem of finding the correct state *s* is much more difficult.

Dead States in DFA's

Technically, the automaton in Fig. 3.54 is not quite a DFA. The reason is that a DFA has a transition from every state on every input symbol in its input alphabet. Here, we have omitted transitions to the dead state 0, and we have therefore omitted the transitions from the dead state to itself on every input. Previous NFA-to-DFA examples did not have a way to get from the start state to 0, but the NFA of Fig. 3.52 does.

However, when we construct a DFA for use in a lexical analyzer, it is important that we treat the dead state differently, since we must know when there is no longer any possibility of recognizing a longer lexeme. Thus, we suggest always omitting transitions to the dead state and elimi-nating the dead state itself. In fact, the problem is harder than it appears, since an NFA-to-DFA construction may yield several states that cannot reach any accepting state, and we must know when any of these states have been reached. Section 3.9.6 discusses how to combine all these states into one dead state, so their identification becomes easy. It is also interesting to note that if we construct a DFA from a regular expression using Algorithms 3.20 and 3.23, then the DFA will not have any states besides 0 that cannot lead to an accepting state.

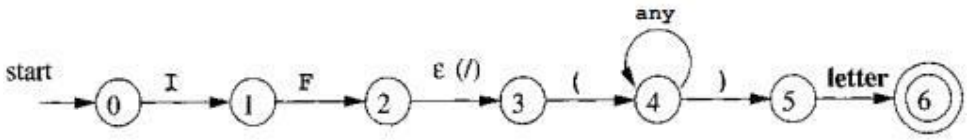


Figure 3.55: NFA recognizing the keyword IF

UNIT-3

PARSING

TOP DOWN PARSING:

 Top-down parsing can be viewed as the problem of constructing a parse tree for the given input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first left to right).

 Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

It is classified in to two different variants namely; one which uses Back Tracking and the other is Non Back Tracking in nature.

Non Back Tracking Parsing: There are two variants of this parser as given below.

1. Table Driven Predictive Parsing :

i. LL (1) Parsing

2. Recursive Descent parsing

Back Tracking

1. Brute Force method

NON BACK TRACKING:

LL (1) Parsing or Predictive Parsing

LL (1) stands for, left to right scan of input, uses a Left most derivation, and the parser takes 1 symbol as the look ahead symbol from the input in taking parsing action decision.

A non recursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation.

If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols a such that

$$S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$$

The table-driven parser in the figure has

 An input buffer that contains the string to be parsed followed by a $ Symbol, used to indicate end of input.

 A stack, containing a sequence of grammar symbols with a $ at the bottom of the stack, which initially contains the start symbol of the grammar on top of $.

 A parsing table containing the production rules to be applied. This is a two dimensional array M [Non terminal, Terminal].

 A parsing Algorithm that takes input String and determines if it is conformant to Grammar and it uses the parsing table and stack to take such decision.

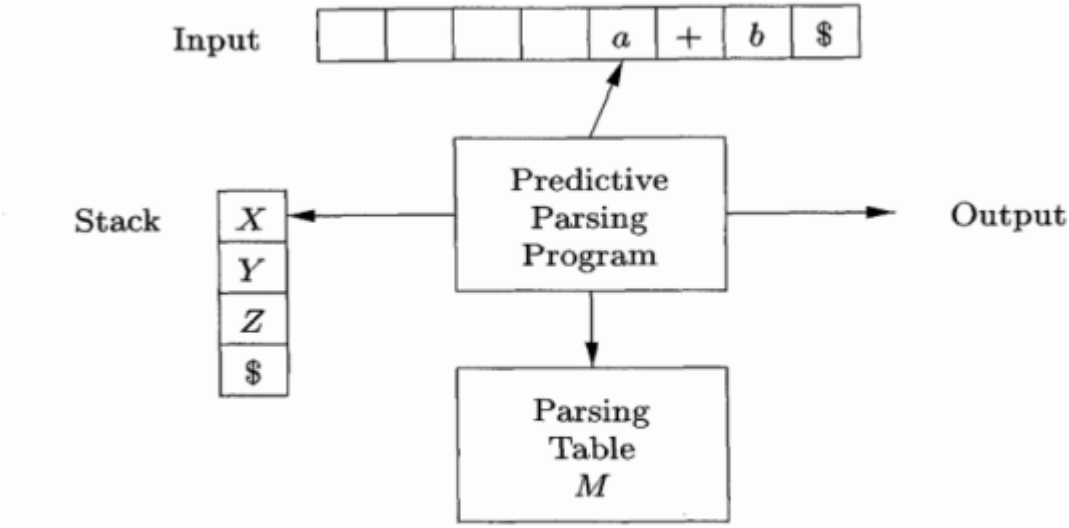


Figure 2.2: Model for table driven parsing

The Steps Involved In constructing an LL(1) Parser are:

1. Write the Context Free grammar for given input String

2. Check for Ambiguity. If ambiguous remove ambiguity from the grammar

3. Check for Left Recursion. Remove left recursion if it exists.

4. Check For Left Factoring. Perform left factoring if it contains common prefixes in

more than one alternates.

5. Compute FIRST and FOLLOW sets

6. Construct LL(1) Table

7. Using LL(1) Algorithm generate Parse tree as the Output

Context Free Grammar (CFG): CFG used to describe or denote the syntax of the programming language constructs. The CFG is denoted as G, and defined using a four tuple notation.

Let G be CFG, then G is written as, G= (V, T, P, S)

Where

 V is a finite set of Non terminal; Non terminals are syntactic variables that denote sets of strings. The sets of strings denoted by non terminals help define the language generated by the grammar. Non terminals impose a hierarchical structure on the language that is key to syntax analysis and translation.

 T is a Finite set of Terminal; Terminals are the basic symbols from which strings are formed. The term "token name" is a synonym for "'terminal" and frequently we will use the word "token" for terminal when it is clear that we are talking about just the token name. We assume that the terminals are the first components of the tokens output by the lexical analyzer.

 S is the Starting Symbol of the grammar, one non terminal is distinguished as the start symbol, and the set of strings it denotes is the language generated by the grammar. P is finite set of Productions; the productions of a grammar specify the manner in which the terminals and non terminals can be combined to form strings, each production is in α->β form, where α is a single non terminal, β is (VUT)*.Each production consists of:

(a) A non terminal called the head or left side of the production; this production defines some of the strings denoted by the head.

(b) The symbol ->. Some times: = has been used in place of the arrow.

(c) A body or right side consisting of zero or more terminals and non-terminals. The components of the body describe one way in which strings of the non terminal at the head can be constructed.

 Conventionally, the productions for the start symbol are listed first.

Example: Context Free Grammar to accept Arithmetic expressions.

The terminals are **+, *, -, (,), id**.

The **Non terminal symbols** are **expression**, **term**, **factor** and expression is the starting symbol.

$$
\begin{aligned}
\textit{expression} &\rightarrow \textit{expression} + \textit{term} \\
\textit{expression} &\rightarrow \textit{expression} - \textit{term} \\
\textit{expression} &\rightarrow \textit{term} \\
\textit{term} &\rightarrow \textit{term} * \textit{factor} \\
\textit{term} &\rightarrow \textit{term} / \textit{factor} \\
\textit{term} &\rightarrow \textit{factor} \\
\textit{factor} &\rightarrow (\textit{expression}) \\
\textit{factor} &\rightarrow \textit{id}
\end{aligned}
$$

Figure 2.3 : Grammar for Simple Arithmetic Expressions

Notational Conventions Used In Writing CFGs:

To avoid always having to state that —these are the terminals," "these are the non terminals," and so on, the following notational conventions for grammars will be used throughout our discussions.

1. These symbols are terminals:

(a) Lowercase letters early in the alphabet, such as a, b, e.

(b) Operator symbols such as +, *, and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0, 1. . . 9.

(e) Boldface strings such as id or if, each of which represents a single terminal symbol.

2. These symbols are non terminals:

(a) Uppercase letters early in the alphabet, such as A, B, C.

(b) The letter S, which, when it appears, is usually the start symbol.

(c) Lowercase, italic names such as expr or stmt.

(d) When discussing programming constructs, uppercase letters may be used to represent

Non terminals for the constructs. For example, non terminal for expressions, terms,

and factors are often represented by E, T, and F, respectively.

Using these conventions the grammar for the arithmetic expressions can be written as

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid id$$

DERIVATIONS:

The construction of a parse tree can be made precise by taking a derivational view, in

which productions are treated as rewriting rules. Beginning with the start symbol, each rewriting

step replaces a Non terminal by the body of one of its productions. This derivational view

corresponds to the top-down construction of a parse tree as well as the bottom construction of the

parse tree.

 Derivations are classified in to Let most Derivation and Right Most Derivations.

Left Most Derivation (LMD):

It is the process of constructing the parse tree or accepting the given input string, in

which at every time we need to rewrite the production rule it is done with left most non terminal

only.

Ex: - If the Grammar is E-> E+E | E*E | -E| (E) | id and the input string is id + id* id

The production E -> - E signifies that if E denotes an expression, then – E must also denote an

expression. The replacement of a single E by - E will be described by writing

E => -E which is read as “E derives _E”

For a general definition of derivation, consider a non terminal A in the middle of a

sequence of grammar symbols, as in αAβ, where α and β are arbitrary strings of grammar

symbol. Suppose A ->γ is a production. Then, we write αAβ => αγβ. The symbol => means

"derives in one step". Often, we wish to say, "Derives in zero or more steps." For this purpose,

we can use the symbol , If we wish to say, "Derives in one or more steps." We cn use

the symbol . If S a, where S is the start symbol of a grammar G, we say that α is a

sentential form of G.

The Leftmost Derivation for the given input string id + id* id is

E => E +E

=> id + E

=> id + E * E

=> id + id * E

=> id + id * id

NOTE: Every time we need to start from the root production only, the under line using at Non terminal indicating that, it is the non terminal (left most one) we are choosing to rewrite the productions to accept the string.

Right Most Derivation (RMD):

It is the process of constructing the parse tree or accepting the given input string, every time we need to rewrite the production rule with Right most Non terminal only.

The Right most derivation for the given input string **id + id* id** is

E => E + E
=> E + E * E
=> E + E * id
=> E + id * id
=> id + id * id

What is a Parse Tree?

A parse tree is a graphical representation of a derivation that filters out the order in which

productions are applied to replace non terminals.

- Each interior node of a parse tree represents the application of a production.
- All the interior nodes are Non terminals and all the leaf nodes terminals.

 All the leaf nodes reading from the left to right will be the output of the parse tree.

 If a node n is labeled X and has children n1,n2,n3,…nk with labels X1,X2,…Xk respectively, then there must be a production A->X1X2…Xk in the grammar.

Example1:- Parse tree for the input string - (id + id) using the above Context free Grammar is

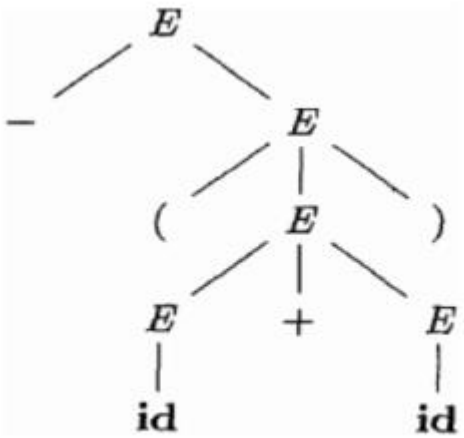


Figure 2.4 : Parse Tree for the input string - **(id + id)**

The Following figure shows step by step construction of parse tree using CFG for the parse tree for the input string - (id + id).

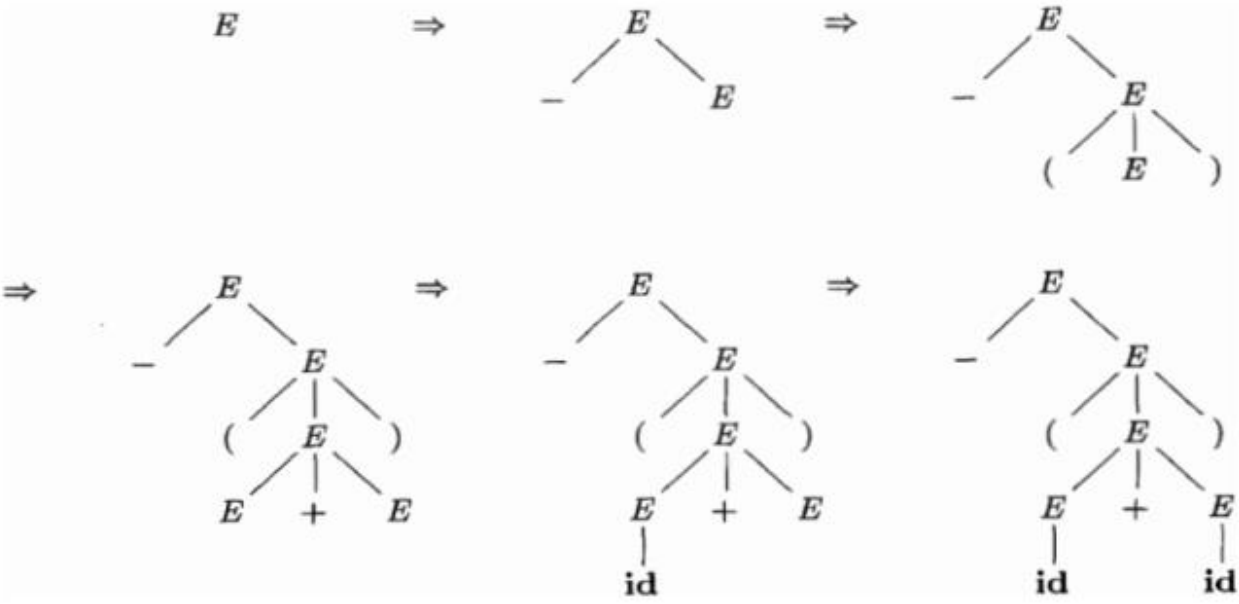


Figure 2.5 : Sequence outputs of the Parse Tree construction process for the input string **–(id+id)**

Example2:- Parse tree for the input string **id+id*id** using the above Context free Grammar is

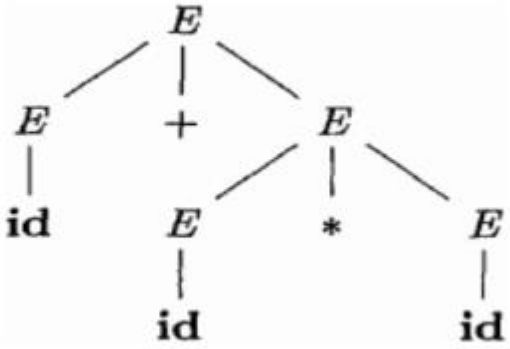


Figure 2.6: Parse tree for the input string id+ id*id

AMBIGUITY in CFGs:

Definition: A grammar that produces more than one parse tree for some sentence (input string)

is said to be ambiguous.

In other words, an ambiguous grammar is one that produces more than one leftmost

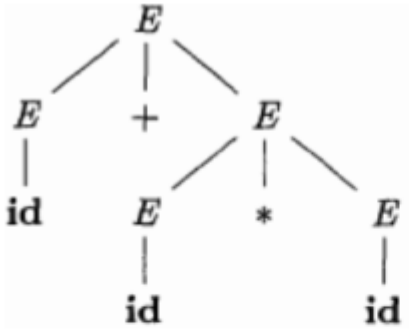
derivation or more than one rightmost derivation for the same sentence.

Or If the right hand production of the grammar is having two non terminals which are

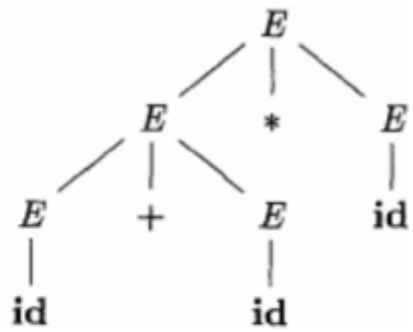
exactly same as left hand side production Non terminal then it is said to an ambiguous grammar.

Example : If the Grammar is E-> E+E | E*E | -E| (E) | id and the Input String is id + id* id

Two parse trees for given input string are



(a)



(b)

Two Left most Derivations for given input String are :

E => E +E
=> id + E
=> id + E * E
=> id + id * E
=> id + id * id

(a)

E => E * E
=> E+E*E
=> id + E * E
=> id+ id* E
=> id + id * id

(b)

The above Grammar is giving two parse trees or two derivations for the given input string so, it is an ambiguous Grammar

Note: LL (1) parser will not accept the ambiguous grammars or We cannot construct an LL(1) parser for the ambiguous grammars. Because such grammars may cause the Top Down parser to go into infinite loop or make it consume more time for parsing. If necessary we must remove all types of ambiguity from it and then construct.

ELIMINATING AMBIGUITY: Since Ambiguous grammars may cause the top down Parser go into infinite loop, consume more time during parsing.

Therefore, sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. The general form of ambiguous productions that cause ambiguity in grammars is

$$A \rightarrow A\alpha \mid \beta$$

This can be written as (introduce one new non terminal in the place of second non terminal)

$$A \rightarrow \beta\, A'$$
$$A' \rightarrow \alpha\, A' \mid \varepsilon$$

Example : Let the grammar is **E⟶ E+E | E*E | -E| (E) | id .** It is shown that it is ambiguous that can be written as

E → E+E
E → E-E
E → E*E
E → -E
E → (E)
E → id

In the above grammar the 1st and 2nd productions are having ambiguity. So, they can be written as

E-> E+E | E*E this production again can be written as
E-> E+E | β , where β is E*E
The above production is same as the general form. so, that can be written as
E->E+T|T
T->β

The value of β is E*E so, above grammar can be written as

1) **E->E+T|T**
2) **T-> E*E** **The first production is free from ambiguity** and substitute E->T in the 2nd production then it can be written as
   T-> T*T | -E| (E) | **id** this production again can be written as
   T->T*T | β where β is -E| (E) | id, introduce new non terminal in the Right hand side production then it becomes
   T->T*F | F
   F-> -E | (E) | id now the entire grammar turned in to it equivalent unambiguous,

The Unambiguous grammar equivalent to the given ambiguous one is

1) **E → E + T | T**
2) **T → T * F | F**
3) **F → -E | (E) | id**

LEFT RECURSION:

Another feature of the CFGs which is not desirable to be used in top down parsers is left

recursion. A grammar is left recursive if it has a non terminal A such that there is a derivation

A=>Aα for some string α in (TUV)*. LL(1) or Top Down Parsers can not handle the Left Recursive grammars, so we need to remove the left recursion from the grammars before being used in Top Down Parsing.

The General form of Left Recursion is:

$$A \rightarrow A\alpha \mid \beta$$

The above left recursive production can be written as the non left recursive equivalent :

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Example : - Is the following grammar left recursive? If so, find a non left recursive grammar equivalent to it.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow -E \mid (E) \mid id$$

Yes ,the grammar is left recursive due to the first two productions which are satisfying the general form of Left recursion, so they can be rewritten after removing left recursion from $E \rightarrow E + T$, and $T \rightarrow T * F$ is

$$E \rightarrow TE'$$
$$E' \rightarrow +T\ E' \mid \epsilon$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *F\ T' \mid \epsilon$$
$$F \rightarrow (E) \mid id$$

LEFT FACTORING:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing. A grammar in which more than one production has common prefix is to be rewritten by factoring out the prefixes.

For example, in the following grammar there are n A productions have the common prefix α, which should be removed or factored out without changing the language defined for A.

$$A \rightarrow \alpha A1 \mid \alpha A2 \mid \alpha A3 \mid \alpha A4 \mid \ldots \mid \alpha An$$

We can factor out the α from all n productions by adding a new A production $A \rightarrow \alpha A'$ and rewriting the A′ productions grammar as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow A1 \mid A2 \mid A3 \mid A4 \ldots \mid An$$

FIRST and FOLLOW:

The construction of both top-down and bottom-up parsers is aided by two functions,

FIRST and FOLLOW, associated with a grammar G. During top down parsing, FIRST and

FOLLOW allow us to choose which production to apply, based on the next input (look a head) symbol.

Computation of FIRST:

FIRST function computes the set of terminal symbols with which the right hand side of the productions begin. To compute FIRST (A) **for all grammar symbols**, apply the following rules until no more terminals or € can be added to any FIRST set.

1. If A is a terminal, then FIRST {A} = {A}.
2. If A is a Non terminal and A->X1X2…Xi
   FIRST(A)=FIRST(X1) if X1is not null, if X1 is a non terminal and X1->€, add FIRST(X2) to FIRST(A), if X2-> € add FIRST(X3) to FIRST(A), … if Xi->€ , i.e., all Xi's for i=1..i are null, add € FIRST(A).
3. If A ->€ is a production, then add € to FIRST (A).

Computation Of FOLLOW:

Follow (A) is nothing but the set of terminal symbols of the grammar that are immediately following the Non terminal A. If **a** is to the immediate right of non terminal A, then Follow(A)= {**a**}. To compute FOLLOW (A) for **all non terminals** A, apply the following rules until no more symbols can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where *S* is the start symbol, and $ is the input right end marker.
2. If there is a production A-> αBβ, then everything in FIRST (β) except Є is in FOLLOW(B).
3. If there is a production A->αB or a production A-> αBβ with FIRST(β) contains Є, then FOLLOW (B) = FOLLOW (A).

Example: - Compute the FIRST and FOLLOW values of the expression grammar

1. **E → TE′**
2. **E′→ +TE′ | Є**
3. **T → FT′**
4. **T′ → *FT′ | Є**
5. **F → (E) | id**

Computing FIRST Values:

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}

FIRST (E′) = {+, Є}

FIRST (T′) = {*, Є}

Computing FOLLOW Values:

FOLLOW (E) = { $,), } Because it is the start symbol of the grammar.

FOLLOW (E′) = {FOLLOW (E)} satisfying the 3rd rule of FOLLOW()

= { $,) }

FOLLOW (T) = { FIRST E′} It is Satisfying the 2nd rule.

U { FOLLOW(E′) }

= {+, FOLLOW (E′)}

= { +, $,) }

FOLLOW (T′) = { FOLLOW(T)} Satisfying the 3rd Rule

= { +, $,) }

FOLLOW (F) = { FIRST (T′) } It is Satisfying the 2nd rule.

U { FOLLOW(E′) }

= {*, FOLLOW (T)}

= { *, +, $,)}

NON TERMINAL	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E′	{ +, € }	{ $,) }
T	{ (, id}	{ +, $,) }
T′	{*, €}	{ +, $,) }
F	{ (, id}	{ *, +, $,) }

Table 2.1: FIRST and FOLLOW values

Constructing Predictive Or LL (1) Parse Table:

It is the process of placing the all productions of the grammar in the parse table based on the FIRST and FOLLOW values of the Productions.

The rules to be followed to Construct the Parsing Table (M) are :

1. For Each production A-> α of the grammar, do the bellow steps.
2. For each terminal symbol 'a' in FIRST (α), add the production A-> α to M [A, a].
3. i. If € is in FIRST (α) add production A->α to M [A, b], where b is all terminals in FOLLOW (A).
   ii. If € is in FIRST(α) and $ is in FOLLOW (A) then add production A->α to M [A, $].
4. Mark other entries in the parsing table as error .

NON-TERMINALS	INPUT SYMBOLS					
	+	*	(	)	id	$
E			E TE′		E id	
E′	E′ +TE′			E′ €		E′ €
T			T FT′		T FT′	
T′	T′ €	T′ *FT′		T′ €		T′ €
F			F (E)		F id	

Table 2.2: LL (1) Parsing Table for the Expressions Grammar

Note: if there are no multiple entries in the table for single a terminal then grammar is accepted by LL(1) Parser.

LL (1) Parsing Algorithm:

The parser acts on basis on the basis of two symbols

i. A, the symbol on the top of the stack

ii. a, the current input symbol

There are three conditions for A and _a', that are used fro the parsing program.

1. If A=a=$ then parsing is Successful.

2. If A=a≠$ then parser pops off the stack and advances the current input pointer to the

next.

3. If A is a Non terminal the parser consults the entry M [A, a] in the parsing table. If

M[A, a] is a Production A-> X1X2..Xn, then the program replaces the A on the top of

the Stack by X1X2..Xn in such a way that X1 comes on the top.

STRING ACCEPTANCE BY PARSER:

If the input string for the parser is id + id * id, the below table shows how the parser accept the string with the help of Stack.

Stack	Input	Action	Comments
$E	id+ id* id $	E TE`	E on top of the stack is replaced by TE`
$E`T	id+ id*id $	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id+ id*id $	F id	F on top of the stack is replaced by id
$E`T`id	id+ id*id $	pop and remove id	Condition 2 is satisfied
$E`T`	+id*id$	T` €	T` on top of the stack is replaced by €
$E`	+id*id$	E` +TE`	E` on top of the stack is replaced by +TE`
$E`T+	+id*id$	Pop and remove +	Condition 2 is satisfied
$E`T	id*id$	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id*id$	F id	F on top of the stack is replaced by id
$E`T`id	id * id$	pop and remove id	Condition 2 is satisfied

$E`T`	*id$	T` *FT`	T` on top of the stack is replaced by *FT`
$E`T`F*	*id$	pop and remove *	Condition 2 is satisfied
$E`T`F	id$	F id	F on top of the stack is replaced by id
$E`T`id	id$	Pop and remove id	Condition 2 is satisfied
$E`T`	$	T` €	T` on top of the stack is replaced by €
$E`	$	E` €	E` on top of the stack is replaced by €
$	$	Parsing is successful	Condition 1 satisfied

Table2.3 : Sequence of steps taken by parser in parsing the input **token stream id+ id* id**

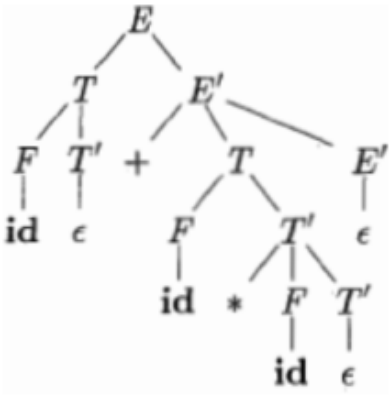


Figure 2.7: Parse tree for the input id + id* id

RECURSIVE DESCENT PARSING :

A recursive-descent parsing program consists of a set of recursive procedures, one for each non terminal. Each procedure is responsible for parsing the constructs defined by its non terminal, Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

If the given grammar is

E → TE′
E′ → +TE′ | €

T → FT′
T′ → *FT′ | €

F → (E) | id

Reccursive procedures for the recursive descent parser for the given grammar are given below.

```
procedure E( )
{
        T( );
         E′( );
}
```

```
procedure T ( )
{
        F(  );
        T′( );
}
Procedure E′( )
{
        if input = _+‘
        {
                advance(
                );T ( );
                E′( );
                return true;
        }
        else error;
}
procedure T′( )
{
        if input = _*‘
        {
                advance(
                );F ( );
```

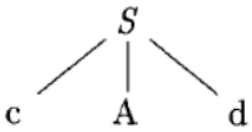
```
            T′( );
        return true;
        }
        else return error;
}
procedure F( )
{
        if input = _(_
        {
            advance(
            );E ( );
            if input = _)‘
            advance(
            ); return
            true;
        }
        else if input =  –idll
        {

            advance(
            );return
            true;
        }
        else return error;
}
advance()
{
        input = next token;
}
```

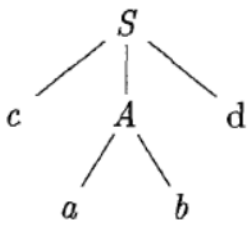
BACK TRACKING: This parsing method uses the technique called Brute Force method during the parse tree construction process. This allows the process to go back (back track) and redo the steps by undoing the work done so far in the point of processing.

Brute force method: It is a Top down Parsing technique, occurs when there is morethan one alternative in the productions to be tried while parsing the input string. It selects alternatives in the order they appear and when it realizes that something gone wrong it tries withnext alternative.
For example, consider the grammar bellow.

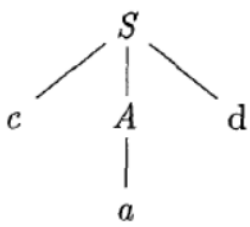
S → cAd
A → ab | a

To generate the input string –cadll, initially the first parse tree given below is generated. As the string generated is not –cadll, input pointer is back tracked to position –All, to examine the next alternate of –All. Now a match to the input string occurs as shown in the 2nd parse trees given below.





(1)



(2)

BOTTOM-UP PARSING

Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom nodes) and working up towards the root (the top node). It involves –reducing an input string ‗w' to the Start Symbol of the grammar. in each reduction step, a perticular substring matching the right side of the production is replaced by symbol on the left of that production and it is the Right most derivation. For example consider the following Grammar:

E → E+T|T

T → T*F

F → (E)|id

Bottom up parsing of the input string **"id * id "**is as follows:

INPUT STRING	SUB STRING	REDUCING PRODUCTION
id*id	Id	F->id
F*id	T	F->T
T*id	Id	F->id
T*F	*	T->T*F
T	T*F	E->T
E		Start symbol. Hence, the input String is accepted

Parse Tree representation is as follows:

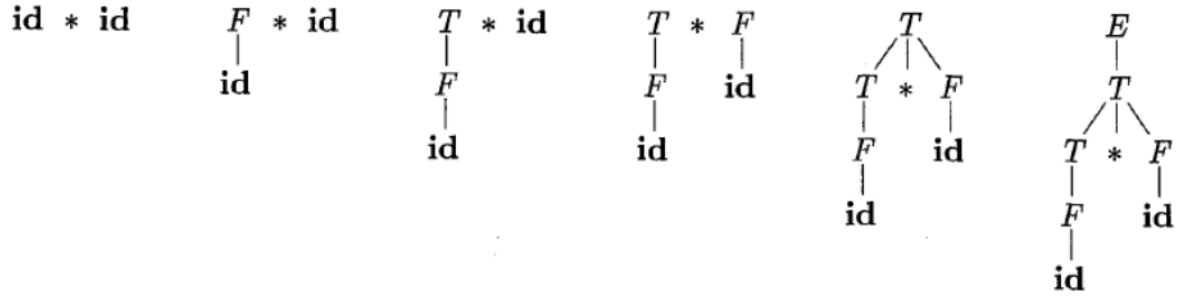


Figure 3.1 : A Bottom-up Parse tree for the input String "id*id"

Bottom up parsing is classified in to 1. Shift-Reduce Parsing, 2. Operator Precedence parsing ,and 3. [Table Driven] L R Parsing

i. SLR(1)
ii. CALR (1)
iii. LALR(1)

SHIFT-REDUCE PARSING:

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed, We use $ to mark the bottom of the stack and also the right end of the input. And it makes use of the process of shift and reduce actions to accept the input string. Here, the parse tree is Constructed bottom up from the leaf nodes towards the root node.

When we are parsing the given input string, if the match occurs the parser takes the reduce action otherwise it will go for shift action. And it can accept ambiguous grammars also.

For example, consider the below grammar to accept the input string –id * id–, using S-R parser

E → E+T|T

T → T*F | F

F → (E)|id

Actions of the Shift-reduce parser using Stack implementation

STACK	INPUT	ACTION
$	Id*id$	Shift
$id	*id$	Reduce with F→ d
$F	*id$	Reduce with T→ F
$T	*id$	Shift
$T*	id$	Shift
$T*id	$	Reduce with F id
$T*F	$	Reduce with T T*F
$T	$	Reduce with E T
$E	$	Accept

Consider the following grammar:

S → aAcBe

A → Ab|b

B → d

Let the input string is –abbcde**ll**. The series of shift and reductions to the start symbol are as follows.

abbcde ⇨ aAbcde ⇨ aAcde ⇨ aAcBe ⇨ S

Note: in the above example there are two actions possible in the second Step, these are asfollows :

1. Shift action going to 3rd Step
2. Reduce action, that is A->b

If the parser is taking the 1st action then it can successfully accepts the given input string, if it is going for second action then it can't accept given input string. This is called shift reduce conflict. Where, S-R parser is not able take proper decision, so it not recommended for parsing.

LR Parsing:

Most prevalent type of bottom up parsing is LR (k) parsing. Where, L is left to right scan of the given input string, R is Right Most derivation in reverse and K is no of input symbols as the Look ahead.

- ∑ It is the most general non back tracking shift reduce parsing method
- ∑ The class of grammars that can be parsed using the LR methods is a proper superset ofthe class of grammars that can be parsed with predictive parsers.
- ∑ An LR parser can detect a syntactic error as soon as it is possible to do so, on a left toright scan of the input.

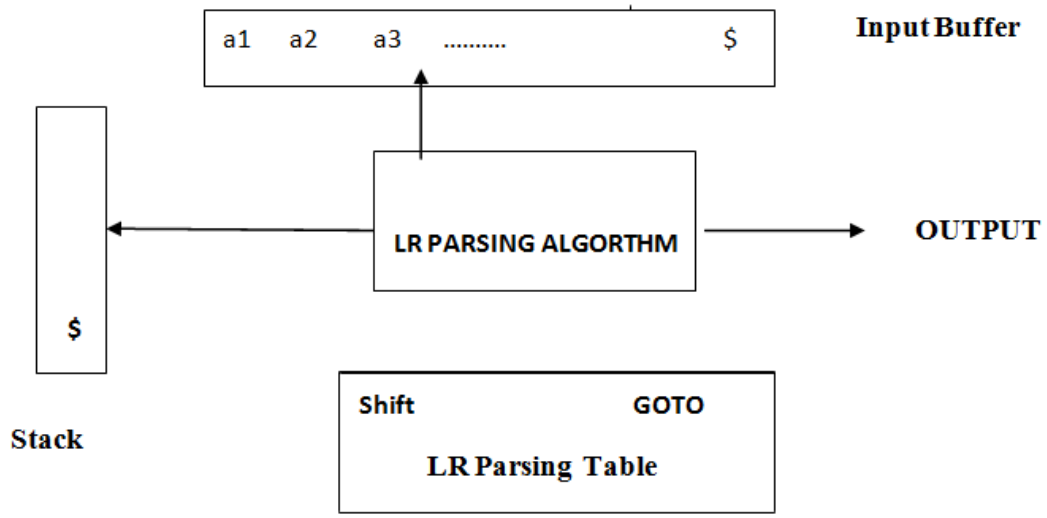


Figure 3.3: Components of LR Parsing

LR Parser Consists of

- An input buffer that contains the string to be parsed followed by a $ Symbol, used toindicate end of input.
- A stack containing a sequence of grammar symbols with a $ at the bottom of the stack,which initially contains the Initial state of the parsing table on top of $.
- A parsing table (M), it is a two dimensional array M[state, terminal or Non terminal] andit contains two parts

1. **ACTION Part**
   The ACTION part of the table is a two dimensional array indexed by state and the input symbol, i.e. **ACTION**[state][input], An action table entry can have one of following four kinds of values in it. They are:
   1. Shift X, where X is a State number.
   2. Reduce X, where X is a Production number.
   3. Accept, signifying the completion of a successful parse.
   4. Error entry.
2. **GO TO Part**
   The GO TO part of the table is a two dimensional array indexed by state and a Non terminal, i.e. GOTO[state][NonTerminal]. A GO TO entry has a state number in the table.

- A parsing Algorithm uses the current State X, the next input symbol ‗a' to consult the entry at action[X][a]. it makes one of the four following actions as given below:

1. If the action[X][a]=shift Y, the parser executes a shift of Y on to the top of the stack and advances the input pointer.
2. If the action[X][a]= reduce Y (Y is the production number reduced in the State X), if the production is Y->β, then the parser pops 2*β symbols from the stack and push Y on to the Stack.
3. If the action[X][a]= accept, then the parsing is successful and the input string is accepted.
4. If the action[X][a]= error, then the parser has discovered an error and calls the error

routine.

The parsing is classified in to

1. LR (0)
2. Simple LR (1)
3. Canonical LR (1)
4. Look ahead LR (1)

LR (1) Parsing: Various steps involved in the LR (1) Parsing:

1. Write the Context free Grammar for the given input string
2. Check for the Ambiguity
3. Add Augment production
4. Create Canonical collection of LR (0) items
5. Draw DFA
6. Construct the LR (0) Parsing table
7. Based on the information from the Table, with help of Stack and Parsing algorithm generate the output.

Augment Grammar

The Augment Grammar G`, is G with a new starting symbol S` an additional production S` S. this helps the parser to identify when to stop the parsing and announce the acceptance of the input. The input string is accepted if and only if the parser is about to reduce by S` S. For example let us consider the Grammar below:

E → E+T|T
T → T*F
F → (E) | id **the Augment grammar G` is Represented by**

E` → E
E → E+T|T
T → T*F
F → (E) | id

LR (0) items

An LR (0) item of a Grammar is a production G with dot at some position on the right side of the production. An item indicates how much of the input has been scanned up to a given point in the process of parsing. For example, if the Production is X → YZ then, The LR (0) items are:

1. X → •AB, indicates that the parser expects a string derivable from AB.
2. X → A•B, indicates that the parser has scanned the string derivable from the A and expecting the string from Y.
3. X → AB•, indicates that he parser has scanned the string derivable from AB.

If the grammar is X → € the, the LR (0) item is

X→ •, indicating that the production is reduced one.

Canonical collection of LR(0) Items:

This is the process of grouping the LR (0) items together based on the closure and Go to operations

Closure operation

If I is an initial State, then the Closure (I) is constructed as follows:

1. Initially, add Augment Production to the state and check for the • symbol in the Righthand side production, if the • is followed by a Non terminal then Add Productions which are Stating with that Non Terminal in the State I.
2. If a production X→α•Aβ is in I, then add Production which are starting with X in the State I. Rule 2 is applied until no more productions added to the State I(meaning that the • is followed by a Terminal symbol).

Example :

0. E` → E		**E` → • E**
1. E → E+T	**LR (0) items for the Grammar is**	**E → •E+T**
2. T → F		**T → •F**
3. T → T*F		**T → •T*F**
4. F → (E)		**F → • (E)**
5. F → id		**F → • id**

Closure (I_0) State

Add **E` → • E** in I_0 State

Since, the _•' symbol in the Right hand side production is followed by A Non terminal E. So, add productions starting with E in to Io state. So, the state becomes

The 1st and 2nd productions are satisfies the 2nd rule. So, add productionswhich are starting with E and T in I_0

Note: once productions are added in the state the same production shouldnot added for the 2nd time in the same state.So,

→

→

the state becomes

E` •E →

0. E → •E+T

1. T → •F

The 1st and 2nd productions are satisfies the 2nd rule. So, add productions which are starting with E and T in I_0

Note: once productions are added in the state the same production should not added for the 2nd time in the same state. So, the state becomes

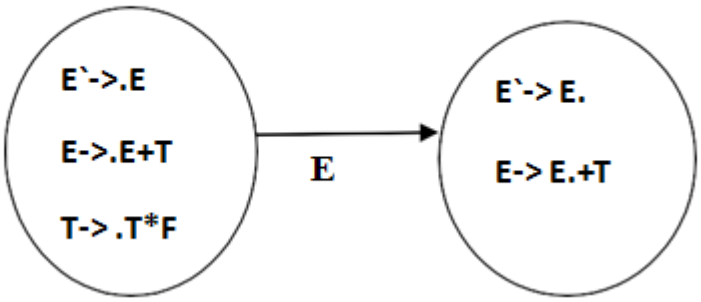
0. E` → •E
1. E → •E+T
2. T → •F
3. T → •T*F
4. F → •(E)
5. F → • id

→
→

GO TO Operation

Go to (I_0, X), where I_0 is set of items and X is the grammar Symbol on which we are moving the „•" symbol. It is like finding the next state of the NFA for a give State I_0 and theinput symbol is X. For example, if the production is **E •E+T**

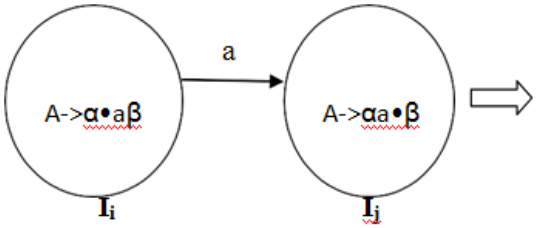
Go to (I_0, E) is E → E•+T,E` → E. = Closure ({E` → E•, E → E•+T})



Construction of LR (0) parsing Table:

Once we have Created the canonical collection of LR (0) items, need to follow the stepsmentioned below:

If there is a transaction from one state (I_i) to another state(I_j) on a terminal value then,we should write the shift entry in the action part as shown below:



States	ACTION		GO TO
	a	$	A
I_i	S_j		
I_j			

If there is a transaction from one state (I_i) to another state (I_j) on a Non terminal val ue then, we should write the subscript value of I_i in the GO TO part as shown below: part as shown below:

For Example, Construct the LR (0) parsing Table for the given Grammar (G)

S → aB
B → bB | b

Sol: 1. Add Augment Production and insert „•" symbol at the first position for every production in G

0. S′ → •S
1. S → •aB
2. B → •bB
3. B →•b

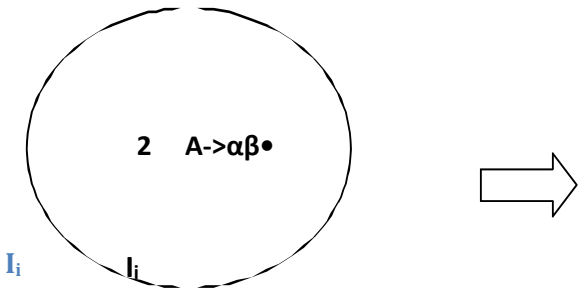
1 If there is one state (I_i), where there is one production **(A->αβ•)** which has no transitions to the next State. Then, the production is said to be a reduced production. For allterminals X in FOLLOW (A), write the reduce entry along with their production numbers. If the Augment production is reducing then write accept.

1 S -> •aAb

2 A->αβ•

Follow(S) = {$}

Follow (A) = (b}



States	ACTION			GO TO	
	a	b	$	S	A
I_i		r_2			

SLR (1) table for the Grammar

$S \rightarrow aB$

$B \rightarrow bB \mid b$

Follow (S) = {$}, Follow (B) = {$}

States	ACTION			GOTO	
	A	b	$	S	B
I_0	S_2			1	
I_1			ACCEPT		
I_2		S_4			3
I_3			R_1		
I_4		S_4	R_3		5
I_5			R_2		

Note: When Multiple Entries occurs in the SLR table. Then, the grammar is not accepted bySLR(1) Parser.

Canonical LR (1) Parsing: Various steps involved in the CLR (1) Parsing:

1. Write the Context free Grammar for the given input string
2. Check for the Ambiguity
3. Add Augment productio
4. Create Canonical collection of LR (1) items
5. Draw DFA
6. Construct the CLR (1) Parsing table
7. Based on the information from the Table, with help of Stack and Parsingalgorithm generate the output.

LR (1) items :

The LR (1) item is defined by **production**, **position of data** and a **terminal symbol**. The terminal is called as ***Look ahead symbol***.

General form of LR (1) item is

S->α•Aβ , $

Rules to create canonical collection:

1. Every element of I is added to closure of I
2. If an LR (1) item [**X-> A•BC, a]** exists in I, and there exists a production B->$b_1b_{2.....}$, then add item **[B->• b_1b_2, z]** where z is a terminal in **FIRST(Ca),** if it is not already in Closure(I).keep applying this rule until there are no more elements adde.

For example, if the grammar is

S->CC C->cC C->d

The Canonical collection of LR (1) items can be created as follows:

0. S'->•S (Augment Production)
1. S->•CC
2. C->•cC
3. C->•d

I_0 State : Add Augment production and compute the Closure, the look ahead symbol for the Augment Production is $.

S'->•S, $= Closure(S'->•S, $)

The dot symbol is followed by a Non terminal S. So, add productions starting with S in I_0 State.

S->•CC, FIRST ($), using 2nd rule

S->•CC, $

The dot symbol is followed by a Non terminal C. So, add productions starting with C in I_0 State.

C->•cC, FIRST(C, $)
C->•d, FIRST(C, $)

FIRST(C) = {c, d} so, the items are

C->•cC, c/d
C->•d, c/d

The dot symbol is followed by a terminal value. So, close the I_0 State. So, the productions in the

I_0 are

S′->•S , $
S->•CC , $
C->•cC, c/d
C->•d , c/d

I_1 = Goto (I_0, S)= S′->S•,$

I_2 = Go to (I_0 , C)= Closure(S-> C•C, $)

S-> **C->•cC , $**

C->•d,$ So, the I_2 State is

S->C•C,$

C->•cC , $
C->•d,$

I_3= Goto(I_0,c)= Closure(C->c•C, c/d)

C->•cC, c/d
C->•d , c/d So, the I_3 State is

C->c•C, c/d
C->•cC, c/d
C->•d , c/d

I_4= Goto(I_0, d)= Colsure(C->d•, c/d) = C->d•, c/dI_5

= Goto (I_2, C)= closure(S->CC•,$)= S->CC•, $

I_6= Goto (I_2, c)= closure(C->c•C , $)=

C->•cC, $
C->•d , $ S0, the I_6 State is

C->c•C , $
C->•cC , $
C->•d,$

I_7 = Go to (I_2, d)= Closure(C->d•,$) = C->d•, $

Goto(I_3, c)= closure(C->•cC, c/d)= I_3.

I_8= Go to (I_3, C)= Closure(C->cC•, c/d) = C->cC•, c/d

Go to (I_3, c)= Closure(C->c•C, c/d) = I_3

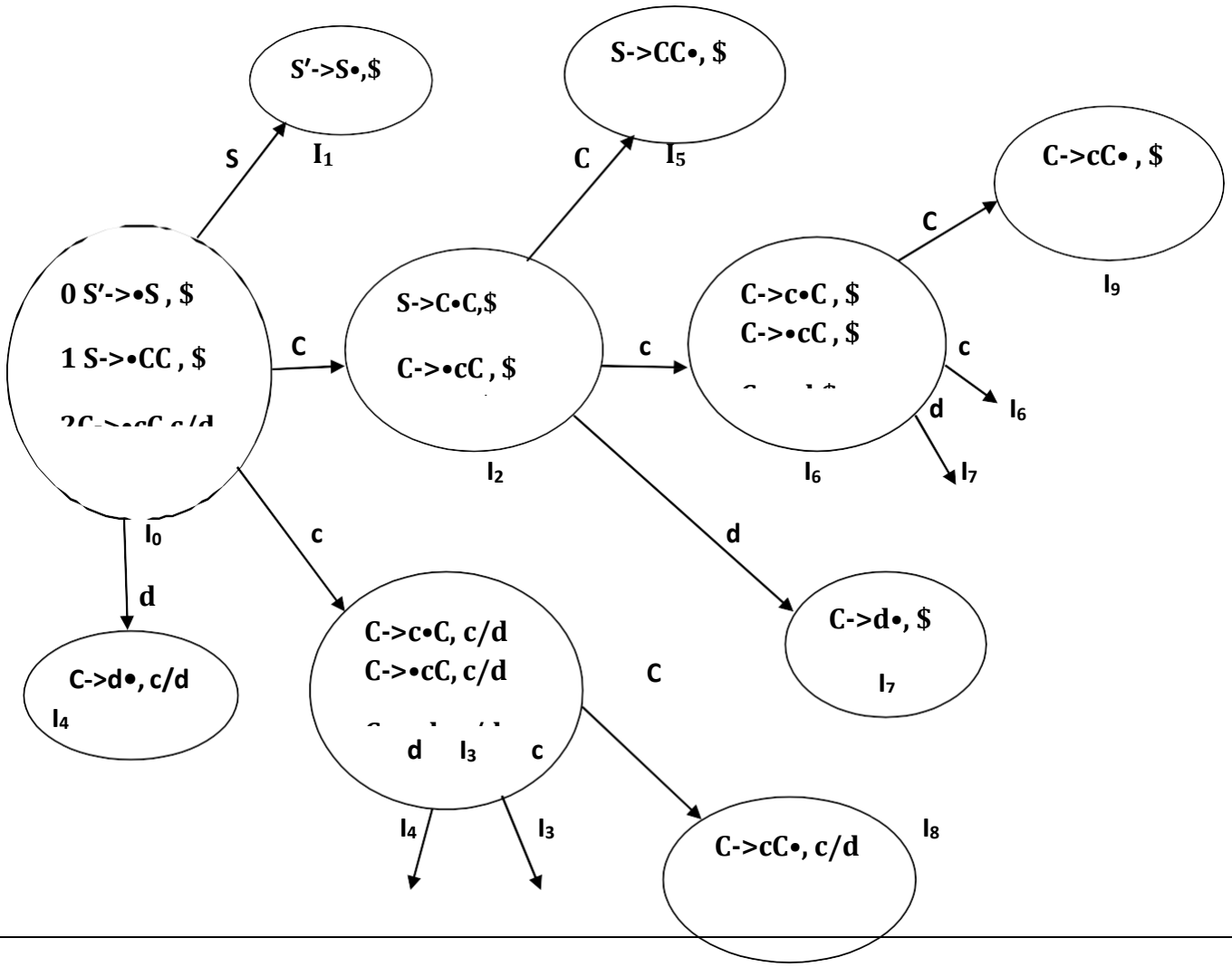
Go to (I_3, d)= Closure(C->d•, c/d) = I_4

I_9= Go to (I_6, C)= Closure(C->cC• , $) = C->cC• , $

Go to (I_6, c)= Closure(C->c•C , $) = I_6

Go to (I_6, d)= Closure(C->d•,$) = I_7

Drawing the Finite State Machine DFA for the above LR (1) items



Construction of CLR (1) Table

Rule1: if there is an item [A->α•Xβ,b] in I_i and goto(I_i,X) is in I_j then action [I_i][X]= Shiftj, Where X is Terminal.
Rule2: if there is an item [A->α•, b] in I_i and (A≠S`) set action [I_i][b]= reduce along withthe production number.
Rule3: if there is an item [S`->S•, $] in I_i then set action [I_i][$]= Accept.
Rule4: if there is an item [A->α•Xβ,b] in I_i and go to(I_i,X) is in I_j then goto [I_i][X]= j,Where X is Non Terminal.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Table : LR (1) Table

LALR (1) Parsing

The CLR Parser avoids the conflicts in the parse table. But it produces more number of States when compared to SLR parser. Hence more space is occupied by the table in the memory. So LALR parsing can be used. Here, the tables obtained are smaller than CLR parse table. But it also as efficient as CLR parser. Here LR (1) items that have same productions but different look-aheads are combined to form a single set of it

ems.

For example, consider the grammar in the previous example. Consider the states I_4 and I_7as given below:

I_4= Goto(I_0, d)= Colsure(C->d•, c/d) = C->d•, c/dI_7

= Go to (I_2 , d)= Closure(C->d•,$) = C->d•, $

These states are differing only in the look-aheads. They have the same productions. Hence thesestates are combined to form a single state called as I_{47}.

Similarly the states I_3 and I_6 differing only in their look-aheads as given below:

$I_{3=}$ Goto(I_0,c)=

C->c•C, c/d
C->•cC, c/d
C->•d , c/d

I_6= Goto (I_2, c)=

C->c•C , $
C->•cC , $
C->•d,$

These states are differing only in the look-aheads. They have the same productions. Hence thesestates are combined to form a single state called as I_{36}.

Similarly the States I_8 and I_9 differing only in look-aheads. Hence they combined to formthe state I_{89}.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			ACCEPT		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		5
I_5			R_1		
I_{89}	R_2	R_2	R_2		

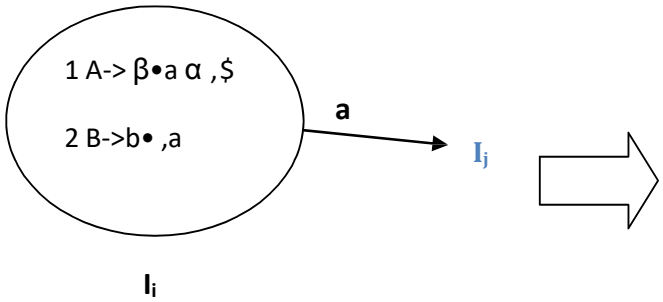
Table: LALR Table

Conflicts in the CLR (1) Parsing : When multiple entries occur in the table. Then, the situation is said to be a Conflict.

Shift-Reduce Conflict in CLR (1) Parsing

Shift Reduce Conflict in the CLR (1) parsing occurs when a state has

3. A Reduced item of the form A → α•, a and
4. An incomplete item of the form A → β•aα as shown below:

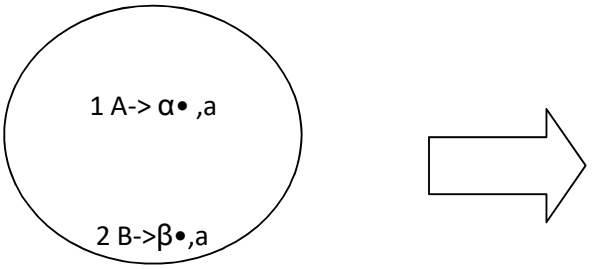


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce / Reduce Conflict in CLR (1) Parsing

Reduce- Reduce Conflict in the CLR (1) parsing occurs when a state has two or morereduced items of the form

3. **A → α•**
4. **B → β• If two productions in a state (I) reducing on same look ahead symbol as shown below:**



States	Action		GOTO	
	a	$	A	B
I_i	$r_1/r2$			

String Acceptance using LR Parsing:

Consider the above example, if the input String is **cdd**

States	ACTION			GOTO	
	c	D	$	S	C
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

0 S'->•S (Augment Production)

1 S->•CC

2 C->•cC

3 C->•d

STACK	INPUT	ACTION
$0	cdd$	Shift S_3
$0c3	dd$	Shift S_4
$0c3d4	d$	Reduce with R3,C->d, pop 2*β symbols from the stack
$0c3C	d$	Goto (I_3, C)=8Shift S_6
$0c3C8	d$	Reduce with R_2 ,C->cC, pop 2*β symbols from the stack
$0C	d$	Goto (I_0, C)=2
$0C2	d$	Shift S_7

$0C2d7	$	Reduce with R3,C->d, pop 2*β symbols from the stack
$0C2C	$	Goto (I_2, C)=5
$0C2C5	$	Reduce with R_1,S->CC, pop 2*β symbols from the stack
$0S	$	Goto (I_0, S)=1
$0S1	$	Accept

UNIT-IV

Syntax Directed Translation and Intermediate code Generator

In Intermediate code generation we use syntax directed methods to translate the source program into an intermediate form programming language constructs such as declarations, assignments and flow-of-control statements.

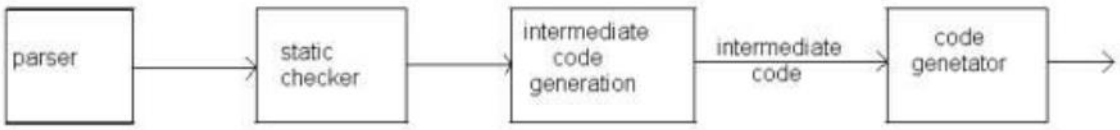


Figure 4.1 : Intermediate Code Generator

Intermediate code is:

- ∑ The output of the Parser and the input to the Code Generator.
- ∑ Relatively machine-independent and allows the compiler to be retargeted.
- ∑ Relatively easy to manipulate (optimize).

What are the Advantages of an intermediate language?

Advantages of Using an Intermediate Language includes :

1. Retargeting is facilitated - Build a compiler for a new machine by attaching a new code generator to an existing front-end.

2. Optimization - reuse intermediate code optimizers in compilers for different languagesand different machines.

Note: the terms –intermediate code**ll**, –intermediate language**ll**, and –intermediate representation**ll** are all used interchangeably.

Types of Intermediate representations / forms: There are three types of intermediaterepresentation:-

1. Syntax Trees
2. Postfix notation
3. Three Address Code

Semantic rules for generating three-address code from common programming languageconstructs are similar to those for constructing syntax trees of for generating postfix notation.

Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified. A syntax tree for the assignment statement a:=b*-c+b*-cappear in the following figure.

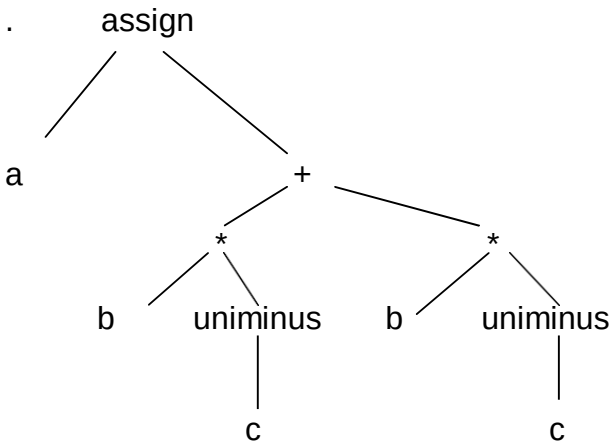


Figure 4.2 : Abstract Syntax Tree for the statement a:=b*-c+b*-c

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the in which a node appears immediately after its children. The postfix notation for the syntax tree in the fig is

a b c **uminus** + b c **uminus** * + **assign**

The edges in a syntax tree do not appear explicitly in postfix notation. They can be recovered in the order in which the nodes appear and the no. of operands that the operator at a node expects. The recovery of edges is similar to the evaluation, using a staff, of an expression inpostfix notation.

What is Three Address Code?

Three-address code is a sequence of statements of the general form : X := Y Op Z

where x, y, and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement. Thus a source language expression like x+y*z might be translated into a sequence

t1 := y * z
t2 : = x +
t1

Where t1 and t2 are compiler-generated temporary names. This unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization. The use of names for the intermediate values computed by a program allow- three-address code to be easily rearranged – unlike postfix notation. Three - address code is a linearzed representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

Intermediate code using Syntax for the above arithmetic

expressiont1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 +
t4a := t5

The reason for the term‖three-address code‖ is that each statement usually contains three addresses, two for the operands and one for the result. In the implementations of three-address code given later in this section, a programmer-defined name is replaced by a pointer tc a symbol-table entry for that name.

Types of Three-Address Statements

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three- address statement in the array holding inter- mediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using ‖back patching,‖ discussed in Section
8.6. Here are the common three-address statements used in the remainder of this book:

1. **Assignment statements** of the form x: = y op z, where op is a binary arithmetic or logical operation.

2. **Assignment instructions** of the form x:= op y, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3. **Copy statements** of the form x: = y where the value of y is assigned to x.

4. **The unconditional jump** goto L. The three-address statement with label L is the next to be executed.

5. **Conditional jumps** such as if x relop y goto L. This instruction applies a relational operator (<, =, >=, etc.) to x and y, and executes the statement with label L next if x stands in relation relop to y. If not, the three-address statement following if x relop y goto L is executed next, as in the usual sequence.

6. **param x and call p, n** for procedure calls and return y, where y representing a returned value is optional. Their typical use is as the sequence of three-address statements

param

x1

param

x2

param

xncall p,

n

Generated as part of a call of the procedure p(x,, x~,..., xǁ). The integer n indicating the number of actual parameters in ǁcall p, nǁ is not redundant because calls can be nested. The implementation of procedure calls is outline d in Section 8.7.

7. **Indexed assignments** of the form x: = y[i] and x [i]: = y. The first of these sets x to the value in the location i memory units beyond location y. The statement x[i]:=y sets the contents ofthe location i units beyond x to the value of y. In both these instructions, x, y, and i refer to data objects.

8. **Address and pointer assignments** of the form x:= &y, x:= *y and *x: = y. The first of these sets the value of x to be the location of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an I-value such as A[i, j], and x is a pointer name or temporary. That is, the r-value of x is the l-value (location) of some object!. In the statement x: = ~y, presumablyy is a pointer or a temporary whose r- value is a location. The r-value of x is made equal to the contents of that location. Finally, +x: = y sets the r-value of the object pointed to by x to the r- value of y.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must clearly be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restricted instruction set may force the front end to generate long sequences of statements for some source, language operations. The optimizer and code generator may then have to work harder if good code is to be generated.

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of non-terminal E on the left side of E □ E1 + E will be

computed into a new temporary t. In general, the three- address code for id: = E consists of code to evaluate E into some temporary t, followed by the assignment id.place: = t. If an expression is a single identifier, say y, then y itself holds the value of the expression. For the moment, we create a new name every time a temporary is needed; techniques for reusing temporaries are given in Section S.3. The S-attributed definition in Fig. 8.6 generates three-address code for assignment statements. Given input a: = b+ – c + b+ – c, it produces the code in Fig. 8.5(a). The synthesized attribute S.code represents the three- address code for the assignment S. The non- terminal E has two attributes:

1. E.place, the name that will hold the value of E, and

2. E.code, the sequence of three-address statements evaluating E.

The function newtemp returns a sequence of distinct names t1, t2,... in response to successive calls. For convenience, we use the notation gen(x ': =' y '+' z) in Fig. 8.6 to represent the three-address statement x: = y + z. Expressions appearing instead of variables like x, y, and z are evaluated when passed to gen, and quoted operators or operands, like '+', are taken literally. In practice, three- address statements might be sent to an output file, rather than built up into the code attributes. Flow-of-control statements can be added to the language of assignments in Fig.
8.6 by productions and semantic rules)like the ones for while statements in Fig. 8.7. In the figure, the code for S - while E do S, is generated using' new attributes S.begin and S.after to mark the first statement in the code for E and the statement following the code for S, respectively.

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **id** := E	*S.code* := *E.code* ‖ *gen*(**id**.*place* ':=' *E.place*)
$E \rightarrow E_1 + E_2$	*E.place* := *newtemp*; *E.code* := E_1.*code* ‖ E_2.*code* ‖ *gen*(*E.place* ':=' E_1.*place* '+' E_2.*place*)
$E \rightarrow E_1 * E_2$	*E.place* := *newtemp*; *E.code* := E_1.*code* ‖ E_2.*code* ‖ *gen*(*E.place* ':=' E_1.*place* '*' E_2.*place*)
$E \rightarrow - E_1$	*E.place* := *newtemp*; *E.code* := E_1.*code* ‖ *gen*(*E.place* ':=' 'uminus' E_1.*place*)
$E \rightarrow (E_1)$	*E.place* := E_1.*place*; *E.code* := E_1.*code*
$E \rightarrow$ **id**	*E.place* := **id**.*place*; *E.code* := ''

These attributes represent labels created by a function new label that returns a new label every time it is called.

IMPLEMENTATIONS OF THREE-ADDRESS STATEMENTS:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

QUADRUPLES:

A quadruple is a record structure with four fields, which we call op, arg l, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unary operators like x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result. The quadruples in Fig. H.S(a) are for the assignment a: = b+ – c + b i – c. They are obtained from the three-address code
.The contents of fields arg 1, arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

TRIPLES:

To avoid entering temporary names into the symbol table. We might refer to a temporary value bi the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: op, arg 1 and arg2, as Shown below. The fields arg l and arg2, for the arguments of op, are either pointers to the symbol table (for programmer- defined names or constants) or pointers into the triple structure (for temporary values). Since three fields are used, this intermediate code format is known as triples.‘ Except for the treatment of programmer-defined names, triples correspond to the representation of a syntax tree or dag by an array of nodes, as in

	op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		A

Table 8.8 (a) : Qudraples

	op	Arg1	Arg2
(0)	uminus	C	
(1)	*	B	(0)
(2)	uminus	C	
(3)	*	B	(2)
(4)	+	(1)	(3)
(5)	:=	A	(4)

Table8.8(b) : Triples :Triples

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves. In practice, the

information needed to interpret the different kinds of entries in the arg 1 and arg2 fields can be encoded into the op field or some additional fields. The triples in Fig. 8.8(b) correspond to the quadruples in Fig. 8.8(a). Note that

the copy statement a:= t5 is encoded in the triple representation by placing a in the arg 1 field and using the operator assign. A ternary operation like x[i]: = y requires two entries in the triple structure, as shown in Fig. 8.9(a), while x: = y[i] is naturally represented as two operations inFig. 8.9(b).

	op	arg 1	arg 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y

	op	arg 1	arg 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

(b) x := y[i]

Fig. 8.9. More triple representations.

Indirect Triples

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples. For example, let us use an array statement to list pointers to triples in the desired order. Then the triples in Fig. 8.8(b) might be represented as in Fig. 8.10.

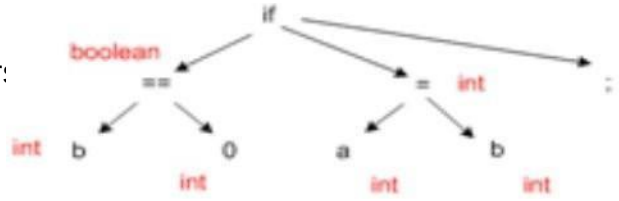
	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg 1	arg 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Figure 8.10 : Indirect Triples SEMANTIC

ANALYSIS : This phase focuses mainly on the

. Checking the semantics ,
. Error reporting
. Disambiguate overloaded operators
. Type coercion ,
. Static checking



- Type checking
-Control flow checking
- Uniqueness checking
- Name checking aspects of translation

Assume that the program has been verified to be syntactically correct and converted into some kind of intermediate representation (a parse tree). One now has parse tree available. The next phase will be semantic analysis of the generated parse tree. Semantic analysis also includes error reporting in case any semantic error is found out.

Semantic analysis is a pass by a compiler that adds semantic information to the parse tree and performs certain checks based on this information. It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.) Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding). Sometimes also some early code optimization is done inthis phase. For this phase the compiler usually maintains symbol tables in which it stores what each symbol (variable names, function names, etc.) refers to.

FOLLOWING THINGS ARE DONE IN SEMANTIC ANALYSIS:

Disambiguate Overloaded operators: If an operator is overloaded, one would like to specify the meaning of that particular operator because from one will go into code generation phase next.

TYPE CHECKING: The process of verifying and enforcing the constraints of types is called type checking. This may occur either at compile-time (a static check) or run-time (a dynamic check). Static type checking is a primary task of the semantic analysis carried out by a compiler. If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

UNIQUENESS CHECKING: Whether a variable name is unique or not, in the its scope.

Type coersion: If some kind of mixing of types is allowed. Done in languages which are notstrongly typed. This can be done dynamically as well as statically.

NAME CHECKS: Check whether any variable has a name which is not allowed. Ex. Name issame as an identifier (Ex. int in java).

- ∑ Parser cannot catch all the program errors
- ∑ There is a level of correctness that is deeper than syntax analysis
- ∑ Some language features cannot be modeled using context free grammar formalism

- Whether an identifier has been declared before use, this problem is of identifying a language {w α w | w ε Σ *}

- This language is not context free

A parser has its own limitations in catching program errors related to semantics, something that is deeper than syntax analysis. Typical features of semantic analysis cannot be modeled using context free grammar formalism. If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore.

Example: in

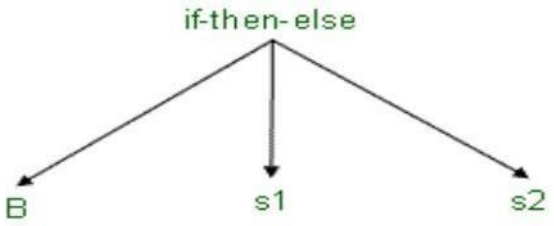
string x; int

y;
y = x + 3 the use of x is a type
errorint a, b;
a = b + c c is not declared

An identifier may refer to different variables in different parts of the program . An identifier may be usable in one part of the program but not another These are a couple of examples which tell us that typically what a compiler has to do beyond syntax analysis. The third point can be explained like this: An identifier x can be declared in two separate functions in the program, once of the type int and then of the type char. Hence the same identifier will have to be bound to these two different properties in the two different contexts. The fourth point can be explained in this manner: A variable declared within one function cannot be used within the scope of the definition of the other function unless declared there separately. This is just an example. Probably you can think of many more examples in which a variable declared in one scope cannot be used in another scope.

ABSTRACT SYNTAX TREE: Is nothing but the condensed form of a parse tree, It is

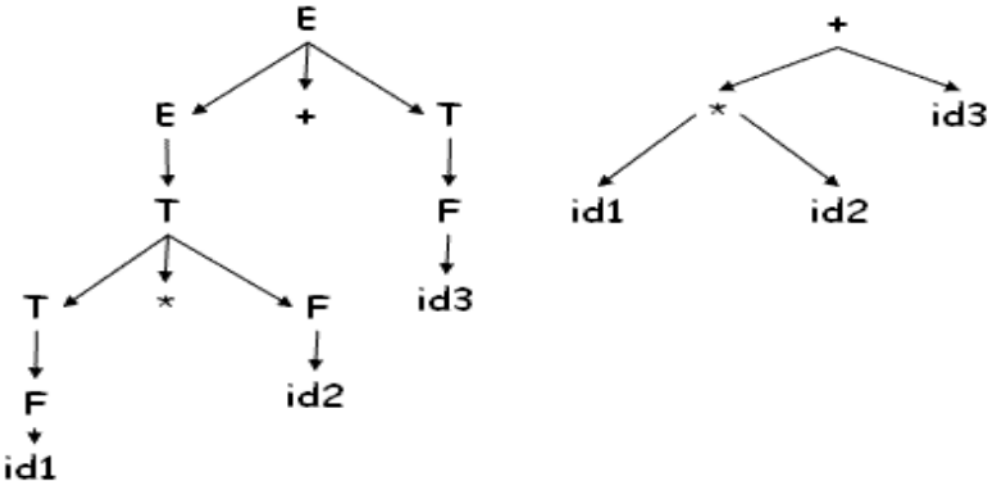
- ∑ Useful for representing language constructs so naturally.
- ∑ The production S→if B then s1 else s2 may appear as



In the next few slides we will see how abstract syntax trees can be constructed from syntax directed definitions. Abstract syntax trees are condensed form of parse trees. Normally operators and keywords appear as leaves but in an abstract syntax tree they are associated with the interior nodes that would be the parent of those leaves in the

parse tree. This is clearly indicated by the examples in these slides.

Chain of single productions may be collapsed, and operators move to the parent nodes



Chain of single productions are collapsed into one node with the operators moving up to becomethe node.

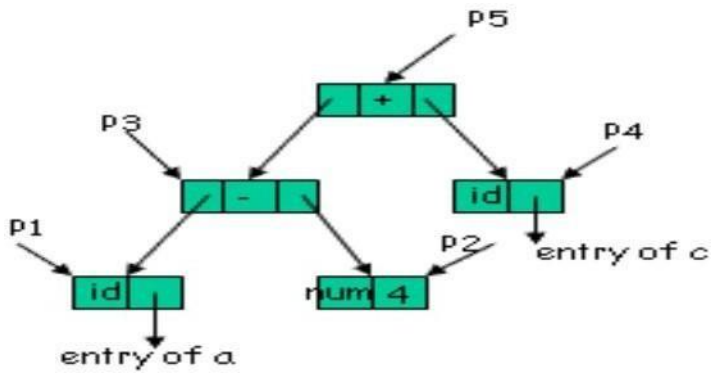
CONSTRUCTING ABSTRACT SYNTAX TREE FOR EXPRESSIONS:

In constructing the Syntax Tree, we follow the convention that :

. Each node of the tree can be represented as a record consisting of at least two fields to storeoperators and operands.
.***operators*** **:** one field for operator, remaining fields ptrs to operands mknode(op,left,right)
.***identifier*** **:** one field with label id and another ptr to symbol table mkleaf(id, id.entry)
.***number*** **:** one field with label num and another to keep the value of the number mkleaf(num,val)

Each node in an abstract syntax tree can be implemented as a record with several fields. In the node for an operator one field identifies the operator (called the label of the node) and the remaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may have additional fields to hold values (or pointers to values) of attributes attached to the node. The functions given in the slide are used to create the nodes of abstract syntax trees for expressions. Each function returns a pointer to a newly created note.

For Example: the followingsequence of function calls creates a parsetree for **w= a-4 + c**



P 1 = mkleaf(id, entry.a)P 2 = mkleaf(num, 4)

P 3 = mknode(-, P 1 , P 2)P

4 = mkleaf(id, entry.c)

P 5 = mknode(+, P 3 , P 4)

An example showing the formation of an abstract syntax tree by the given function calls for the expression a-4+c.The call sequence can be defined based on its postfix form, which is explained blow.

A- Write the postfix equivalent of the expression for which we want to construct a

syntax treeFor above string w=a-4+c, it is **a4-c+**

B- Call the functions in the sequence, as defined by the sequence in the postfix expression whichresults in the desired tree. In the case above, call mkleaf() for a, mkleaf() for 4, mknode() for -, mkleaf() for c , and mknode() for + at last.

1. P1 = **mkleaf**(id, a.entry) : A leaf node made for the identifier a, and an entry for a is made inthe symbol table.

2. P2 = **mkleaf**(num,4) : A leaf node made for the number 4, and entry for its value.

3. P3 = **mknode**(-,P1,P2) : An internal node for the -, takes the pointer to previously made nodesP1, P2 as arguments and represents the expression a-4.

4. P4 = **mkleaf**(id, c.entry) : A leaf node made for the identifier c , and an entry for c.entry madein the symbol table.

5. P5 = **mknode**(+,P3,P4) : An internal node for the + , takes the pointer to previously madenodes P3,P4 as arguments and represents the expression a- 4+c .

Following is the syntax directed definition for constructing syntax tree above

E → E 1 + T	E.ptr = mknode(+, E1 .ptr, T.ptr)
E → T	E.ptr = T.ptr
T → T 1 * F	T.ptr := mknode(*, T1 .ptr, F.ptr)
T → F	T.ptr := F.ptr
F → (E)	F.ptr := E.ptr
F → id	F.ptr := mkleaf(id, id.entry)
F → num	F.ptr := mkleaf(num, val)

Now we have the syntax directed definitions to construct the parse tree for a given grammar. Allthe rules mentioned in slide 29 are taken care of and an abstract syntax tree is formed.

ATTRIBUTE GRAMMARS: A CFG G=(V,T,P,S), is called an Attributed Grammar iff ,where in G, each grammar symbol XЄ VUT, has an associated set of attributes, and each production, p ЄP, is associated with a set of attribute evaluation rules called Semantic Actions.

In an **AG,** the **v**alues of attributes at a parse tree node are computed by semantic rules. There aretwo different specifications of **AGs** used by the **Semantic** Analyzer in evaluating the semantics of the program constructs. They are,

- **Syntax directed definition(SDD)s**
  - High level specifications
  - Hides implementation details
  - Explicit order of evaluation is not specified
- **Syntax directed Translation schemes(SDT)s**
  - ∑ Nothing but an SDD, which indicates order in which semantic rules are to beevaluated and
  - ∑ Allow some implementation details to be shown.

An **attribute grammar** is the formal expression of the syntax-derived semantic checks associated with a grammar. It represents the rules of a language not explicitly imparted by the syntax. In a practical way, it defines the information that is needed in the abstract syntax tree in order to successfully perform semantic analysis. This information is stored as attributes of the nodes of the abstract syntax tree. The values of those attributes are calculated by semantic rule.

There are two ways for writing attributes:

1) **Syntax Directed Definition(SDD):** Is a context free grammar in which a set of semantic actions are embedded (associated) with each production of G.

It is a high level specification in which implementation details are hidden, e.g., S.sys = A.sys + B.sys;

/* does not give any implementation details. It just tells us. This kind of attribute equation we will be using, Details like at what point of time is it evaluated and in what manner are hidden from the programmer.*/

E ⟶E1 + T	{ E.val = E1 .val+ E2.val }
E⟶ T	{ E.val = T.val }
T ⟶T 1 * F	{ T.val = T1 .val+ F.val)
T ⟶F	{ T.val = F.val }
F ⟶(E)	{ F.val = E.val }
F ⟶id	{ F.val = id.lexval }
F⟶ num	{ F.val = num.lexval }

2) **Syntax directed Translation(SDT) scheme**: Sometimes we want to control the way the attributes are evaluated, the order and place where they are evaluated. This is of a slightly lower level.

An SDT is an SDD in which semantic actions can be placed at any position in the body of theproduction.

For example , following SDT prints the prefix equivalent of an arithmetic expression consisting a
+ and * operators.

L → En{ printf(„E.val") }
E → { printf(„+") }E1 + TE
→ T

T → { printf(„*") }T 1 * FT
→ F
F → (E)

F → { printf(„id.lexval") }id
F → { printf(„num.lexval") } num

This action in an SDT, is executed as soon as its node in the parse tree is visited in a preordertraversal of the tree.

Conceptually both the SDD and SDT schemes will:

- ∑ Parse input token stream
- ∑ Build parse tree
- ∑ Traverse the parse tree to evaluate the semantic rules at the parse tree nodesEvaluation may:
- ∑ Generate code
- ∑ Save information in the symbol table
- ∑ Issue error messages
- ∑ Perform any other activity

To avoid repeated traversal of the parse tree, actions are taken simultaneously when a token isfound. So calculation of attributes goes along with the construction of the parse tree.

Along with the evaluation of the semantic rules the compiler may simultaneously generate code, save the information in the symbol table, and/or issue error messages etc. at the same time while building the parse tree.

This saves multiple passes of the parse

tree.Example

Number → sign
listsign → + | -
list → list bit | bit
bit → 0 | 1

Build attribute grammar that annotates Number with the value it represents

. Associate attributes with grammar symbols

symbol	**attributes**
Number	value
sign	negative
list	position, value
bit	position, value

production Attribute rule number→sign
listlist.position 0

if sign.negative

then number.value -
list.valueelse number.value
list.value
sign→+ sign.negative false sign→- sign.negative true list→bit
bit.position list.position
list.value
→bit.valuelist$_0$
list $_1$ bit
list$_1$ .position list $_0$ .position +
1bit.position list $_0$ .position
list$_0$ .value list$_1$ .value + bit.value
bit →0 bit.value 0 bit→1 bit.value $2^{bit.position}$

Explanation of attribute rules

Num -> sign list /*since list is the rightmost so it is assigned position 0
*Sign determines whether the value of the number would be
same or the negative of the value of list/

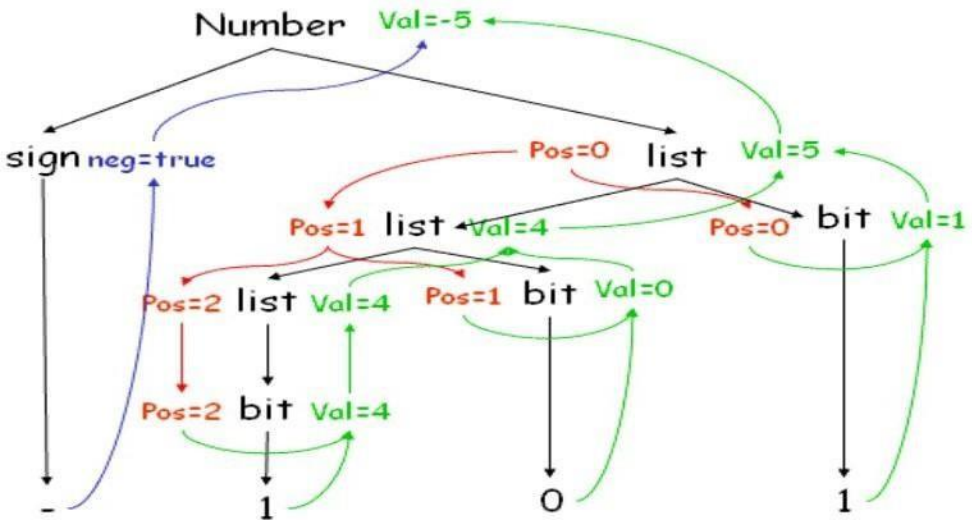
Sign -> + | - /*Set the Boolean attribute (negative) for sign*/

List -> bit /*bit position is the same as list position because this bit is the rightmost
value of the list is same as bit./

List0 -> List1 bit /*position and value
calculations*/**Bit -> 0 | 1** /*set the corresponding
value*/

Attributes of RHS can be computed from attributes of LHS and vice versa.

The Parse Tree and the Dependence graph are as under



.

Dependence graph shows the dependence of attributes on other attributes, along with the syntax tree. Top down traversal is followed by a bottom up traversal to resolve the dependencies. Number, val and neg are synthesized attributes. Pos is an inherited attribute.

Attributes : . Attributes fall into two classes namely *synthesized attributes* and *inherited attributes*. Value of a synthesized attribute is computed from the values of its children nodes. Value of an inherited attribute is computed from the sibling and parent nodes.

The attributes are divided into two groups, called synthesized attributes and inherited attributes. The synthesized attributes are the result of the attribute evaluation rules also using the values of the inherited attributes. The values of the inherited attributes are inherited from parent nodes and siblings.

Each grammar production ~~A→~~ a has associated with it a set of semantic rules of

the formb = f (c_1 , c_2 , ..., c_k) , Where f is a function, and either ,b is a synthesized

attribute of A Or

- b is an inherited attribute of one of the grammar symbols on the right

. attribute b depends on attributes c_1 , c_2 , ..., c_k

Dependence relation tells us what attributes we need to know before hand to calculate a particular attribute.

Here the value of the attribute b depends on the values of the attributes $\mathbf{c_1}$ to $\mathbf{c_k}$. If c_1 to c_k belong to the children nodes and b to A then b will be called a synthesized attribute. And if b belongs to one among a (child nodes) then it is an inherited attribute of one of the grammar symbols on the right.

Synthesized Attributes : A syntax directed definition that uses only synthesized attributes is said to be an S- attributed definition

. A parse tree for an S-attributed definition can be annotated by evaluating semantic rules forattributes

S-attributed grammars are a class of attribute grammars, comparable with L-attributed grammars but characterized by having no inherited attributes at all. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis, pose a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created *after* creation of all of their children. Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing .

Syntax Directed Definitions for a desk calculator program

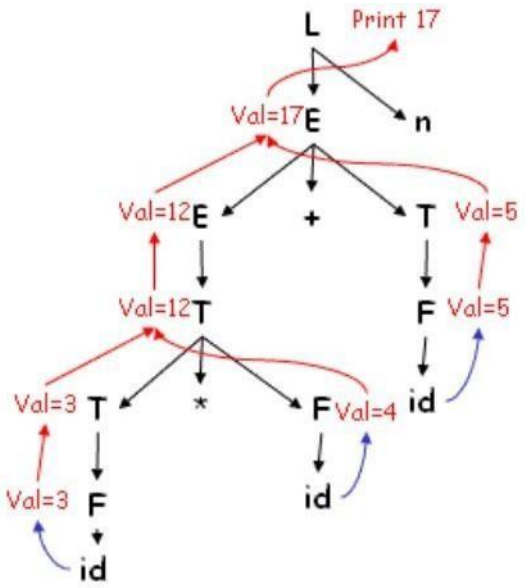
L → E n	Print (E.val)
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

. terminals are assumed to have only synthesized attribute values of which are supplied by lexicalanalyzer

. start symbol does not have any inherited attribute

This is a grammar which uses only synthesized attributes. Start symbol has no parents, hence noinherited attributes.

Parse tree for 3 * 4 + 5 n



Using the previous attribute grammar calculations have been worked out here for 3 * 4 + 5 n.Bottom up parsing has been done.

Inherited Attributes: An inherited attribute is one whose value is defined in terms ofattributes at the parent and/or siblings

. Used for finding out the context in which it appears

. possible to use only S-attributes but more natural to use inherited

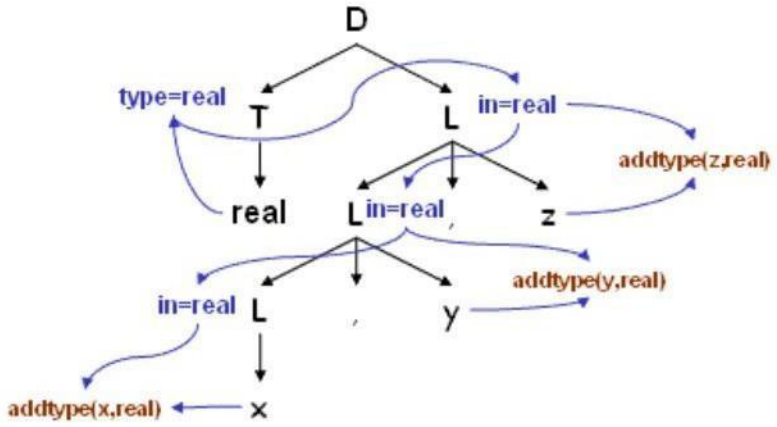
attributesD	T LL.in = T.type
T ⟶ real	T.type = real
T ⟶ int	T.type = int
L ⟶ L_1 , id	L_1 .in = L.in; addtype(id.entry, L.in)
L ⟶ id	addtype (id.entry,L.in)

Inherited attributes help to find the context (type, scope etc.) of a token e.g., the type of a tokenor scope when the same variable name is used multiple times in a program in different functions. An inherited attribute system may be replaced by an S -attribute system but it is more natural to use inherited attributes in some cases like the example given above.

Here addtype(a, b) functions adds a symbol table entry for the id a and attaches to it the type of b

.

Parse tree for real x, y, z



Dependence of attributes in an inherited attribute system. The value of in (an inherited attribute) at the three L nodes gives the type of the three identifiers x , y and z . These are determined by computing the value of the attribute T.type at the left child of the root and then valuating L.in topdown at the three L nodes in the right subtree of the root. At each L node the procedure addtypeis called which inserts the type of the identifier to its entry in the symbol table. The figure also shows the dependence graph which is introduced later.

Dependence Graph : . If an attribute b depends on an attribute c then the semantic rule for bmust be evaluated after the semantic rule for c

. The dependencies among the nodes can be depicted by a directed graph called dependencygraph

Dependency Graph **:** Directed graph indicating interdependencies among the synthesized andinherited attributes of various nodes in a parse tree.

Algorithm to construct dependency

graphfor each node **n** in the parse

tree do

for each attribute **a** of the grammar

symbol doconstruct a node in the

dependency graph

for **a**

for each node n in the parse tree do

for each semantic rule $b = f(c_1, c_2, ..., c_k)$ do

{ associated with production at n }

for i = 1 to k do

Construct an edge from c_i to b

An algorithm to construct the dependency graph. After making one node for everyattribute of all the nodes of the parse tree, make one edge from each of the other attributes on which it depends.

For example ,

- **Suppose A.a = f(X.x , Y.y) is a semantic rule for A → X Y**

- **If production A → X Y has the semantic rule X.x = g(A.a, Y.y)**

The semantic rule A.a = f(X.x , Y.y) for the production A -> XY defines the synthesized attribute a of A to be dependent on the attribute x of X and the attribute y of Y . Thus the dependency graph will contain an edge from X.x to A.a and Y.y to A.a accounting for the two dependencies. Similarly for the semantic rule X.x = g(A.a , Y.y) for the same production there will be an edge from A.a to X.x and an edg e from Y.y to X.x.

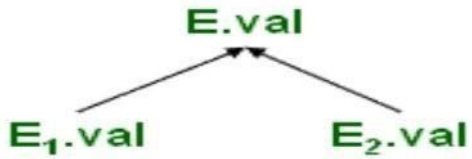
Example

. Whenever following production is used in a

parse treeE E 1 + E 2 E.val = E 1 .val

+ E 2 .val

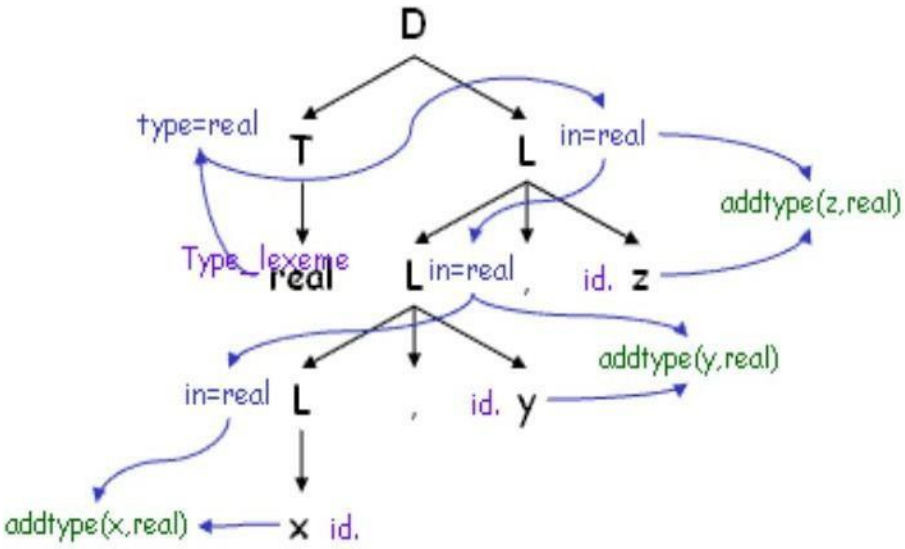
we create a dependency graph



The synthesized attribute E.val depends on E1.val and E2.val hence the two edges one each from E 1 .val & E 2 .val

For example, the dependency graph for the sting **real** id1, id2, id3

. Put a dummy synthesized attribute b for a semantic rule that consists of a procedure call



The figure shows the dependency graph for the statement real id1, id2, id3 along with the parse tree. Procedure calls can be thought of as rules defining the values of dummy synthesized attributes of the nonterminal on the left side of the associated production. Blue arrows constitute the dependency graph and black lines, the parse tree. Each of the semantic rules addtype (id.entry, L.in) associated with the L productions leads to the creation of the dummy attribute**.**

Evaluation Order :

Any topological sort of dependency graph gives a valid order in which semantic rules must beevaluated

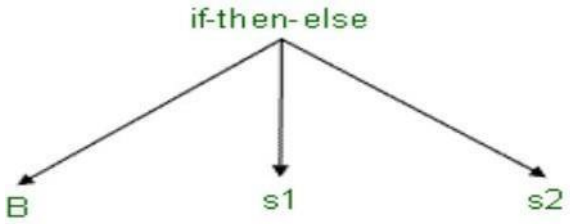
a4 =
reala5
= a4
addtype(id3.entry,
a5) a7 = a5
addtype(id2.entry,
a7)

a9 := a7 addtype(id1.entry, a9)

A topological sort of a directed acyclic graph is any ordering m1, m2, m3 .....mk of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes. Thus if mi -> mj is an edge from mi to mj then mi appears before mj in the ordering. The order of the statements shown in the slide is obtained from the topological sort of the dependency graph in the previous slide. 'an' stands for the attribute associated with the node numbered n in the dependency graph. The numbering is as shown in the previous slide.

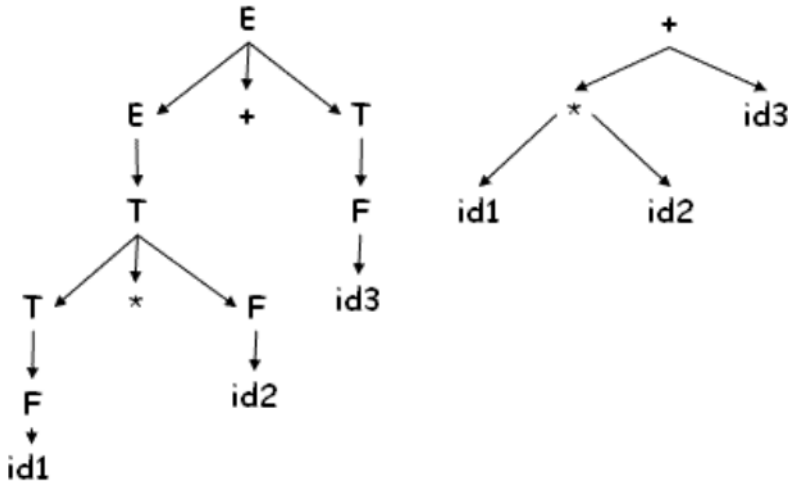
Abstract Syntax Tree is the condensed form of the parse tree, which is

. Useful for representing language constructs.
. The production : **S→if** B **then** s1 else s2 may appear as



In the next few slides we will see how abstract syntax trees can be constructed from syntax directed definitions. Abstract syntax trees are condensed form of parse trees. Normally operators and keywords appear as leaves but in an abstract syntax tree they are associated with the interior nodes that would be the parent of those leaves in the parse tree. This is clearly indicated by the examples in these slides.

. Chain of single productions may be collapsed, and operators move to the parent nodes



Chain of single production are collapsed into one node with the operators moving up to becomethe node.

For Constructing the Abstract Syntax tree for expressions,

. Each node can be represented as a record

. *operators* : one field for operator, remaining fields ptrs to operands mknode(op,left,right)

. *identifier* : one field with label id and another ptr to symbol table mkleaf(id,entry)

. *number* : one field with label num and another to keep the value of the numbermkleaf(num,val)

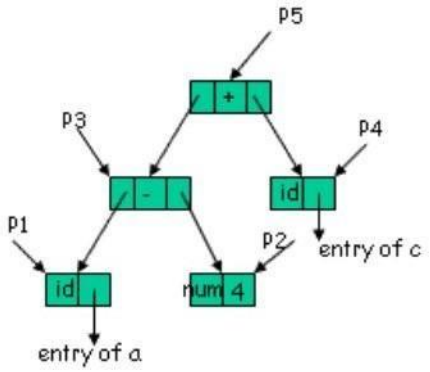
Each node in an abstract syntax tree can be implemented as a record with several fields. In the node for an operator one field identifies the operator (called the label of the node) and the remaining contain pointers to the nodes for operands. Nodes of an abstract syntax tree may have additional fields to hold values (or pointers to values) of attributes attached to the node. The functions given in the slide are used to create the nodes of abstract syntax trees for expressions. Each function returns a pointer to a newly created note.

Example : **The following sequence of function calls creates a parse tree for a- 4 + c**

P 1 = mkleaf(id, entry.a)P 2 = mkleaf(num, 4)

P 3 = mknode(-, P 1 , P 2)P 4 = mkleaf(id, entry.c)

P 5 = mknode(+, P 3 , P 4)



An example showing the formation of an abstract syntax tree by the given function calls for theexpression a-4+c.The call sequence can be explained as:

1. P1 = mkleaf(id,entry.a) : A leaf node made for the identifier Qa R and an entry for Qa R is made in the symbol table.
2. P2 = mkleaf(num,4) : A leaf node made for the number Q4 R.
3. P3 = mknode(-,P1,P2) : An internal node for the Q- Q.I takes the previously made nodes as arguments and represents the expression Qa-4 R.
4. P4 = mkleaf(id,entry.c) : A leaf node made for the identifier Qc R and an entry for Qc R is made in the symbol table.
5. P5 = mknode(+,P3,P4) : An internal node for the Q+ Q.I takes the previously made nodes as arguments and represents the expression Qa- 4+c R.

A syntax directed definition for constructing syntax tree

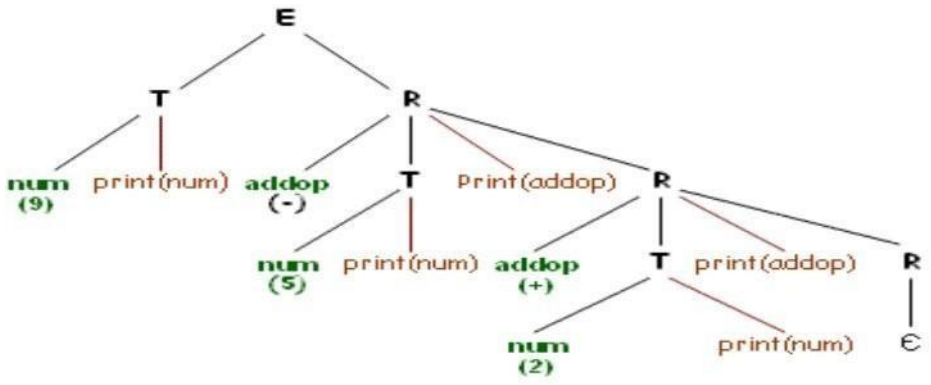
E ⟶ E 1 + T	E.ptr = mknode(+, E 1 .ptr, T.ptr)
E ⟶ T	E.ptr = T.ptr
T ⟶ T 1 * F	T.ptr := mknode(*, T 1 .ptr, F.ptr)
T ⟶ F	T.ptr := F.ptr
F ⟶ (E)	F.ptr := E.ptr
F ⟶ id	F.ptr := mkleaf(id, entry.id)
F ⟶ num	F.ptr := mkleaf(num,val)

Now we have the syntax directed definitions to construct the parse tree for a given grammar. Allthe rules mentioned in slide 29 are taken care of and an abstract syntax tree is formed.

Translation schemes : A CFG where semantic actions occur within the right hand side ofproduction, A translation scheme to map infix to postfix.

E ⟶ T R
R⟶ addop T { print(addop)} R |
e⊤⟶ num {print(num)}

Parse tree for 9 - 5 + 2



We assume that the actions are terminal symbols and Perform depth first order traversal to obtain9 5 - 2 +.

- ∑ When designing translation scheme, ensure attribute value is available when referred to
- ∑ In case of synthesized attribute it is trivial (why?)

In a translation scheme, as we are dealing with implementation, we have to explicitly worry about the order of traversal. We can now put in between the rules some

actions as part of the RHS. We put this rules in order to control the order of traversals. In the given example, we have two terminals (num and addop). It can generally be seen as a number followed by R (which

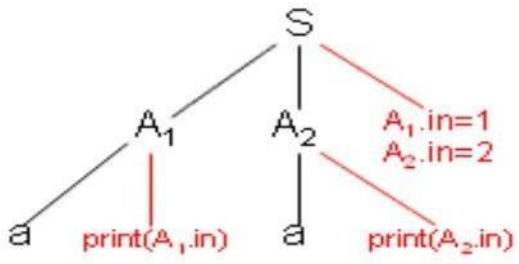
necessarily has to begin with an addop). The given grammar is in infix notation and we need to convert it into postfix notation. If we ignore all the actions, the parse tree is in black, without the red edges. If we include the red edges we get a parse tree with actions. The actions are so far treated as a terminal. Now, if we do a depth first traversal, and whenever we encounter a action we execute it, we get a post-fix notation. In translation scheme, we have to take care of the evaluation order; otherwise some of the parts may be left undefined. For different actions, different result will be obtained. Actions are something we write and we have to control it. Please note that translation scheme is different from a syntax driven definition. In the latter, we do not have any evaluation order; in this case we have an explicit evaluation order. By explicit evaluation order we have to set correct action at correct places, in order to get the desired output. Place of each action is very important. We have to find appropriate places, and that is that translation scheme is all about. If we talk of only synthesized attribute, the translation scheme is very trivial. This is because, when we reach we know that all the children must have been evaluated and all their attributes must have also been dealt with. This is because finding the place for evaluation is very simple, it is the rightmost place.

In case of both inherited and synthesized attributes

. An inherited attribute for a symbol on rhs of a production must be computed in an action beforethat symbol

S →A 1 A 2 {A 1 .in = 1,A 2 .in = 2}

A →a {print(A.in)}



Depth first order traversal gives error *undefined*

. A synthesized attribute for non terminal on the lhs can be computed after all attributes itreferences, have been computed. The action normally should be placed at the end of rhs

We have a problem when we have both synthesized as well as inherited attributes. For the given example, if we place the actions as shown, we cannot evaluate it. This is because, when doing a depth first traversal, we cannot print anything for A1. This is because A1 has not yet been initialized. We, therefore have to find the correct places for the actions. This can be that the inherited attribute of A must be calculated on its left. This can be seen logically from the definition of L-attribute definition, which says that when we reach a node, then everything on its left must have been computed. If we do this, we will always have the attribute evaluated at the

correct place. For such specific cases (like the given example) calculating anywhere on the leftwill work, but generally it must be calculated immediately at the left.

Example: Translation scheme for EQN

S →B	B.pts = 10S.ht = B.ht
B →B$_1$ B$_2$	B$_1$ .pts = B.ptsB $_2$ .pts = B.pts B.ht = max(B $_1$ .ht,B$_2$ .ht)
B →B$_1$ sub B $_2$	B$_1$ .pts = B.pts; B $_2$ .pts = shrink(B.pts) B.ht = disp(B$_1$ .ht,B$_2$ .ht)
B →text	B.ht = text.h * B.pts

We now look at another example. This is the grammar for finding out how do I compose text. EQN was equation setting system which was used as an early type setting system for UNIX. It was earlier used as an latex equivalent for equations. We say that start symbol is a block: S - >B We can also have a subscript and superscript. Here, we look at subscript. A Block is composedof several blocks: B -> B1B2 and B2 is a subscript of B1. We have to determine what is the point size (inherited) and height Size (synthesized). We have the relevant function for height and point size given along side. After putting actions in the right place

$$
\begin{aligned}
S \longrightarrow\ & \{B.pts = 10\}\ B \\
& \{S.ht = B.ht\} \\
B \longrightarrow\ & \{B_1.pts = B.pts\}\ B_1 \\
& \{B_2.pts = B.pts\}\ B_2 \\
& \{B.ht = max(B_1.ht, B_2.ht)\} \\
B \longrightarrow\ & \{B_1.pts = B.pts\}\ B_1\ sub \\
& \{B_2.pts = shrink(B.pts)\}\ B_2 \\
& \{B.ht = disp(B_1.ht, B_2.ht)\} \\
B \longrightarrow\ & text\ \{B.ht = text.h * B.pts\}
\end{aligned}
$$

We have put all the actions at the correct places as per the rules stated. Read it from left to right, and top to bottom. We note that all inherited attribute are calculated on the left of B symbols and synthesized attributes are on the right.

Top down Translation: Use predictive parsing to implement L attributed definitions

E E-1 + T E.val := E 1 .val + T.val

E E-1 - T E.val := E 1 .val - T.val

E → T E.val := T.val

T → (E) T.val := E.val

T → num T.val := num.lexval

We now come to implementation. We decide how we use parse tree and L-attribute definitions to construct the parse tree with a one-to-one correspondence. We first look at the top- down translation scheme. The first major problem is left recursion. If we remove left recursionby our standard mechanism, we introduce new symbols, and new symbols will not work with theexisting actions. Also, we have to do the parsing in a single pass.

TYPE SYSTEM AND TYPE CHECKING:

. If both the operands of arithmetic operators +, -, x are integers then the result is of type integer
. The result of unary & operator is a pointer to the object referred to by the operand.
- If the type of operand is ***X*** then type of result is ***pointer to X***

In Pascal, types are classified under:

1. *Basic* types: These are atomic types with no internal structure. They include the types boolean,character, integer and real.

2. *Sub-range* types: A sub-range type defines a range of values within the range of another type.For example, type A = 1..10; B = 100..1000; U = 'A'..'Z';

3. *Enumerated* types: An enumerated type is defined by listing all of the possible values for the type. For example: type Colour = (Red, Yellow, Green); Country = (NZ, Aus, SL, WI, Pak, Ind, SA, Ken, Zim, Eng); Both the sub-range and enumerated types can be treated as basic types.

4. *Constructed* types: A constructed type is constructed from basic types and other basic types. Examples of constructed types are arrays, records and sets. Additionally, pointers and functions can also be treated as constructed types.

TYPE EXPRESSION:

It is an expression that denotes the type of an expression. The type of a language construct isdenoted by a type expression

- ∑ It is either a basic type or it is formed by applying operators called *type constructor* toother type expressions

∑ A type constructor applied to a type expression is a type expression
∑ A basic type is type expression

- *type error* : error during type checking
- *void* : no type value

The type of a language construct is denoted by a type expression. A type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions. Formally, a type expression is recursively defined as:

1. A basic type is a type expression. Among the basic types are *boolean , char , integer ,* and *real* .A special basic type, *type_error* , is used to signal an error during type checking. Another special basic type is *void* which denotes "the absence of a value" and is used to check statements.

2. Since type expressions may be named, a type name is a type expression.

3. The result of applying a type constructor to a type expression is a type expression.

4. Type expressions may contain variables whose values are type expressions themselves.

TYPE CONSTRUCTORS: are used to define or construct the type of user defined types basedon their dependent types.

Arrays : If T is a type expression and I is a range of integers, then *array* (*I* , *T*) is the typeexpression denoting the type of array with elements of type T and index set I.

For example, the Pascal declaration, var A: array[1 .. 10] of integer; associates the typeexpression ***array* (1..10, *integer*)** with A.

Products *:* If *T1* and *T2* are type expressions, then their Cartesian product ***T1 X T2*** is also a type expression.

Records *:* A record type constructor is applied to a tuple formed from field names and fieldtypes. For example, the declaration

Consider the

declarationtype row =

```
record
addr : integer;
lexeme : array [1 .. 15] of
charend;
var table: array [1 .. 10] of row;
```

The type row has type expression : **record ((addr x integer) x (lexeme x array(1 .. 15, char)))**

and type expression of table is **array(1 .. 10, row)**

Note: Including the field names in the type expression allows us to define another record typewith the same fields but with different names without being forced to equate the two.

Pointers: If T is a type expression, then ***pointer*** (***T***) is a type expression denoting the type"pointer to an object of type T".
For example, in Pascal, the declaration
var p: row declares variable p to have type ***pointer(*** **row** ***)***.

Functions : Analogous to mathematical functions, functions in programming languages may be defined as mapping a domain type D to a range type R. The type of such a function is denoted bythe type expression D R. For example, the built-in function mod of Pascal has domain type int X int, and range type *int* . Thus we say mod has the type: **int x int -> int**

As another example, according to the Pascal declarationfunction f(a, b: char) : integer;

Here the type of f is denoted by the type expression is **char x char pointer(integer)**

SPECIFICATIONS OF A TYPE CHECKER: Consider a language which consists of asequence of declarations followed by a single expression

P ⟶ D ; E

D ⟶ D ; D | id : T

T ⟶ char | integer | array [num] of T |

^ ~~T E~~ literal | num | E mod E | E [E] | E

^

A **type checker** is a translation scheme that synthesizes the type of each expression from the types of its sub-expressions. Consider the above given grammar that generates programs consisting of a sequence of declarations D followed by a single expression E.

Specifications of a type checker for the language of the above grammar: A program generated by this grammar is

key : integer;
key mod
1999

Assumptions:

1. The language has three basic types: *char* , *int* and *type-error*

2. For simplicity, all arrays start at 1. For example, the declaration array[256] of char leads to thetype expression *array* (1.. 256, char).

Rules for Symbol Table entry

D ⟶ id : T	addtype(id.entry, T.type)
T ⟶ char	T.type = char

T → integer	T.type = int
T → ^T_1	T.type = pointer(T_1 .type)
T → array [num] of T_1	T.type = array(1..num, T_1 .type)

TYPE CHECKING OF FUNCTIONS :

Consider the Syntax Directed Definition,

$E \rightarrow E_1 (E_2)$	E. type = if E_2 .type == s and
	E_1 .type == s $\rightarrow$t
	then t
	else type-error

The rules for the symbol table entry are specified above. These are basically the way in whichthe symbol table entries corresponding to the productions are done.

Type checking of functions

The production E -> E (E) where an expression is the application of one expression to another can be used to represent the application of a function to an argument. The rule for checking the type of a function application is

E -> E1 (E2) { *E.type* := if *E2.type* == s and *E1. type* == *s -> t* then *t* else *type_error* **}**

This rule says that in an expression formed by applying E1 to E2, the type of E1 must be a function *s -> t* from the type *s* of E2 to some range type *t* ; the type of E1 (E2) is *t* . The above rule can be generalized to functions with more than one argument byconstructing a product type consisting of the arguments. Thus n arguments of type *T1* , *T2*

... *Tn* can be viewed as a single argument of the type *T1 X T2 ... X Tn* . For

example,root : (real real) X real real

declares a function root that takes a function from reals to reals and a real as arguments andreturns a real. The Pascal-like syntax for this declaration is

function root (function f (real) : real; x: real) : real

TYPE CHECKING FOR EXPRESSIONS: consider the following SDD for expressions

E $\rightarrow$literal	E.type = char
E $\rightarrow$num	E.type = integer
E $\rightarrow$id	E.type = lookup(id.entry)
$E \rightarrow E_1$ mod E_2	E.type = if E_1 .type == integer and

E_2
.type==integer
then integer

	else type_error
E → E_1 [E_2]	E.type = if E_2 .type==integer and E_1 .type==array(s,t) then t else type_error
E → E_1 ^	E.type = if E_1 .type==pointer(t)then t else type_error

To perform type checking of expressions, following rules are used. Where the synthesizedattribute type for E gives the type expression assigned by the type system to the expression generated by E.

The following semantic rules say that constants represented by the tokens literal and num havetype *char* and *integer* , respectively:

E -> literal { *E.type* := *char* } E -

> num { *E.type* := *integer* }

. The function *lookup* (*e*) is used to fetch the type saved in the symbol-table entry pointed to by
e. When an identifier appears in an expression, its declared type is fetched and assigned to theattribute type:

E -> id { *E.type* := *lookup* (id . *entry*) }

. According to the following rule, the expression formed by applying the mod operator to twosub-expressions of type *integer* has type *integer* ; otherwise, its type is *type_error* .

E -> E1 mod E2 { *E.type* := if *E1.type* == *integer* and *E2.type* == *integer* then *integer* else

type_error }

In an array reference E1 [E2], the index expression E2 must have type *integer* , inwhich casethe result is the element type *t* obtained from the type *array* (*s, t*) of E1.

E -> E1 [E2] { *E.type* := if *E2.type* == *integer* and *E1.type* == *array* (*s, t*) then *t* else

type_error }

Within expressions, the postfix operator yields the object pointed to by its operand. The type of Eis the type *t* of the object pointed to by the pointer E:

E E1 { *E.type* := if *E1.type* == *pointer* (*t*) then *t* else *type_error*

TYPE CHECKING OF STATEMENTS: Statements typically do not have values. Special basic type *void* can be assigned to them. Consider the SDD for the grammar below which generates Assignment statements conditional, and looping statements.

S ⟶id := E	S.Type = if id.type == E.type then void else type_error
S ⟶if E then S1	S.Type = if E.type == boolean then S1.type else type_error
S ⟶while E do S1	S.Type = if E.type == boolean then S1.type else type_error
S⟶S1 ; S2	S.Type = if S1.type == voidand S2.type == void then void else type_error

Since statements do not have values, the special basic type *void* is assigned to them, but if anerror is detected within a statement, the type assigned to the statement is *type_error* .

The statements considered below are assignment, conditional, and while statements. Sequences of statements are separated by semi-colons. The productions given below can be combined with those given before if we change the production for a complete program to P -> D; S. The program now consists of declarations followed by statements.

Rules for type checking the statements are given below.

1. S id := E { *S.type* := if id . *type == E.type* then *void* else *type_error* }

This rule checks that the left and right sides of an assignment statement have the same type.

2. S if E then S1 { *S.type* := if *E.type* == *boolean* then *S1.type* else *type_error* }

This rule specifies that the expressions in an if -then statement must have the type *boolean* .

3. S while E do S1 { *S.type* := if *E.type* == *boolean* then *S1.type* else *type_error* }

This rule specifies that the expression in a while statement must have the type *boolean* .

4. S S1; S2 { *S.type* := if *S1.type* == *void* and *S2.type* == *void* then *void* else *type_error* }

UNIT – V

Code optimization and Code generation

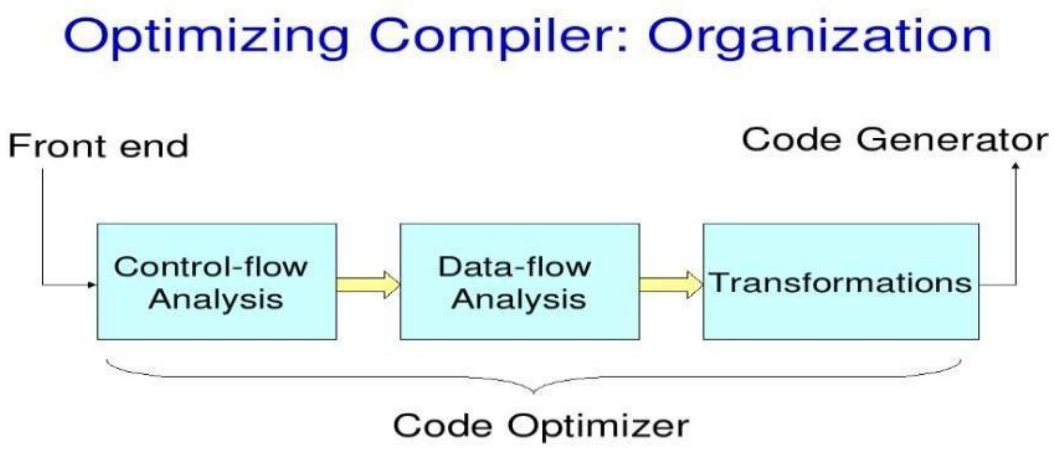
CODE OPTIMIZATION

Considerations for optimization : The code produced by the straight forward compiling algorithms can often be made to run faster or take less space,or both. This improvement is achieved by program transformations that are traditionally called optimizations. Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine. Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

Criteria for code improvement transformations

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.
- First, the transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error.
- Second, a transformation must, on the average, speed up programs by a measurable amount.
- Third, the transformation must be worth the effort.

Some transformations can only be applied after detailed, often time-consuming analysis of the source program, so there is little point in applying them to programs that will be run only a few times.



OBJECTIVES OF OPTIMIZATION: The main objectives of the optimization techniques areas follows

1. Exploit the fast path in case of multiple paths fro a given situation.
2. Reduce redundant instructions.
3. Produce minimum code for maximum work.
4. Trade off between the size of the code and the speed with which it gets executed.
5. Place code and data together whenever it is required to avoid unnecessary searching ofdata/code

During code transformation in the process of optimization, the basic requirements are as follows:

1. Retain the semantics of the source code.
2. Reduce time and/ or space.
3. Reduce the overhead involved in the optimization process.

Scope of Optimization: Control-Flow Analysis

Consider all that has happened up to this point in the compiling process—lexical analysis, syntactic analysis, semantic analysis and finally intermediate-code generation. The compiler has done an enormous amount of analysis, but it still doesn't really know how the program does what it does. In control-flow analysis, the compiler figures out even more information about how the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

Control-flow analysis begins by constructing a control-flow graph, which is a graph of the different possible paths program flow could take through a function. To build the graph, we first divide the code into basic blocks. A basic block is a segment of the code that a program must enter at the beginning and exit only at the end. This means that only the first statement can be reached from outside the block (there are no branches into the middle of the block) and all statements are executed consecutively after the first one is (no branches or halts until the exit). Thus a basic block has exactly one entry point and one exit point. If a program executes the first instruction in a basic block, it must execute every instruction in the block sequentially after it.

A basic block begins in one of several ways:

- The entry point into the function

- The target of a branch (in our example, any label)
- The instruction immediately following a branch or a return

A basic block ends in any of the following ways:

- A jump statement
- A conditional or unconditional branch
- A return statement

Now we can construct the control-flow graph between the blocks. Each basic block is a node in the graph, and the possible different routes a program might take are the connections, i.e.if a block ends with a branch, there will be a path leading from that block to the branch target. The blocks that can follow a block are called its successors. There may be multiple successors or just one. Similarly the block may have many, one, or no predecessors. Connect up the flow graphfor Fibonacci basic blocks given above. What does an if then-else look like in a flow graph?What about a loop? You probably have all seen the gcc warning or javac error about: "Unreachable code at line XXX." How can the compiler tell when code is unreachable?

LOCAL OPTIMIZATIONS

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information; we just work with the statements within the block. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform. We'll consider some of the more common local optimizations as examples.

FUNCTION PRESERVING TRANSFORMATIONS

- ∑ Common sub expression elimination
- ∑ Constant folding
- ∑ Variable propagation
- ∑ Dead Code Elimination
- ∑ Code motion
- ∑ Strength Reduction

1. Common Sub Expression Elimination:

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. An

expression is alive if the operands used to compute the expression have not been changed. Anexpression that is no longer alive is dead.

Example :

```
a=b*c;
d=b*c+x-
y;
```

We can eliminate the second evaluation of b*c from this code if none of the interveningstatements has changed its value. We can thus rewrite the code as

```
t1=b*c;
a=t1;
d=t1+x-
y;
```

Let us consider the following

```
codea=b*c;
b=x;
d=b*c+ x-
y;
```

in this code, we can not eliminate the second evaluation of b*c because the value of b is changeddue to the assignment b=x before it is used in calculating d.

We can say the two expressions are common if

- ∑ They lexically equivalent i.e., they consist of identical operands connected to each otherby identical operator.
- ∑ They evaluate the identical values i.e., no assignment statements for any of their operandsexist between the evaluations of these expressions.
- ∑ The value of any of the operands use in the expression should not be changed even due tothe procedure call.

Example :

```
c=a*b;
x=a;
d=x*b;
```

We may note that even though expressions a*b and x*b are common in the above code,they can not be treated as common sub expressions.

2. Variable Propagation:

Let us consider the above code once again

```
c=a*b;
x=a;
d=x*b+4
;
```

if we replace x by a in the last statement, we can identify a*b and x*b as common sub expressions. This technique is called variable propagation where the use of one variable is replaced by another variable if it has been assigned the value of same

Compile Time evaluation

The execution efficiency of the program can be improved by shifting execution time actions to compile time so that they are not performed repeatedly during the program execution. We can evaluate an expression with constants operands at compile time and replace that expression by a single value. This is called folding. Consider the following statement:

```
a= 2*(22.0/7.0)*r;
```

Here, we can perform the computation 2*(22.0/7.0) at compile time itself.

3. Dead Code Elimination:

If the value contained in the variable at a point is not used anywhere in the program subsequently, the variable is said to be dead at that place. If an assignment is made to a dead variable, then that assignment is a dead assignment and it can be safely removed from the program.

Similarly, a piece of code is said to be dead, which computes value that are never used anywherein the program.

```
c=a*b;
x=a;
d=x*b+4
;
```

Using variable propagation, the code can be written as follows:

```
c=a*b;
x=a;
d=a*b+4
;
```

Using Common Sub expression elimination, the code can be written as follows:

```
t1=
a*b;
c=t1;
x=a;
d=t1+4
;
```

Here, x=a will considered as dead code. Hence it is eliminated.

```
t1= a*b;
c=t1;
d=t1+4
;
```

4. Code Movement:

The motivation for performing code movement in a program is to improve the execution time of the program by reducing the evaluation frequency of expressions. This can be done by moving the evaluation of an expression to other parts of the program. Let us consider the bellow code:

```
If(a<10)
{
b=x^2-y^2;
}
else
{
b=5
;
a=( x^2-y^2)*10;
}
```

At the time of execution of the condition a<10, x^2-y^2 is evaluated twice. So, we can optimizethe code by moving the out side to the block as follows:

```
t= x^2-
y^2;
If(a<10)
{
b=t;
}
else
{
b=5
;
a=t*10;
}
```

5. Strength Reduction:

In the frequency reduction transformation we tried to reduce the execution frequency of the expressions by moving the code. There is other class of transformations which perform equivalent actions indicated in the source program by reducing the strength of operators. By strength reduction, we mean replacing the high strength operator with low strength operator with out affecting the program meaning. Let us consider the bellow example:

```
i=1;
while (i<10)
{
y=i*4;
}
```

The above can written as follows:i=1;

```
t=4;
```

```
while (i<10)
{
y=t;
t=t+4
;
}
```

Here the high strength operator * is replaced with +.

GLOBAL OPTIMIZATIONS, DATA-FLOW ANALYSIS:

So far we were only considering making changes within one basic block. With some Additional analysis, we can apply similar optimizations across basic blocks, making them global optimizations. It's worth pointing out that global in this case does not mean across the entire program. We usually optimize only one function at a time. Inter procedural analysis is an even larger task, one not even attempted by some compilers.

The additional analysis the optimizer does to perform optimizations across basic blocks is called **data-flow analysis**. Data-flow analysis is much more complicated than control-flowanalysis, and we can only scratch the surface here.

Let's consider a global common sub expression elimination optimization as our example. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be **available at that point**. Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis. Each block is a node in the flow graph of a program. The **successor** set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. An expression is defined at the point where it is assigned a value and killed when one of its operands is subsequently assigned a new value. An expression is available at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not subsequently killed. Lets define such useful functions in DF analysis in following lines.

avail[B] = set of expressions available on entry to block B
exit[B] = set of expressions available on exit from B
avail[B] = ∩ **exit[x]: x** ∈ **pred[B]** (i.e. B has available the intersection of the exit of its predecessors)

killed[B] = set of the expressions killed in B

defined[B] = set of expressions defined in B

exit[B] = **avail[B]** - **killed[B]** + **defined[B]**

$$avail[B] = \cap\ (avail[x] - killed[x] + defined[x]) : x \in pred[B]$$

Here is an **Algorithm for Global Common Sub-expression Elimination**:

1) First, compute defined and killed sets for each basic block (this does not involve any of itspredecessors or successors).

2) Iteratively compute the avail and exit sets for each block by running the following algorithmuntil you hit a stable fixed point:

a) Identify each statement **s** of the form **a = b op c** in some block B such that **b op c** is available at the entry to B and neither **b** nor **c** is redefined in B prior to **s**.

b) Follow flow of control backward in the graph passing back to but not through eachblock that defines **b op c**. The last computation of **b op c** in such a block reaches **s**.

c) After each computation **d = b op c** identified in step 2a, add statement **t = d** to thatblock where t is a new temp.

d) Replace **s** by **a = t**.

Try an example to make things clearer:main:

```
BeginFunc 28;
    b = a + 2
    ;c = 4 * b
    ;
    tmp1 = b < c;
    ifNZ tmp1 goto L1
    ;b = 1 ;
    L1:
    d = a + 2
;EndFunc ;
```

First, divide the code above into basic blocks. Now calculate the available expressions for each block. Then find an expression available in a block and perform step 2c above. What common sub-expression can you share between the two blocks? What if the above code were:

```
main:
BeginFunc 28;
    b = a + 2
    ;c = 4 * b
    ;
    tmp1 = b < c ;
    IfNZ tmp1 Goto L1
    ;b = 1 ;
    z = a + 2 ; <========= an additional line here
L1:
```

```
        d = a + 2 ;
    EndFunc ;
```

MACHINE OPTIMIZATIONS

In final code generation, there is a lot of opportunity for cleverness in generating efficient target code. In this pass, specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for this particular architecture.

REGISTER ALLOCATION:

One machine optimization of particular importance is register allocation, which is perhaps the single most effective optimization for all architectures. Registers are the fastest kind of memory available, but as a resource, they can be scarce.

The problem is how to minimize traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus and the different levels of caches. Your Decaf back-end uses a very naïve and inefficient means of assigning registers, it just fills them before performing an operation and spills them right afterwards.

A much more effective strategy would be to consider which variables are more heavily in demand and keep those in registers and spill those that are no longer needed or won't beneeded until much later.

One common register allocation technique is called "register coloring", after the central idea to view register allocation as a graph coloring problem. If we have 8 registers, then we try to color a graph with eight different colors. The graph‘s nodes are made of "webs" and the arcs are determined by calculating interference between the webs. A web represents a variable‘s definitions, places where it is assigned a value (as in **x = …**), and the possible different uses of those definitions (as in **y = x + 2**). This problem, in fact, can be approached as another graph. The definition and uses of a variable are nodes, and if a definition reaches a use, there is an arc between the two nodes. If two portions of a variable‘s definition-use graph are unconnected, thenwe have two separate webs for a variable. In the interference graph for the routine, each node is aweb. We seek to determine which webs don't interfere with one another, so we know we can use the same register for those two variables. For example, consider the following code:

```
i = 10;
j = 20;
x = i + j;
y = j +
k;
```

We say that **i** interferes with **j** because at least one pair of **i**‘s definitions and uses is separated by a definition or use of **j**, thus, **i** and **j** are "alive" at the same time. A variable is alive between the time it has been defined and that definition‘s last use, after which the variable is dead. If two variables interfere, then we cannot use the same register for each. But two variables that don't interfere can since there is no overlap in the liveness and can occupy the same register. Once we have the interference graph constructed, we r-color it so that no two adjacent nodes share the same color (r is the number of registers we have, each color represents a different register).

We may recall that graph-coloring is NP-complete, so we employ a heuristic rather than an optimal algorithm. Here is a simplified version of something that might be used:

1. Find the node with the least neighbors. (Break ties arbitrarily.)
2. Remove it from the interference graph and push it onto a stack
3. Repeat steps 1 and 2 until the graph is empty.
4. Now, rebuild the graph as follows:
    a. Take the top node off the stack and reinsert it into the graph
    b. Choose a color for it based on the color of any of its neighbors presently in the graph,rotating colors in case there is more than one choice.
    c. Repeat a , and b until the graph is either completely rebuilt, or there is no coloravailable to color the node.

If we get stuck, then the graph may not be r-colorable, we could try again with a different heuristic, say reusing colors as often as possible. If no other choice, we have to spill a variable to memory.

INSTRUCTION SCHEDULING:

Another extremely important optimization of the final code generator is instruction scheduling. Because many machines, including most RISC architectures, have some sort of pipelining capability, effectively harnessing that capability requires judicious ordering of instructions.

In MIPS, each instruction is issued in one cycle, but some take multiple cycles to complete. It takes an additional cycle before the value of a load is available and two cycles for a branch to reach its destination, but an instruction can be placed in the "delay slot" after a branch and executed in that slack time. On the left is one arrangement of a set of instructions that requires 7 cycles. It assumes no hardware interlock and thus explicitly stalls between the second and third slots while the load completes and has a Dead cycle after the branch because the delay slot holds a noop. On the right, a more favorable rearrangement of the same instructions will execute in 5 cycles with no dead Cycles.

```
lw          $t2,
4($fp)lw $t3,
8($fp)noop
add $t4, $t2,
$t3subi $t5,
$t5, 1 goto L1
noop
```

```
lw $t2, 4($fp)
lw $t3, 8($fp)
subi $t5, $t5, 1
goto L1

add $t4, $t2, $t3
```

PEEPHOLE OPTIMIZATIONS:

Peephole optimization is a pass that operates on the target assembly and only

considers a few instructions at a time (through a "peephole") and attempts to do simple, machine dependent

code improvements. For example, peephole optimizations might include elimination of multiplication by 1, elimination of load of a value into a register when the previous instruction stored that value from the register to a memory location, or replacing a sequence of instructions by a single instruction with the same effect. Because of its myopic view, a peephole optimizer does not have the potential payoff of a full-scale optimizer, but it can significantly improve code at a very local level and can be useful for cleaning up the final code that resulted from more complex optimizations. Much of the work done in peephole optimization can be though of as find-replace activity, looking for certain idiomatic patterns in a single or sequence of two to three Instructions than can be replaced by more efficient alternatives.

For example, MIPS has instructions that can add a small integer constant to the value in a register without loading the constant into a register first, so the sequence on the left can be replaced with that on the right:

```
li $t0, 10
lw $t1, -8($fp)
add $t2, $t1,
$t0sw $t1, -
8($fp)
lw $t1, -8($fp)
addi $t2, $t1,
10sw $t1, -
8($fp)
```

What would you replace the following sequence

```
with?lw $t0, -8($fp)
sw $t0, -8($fp)
What about this
one?mul $t1, $t0,
2
```

Abstract Syntax Tree/DAG : Is nothing but the condensed form of a parse tree and is

- ∑ . Useful for representing language constructs
- ∑ . Depicts the natural hierarchical structure of the source program

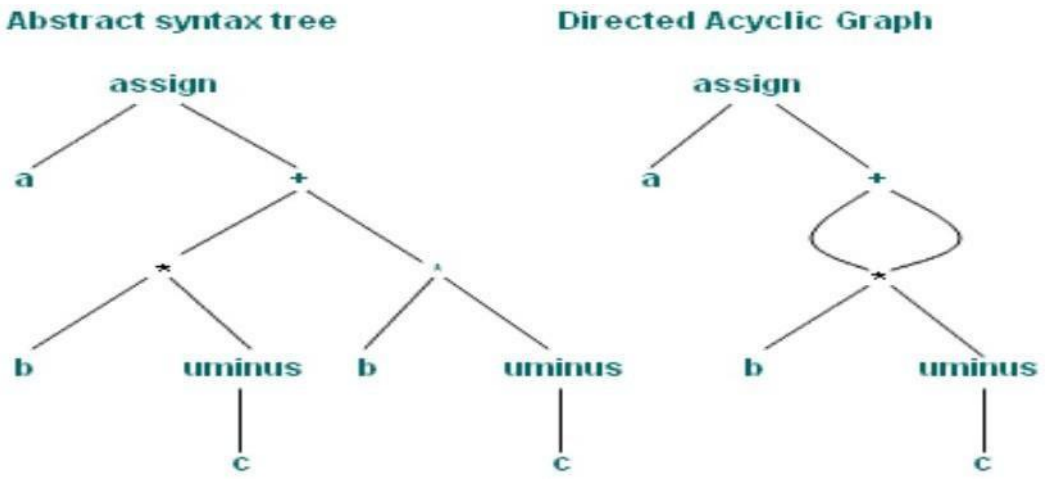
- Each internal node represents an operator
- Children of the nodes represent operands
- Leaf nodes represent operands

.DAG is more compact than abstract syntax tree because common sub expressions are eliminated A syntax tree depicts the natural hierarchical structure of a source program. Its structure has already been discussed in earlier lectures. DAGs are generated as a combination of trees: operands that are being reused are linked together, and nodes may be annotated with variable names (to denote assignments). This way, DAGs are highly compact, since they eliminate local common sub-expressions. On the other hand,

they are not so easy to optimize, since they are more specific tree forms. However, it can be seen that proper building of DAG for a given

sequence of instructions can compactly represent the outcome of the calculation. An example of a syntax tree and DAG has been given in the next slide .

a := b * -c + b * -c



You can see that the node " * " comes only once in the DAG as well as the leaf " b ", but themeaning conveyed by both the representations (AST as well as the DAG) remains the same.

GLOBAL OPTIMIZATIONS, DATA-FLOW ANALYSIS

So far we were only considering making changes within one basic block. With some additional analysis, we can apply similar optimizations across basic blocks, making them global optimizations. It's worth pointing out that global in this case does not mean across the entire program. We usually only optimize one function at a time. Interprocedural analysis is an even larger task, one not even attempted by some compilers. The additional analysis the optimizer must do to perform optimizations across basic blocks is called data-flow analysis. Data-flow analysis is much more complicated than control-flow analysis.

Let's consider a global common sub-expression elimination optimization as our example. Careful analysis across blocks can determine whether an expression is alive on entry to a block. Such an expression is said to be available at that point.

Once the set of available expressions is known, common sub-expressions can be eliminated on a global basis. Each block is a node in the flow graph of a program. The successor set (succ(x)) for a node x is the set of all nodes that x directly flows into. The predecessor set (pred(x)) for a node x is the set of all nodes that flow directly into x. An expression is defined at the point where it is assigned a value and killed when one of its operands is subsequently assigned a new value. An expression is available at some point p in a flow graph if every path leading to p contains a prior definition of that expression which is not
subsequently killed.

avail[B] = set of expressions available on entry to block B
exit[B] = set of expressions available on exit from B
avail[B] = ∩ **exit[x]: x** ∈ **pred[B]** (i.e. B has available the intersection of the exit of its predecessors)

killed[B] = set of the expressions killed in B
defined[B] = set of expressions defined in B
exit[B] = **avail[B]** - **killed[B]** + **defined[B]**

avail[B] = ∩ **(avail[x] - killed[x] + defined[x])** : **x** ∈ **pred[B]**

Here is an algorithm for global common sub-expression elimination:

1) First, compute defined and killed sets for each basic block (this does not involve any of itsredecessors or successors).
2) Iteratively compute the avail and exit sets for each block by running the following algorithmuntil you hit a stable fixed point:
   a) Identify each statement **s** of the form **a = b op c** in some block B such that **b op c** isavailable at the entry to B and neither **b** nor **c** is redefined in B prior to **s**.
   b) Follow flow of control backward in the graph passing back to but not through eachblock that defines **b op c**. The last computation of **b op c** in such a block reaches **s**.
   c) After each computation **d = b op c** identified in step 2a, add statement **t = d** to thatblock where t is a new temp.
   d) Replace **s** by **a = t**.

Lets try an example to make things clearer:main:

```
BeginFunc
28;b = a + 2 ;
c = 4 * b ;
tmp1 = b <
c;
ifNZ tmp1 goto L1
;b = 1 ;
L1:
d = a + 2
;
EndFunc
;
```

First, divide the code above into basic blocks. Now calculate the available expressionsfor each block. Then find an expression available in a block and perform step 2c above.
What common subexpression can you share between the two blocks? What if the abovecode were:

```
main:
BeginFunc
28;b = a + 2 ;
c = 4 * b ;
tmp1 = b < c
;
IfNZ tmp1 Goto L1
;b = 1 ;
z = a + 2 ; <========= an additional line
hereL1:
d = a + 2
;
EndFunc
;
```

Common Sub expression Elimination

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. An expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

```
main()
{
int x, y, z;
x = (1+20)* -x;
y = x*x+(x/y);
y = z = (x/y)/(x*x);
}
straight
translation:tmp1
= 1 + 20 ; tmp2 =
-x ;
```

```
x = tmp1 * tmp2
;tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4 ;
```

```
tmp5 = x / y
;tmp6 = x * x
;
z = tmp5 / tmp6
;y = z ;
```

What sub-expressions can be eliminated? How can valid common sub-expressions (live ones) be determined? Here is an optimized version, after constant folding and propagation and elimination of common sub-expressions:

```
tmp2 = -x ;
x = 21 * tmp2
;tmp3 = x * x ;
tmp4 = x / y ;
y = tmp3 + tmp4
;tmp5 = x / y ;
z = tmp5 / tmp3
;y = z ;
```

Induction Variable Elimination

Constant folding refers to the evaluation at compile-time of expressions whose operands are known to be constant. In its simplest form, it involves determining that all of the operands in an expression are constant-valued, performing the evaluation of the expression at compile-time, and then replacing the expression by its value. If an expression such as **10 + 2 * 3** is encountered, the compiler can compute the result at compile-time (**16**) and emit code as if the input contained the result rather than the original expression. Similarly, constant conditions, such as a conditional branch **if a < b goto L1 else goto L2** where **a** and **b** are constant can be replaced by a **Goto L1** or **Goto L2** depending on the truth of the expression evaluated at compile-time. The constant expression has to be evaluated at least once, but if the compiler does it, it means you don‘t have to do it again as needed during runtime. One thing to be careful about is that the compiler must obey the grammar and semantic rules from the source language that apply to expression evaluation, which may not necessarily match the language you are writing the compiler in. (For example, if you were writing an APL compiler, you would need to take care that you were respecting its Iversonian precedence rules). It should also respect the expected treatment of any exceptional conditions (divide by zero, over/underflow). Consider the Decaf code on the far left and its un optimized TAC translation in the middle, which is then transformed by constant-folding on the far right:

```
a = 10 * 5 + 6 - b; _tmp0 = 10 ;
_tmp1 = 5 ;
_tmp2 = _tmp0 * _tmp1 ;
_tmp3 = 6 ;
_tmp4 = _tmp2 + _tmp3 ;
_tmp5 = _tmp4 –
b;a = _tmp5 ;
_tmp0 = 56 ; _tmp1 = _tmp0 – b ; a = _tmp1 ;
```

Constant-folding is what allows a language to accept constant expressions where a constant isrequired (such as a case label or array size) as in these C language examples:

```
int arr[20 * 4 +
3];switch (i) {
case 10 * 5: ...
}
```

In both snippets shown above, the expression can be resolved to an integer constant at compile time and thus, we have the information needed to generate code. If either expression involved a variable, though, there would be an error. How could you rewrite the grammar to allow the grammar to do constant folding in case statements? This situation is a classic example of the grayarea between syntactic and semantic analysis.

Live Variable Analysis

A variable is live at a certain point in the code if it holds a value that may be needed in thefuture.
Solve backwards:
Find use of a variable This variable is live between statements that have found use as nextstatement Recursive until you find a definition of the variable

Using the sets *use[B]*and *de f[B]*

de f[B] is the set of variables assigned values in *B* prior to any use of that variable in *B use* [*B]*is the set of variables whose values may be used in [*B]* prior to any definition of the variable.

A variable comes live into a block (in *in[B]*), if it is either used before redefinition of it is live coming out of the block and is not redefined in the block .A variable comes live out of a block (in *out[B]*) if and only if it is live coming into one of its successors

$$In[B]=use[B] \cup (out[B]-de\ f[B])$$

$$Out[B]= \bigcup_{S\ succ[B]} in[s]$$

Note the relation between reaching-definitions equations: the roles of *in* and *out* are interchanged

Copy Propagation

This optimization is similar to constant propagation, but generalized to non-constant values. If we have an assignment **a = b** in our instruction stream, we can replace later occurrences of **a** with **b** (assuming there are no changes to either variable in-between). Given theway we generate TAC code, this is a particularly valuable

optimization since it is able to

eliminate a large number of instructions that only serve to copy values from one variable to another. The code on the left makes a copy of **tmp1** in **tmp2** and a copy of **tmp3** in **tmp4**. In theoptimized version on the right, we eliminated those unnecessary copies and propagated the original variable into the later uses:

```
tmp2 = tmp1 ;
tmp3 = tmp2 *
tmp1;tmp4 = tmp3
;
tmp5 = tmp3 * tmp2
;c = tmp5 + tmp4 ;
tmp3 = tmp1 * tmp1
;tmp5 = tmp3 *
tmp1 ;c = tmp5 +
tmp3 ;
```

We can also drive this optimization "backwards", where we can recognize that the original assignment made to a temporary can be eliminated in favor of direct assignment to the final goal:tmp1 = LCall _Binky ;

```
a = tmp1;
tmp2 = LCall _Winky
;b = tmp2 ;
tmp3 = a * b
;c = tmp3 ;
a = LCall
_Binky; b = LCall
_Winky;c = a * b
;
```

CODE GENERATION:

The code generator generates target code for a sequence of three-address statement. It considers each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact, if possible. The code-generation uses descriptors to keep track of register contents and addresses for names.

1. A register descriptor keeps track of what is currently in each register. It is consulted whenever a new register is needed. We assume that initially the register descriptor shows that all registers are empty. (If registers are assigned across blocks, this would not be the case). As the code generation for the block progresses, each register will hold the value of zero or more names atany given time.

2. An address descriptor keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these, since when copied, a value also stays where it was. This information can be stored in the symbol table and is used to determine the accessing method fora name.

CODE GENERATION ALGORITHM :

for each X = Y op Z do

- Invoke a function getreg to determine location L where X must be stored. Usually L is a register.
- Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate

  Mov Y', L

- Generat

  eop Z', L

Again prefer a register for Z. Update address descriptor of X to indicate X is in L. If L is a register update its descriptor to indicate that it contains X and remove X from all other register descriptors.

. If current value of Y and/or Z has no next use and are dead on exit from block and are inregisters, change register descriptor to indicate that they no longer contain Y and/or Z.

The code generation algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x := y op z we perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L will usually be a register, but it could also be a memory location. We shall describe getreg shortly.

2. Consult the address descriptor for u to determine y', (one of) the current location(s) of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of u is not already in L, generate the instruction MOV y', L to place a copy of y in L.

3. Generate the instruction OP z', L where z' is a current location of z. Again, prefer a register to a memory location if z is in both. Update the address descriptor to indicate thatx is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.

4. If the current values of y and/or y have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution ofx := y op z, those registers no longer will contain y and/or z, respectively.

FUNCTION getreg:

1. If Y is in register (that holds no other values) and Y is not live and has no next use after
   X = Y op Z
   then return register of Y for L.
2. Failing (1) return an empty register
3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.
4. Else select memory location X as L

The function **getreg** returns the location L to hold the value of x for the assignment x := y op z.

1. If the name y is in a register that holds the value of no other names (recall that copy instructions such as x := y could cause a register to hold the value of two or more variables

simultaneously), and y is not live and has no next use after execution of x := y op z, then returnthe register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there is one.

3. Failing (2), if x has a next use in the block, or op is an operator such as indexing, that requires a register, find an occupied register R. Store the value of R into memory location (by MOV R, M) if it is not already in the proper memory location M, update the address descriptor M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory.

4. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

Example :

Stmt	code	reg desc	addr desc
t_1 =a-b	mov a,R_0 sub b,R_0	R_0 contains t_1	t_1 in R_0
t_2 =a-c	mov a,R_1 sub c,R_1	R_0 contains t_1 R_1 contains t_2	t_1 in R_0 t_2 in R_1
t_3 =t_1 +t_2	add R_1 ,R_0	R_0contains t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
d=t_3 +t_2	add R_1 ,R_0 mov R_0 ,d	R_0contains d	d in R_0 d in R_0 and memory

For example, the assignment d := (a - b) + (a - c) + (a - c) might be translated into the followingthree- address code sequence:

t_1 = a -

bt $_2$ = a

- c

t_3 = t_1 +

t_2d = t_3 +

t_2

The code generation algorithm that we discussed would produce the code sequence as shown. Shown alongside are the values of the register and address descriptors as code generationprogresses.

DAG for Register allocation:

DAG (Directed Acyclic Graphs) are useful data structures for implementing transformations on basic blocks. A DAG gives a picture of how the value computed by a statement in a basic block is used in subsequent statements of the block. Constructing a DAG from three-address statements is a good way of determining common sub-expressions (expressions computed more than once) within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

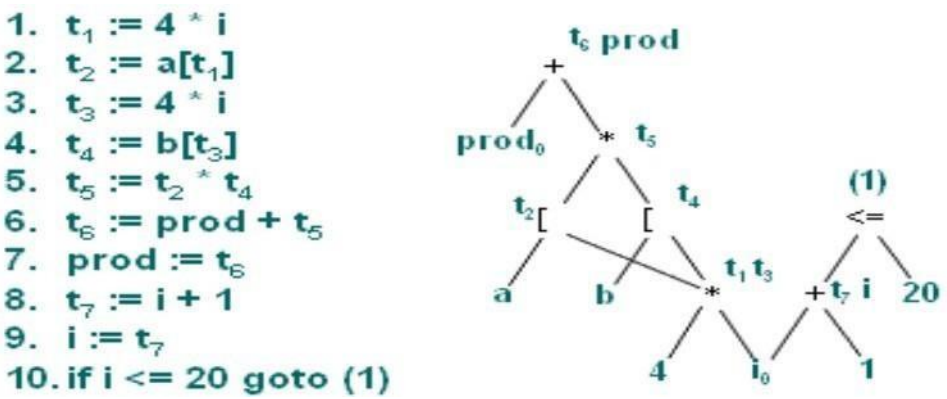
A DAG for a basic block is a directed cyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants. From the operator applied to a name we determine whether the l-value or r-value of a name is needed; most leaves represent r- values. The leaves represent initial values of names, and we subscript them with 0 to avoid confusion with labels denoting "current" values of names as in (3) below.

2. Interior nodes are labeled by an operator symbol.

3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labeling a node are deemed to have that value.

DAG representation Example:



For example, the slide shows a three-address code. The corresponding DAG is shown. We observe that each node of the DAG represents a formula in terms of the leaves, that is, the values possessed by variables and constants upon entering the block. For example, the node labeled t 4 represents the formula

b[4 * i]

that is, the value of the word whose address is 4*i bytes offset from address b, which is theintended value of t 4 .

Code Generation from DAG

S_1= 4 * i	$S_1 = 4 * i$
S_2 = addr(A)-4	S_2 = addr(A)-4
$S_3 = S_2 [S_1]$	$S_3 = S_2 [S_1]$

$S_4 = 4 * i$

S_5 = addr(B)-4	S_5= addr(B)-4
$S_6 = S_5 [S_4]$	$S_6 = S_5 [S_4]$
$S_7 = S_3 * S_6$	$S_7 = S_3 * S_6$

S_8 = prod+S_7

prod = S_8	prod = prod + S_7

S_9 = I+1

I = S_9	I = I + 1
If I <= 20 goto (1)	If I <= 20 goto (1)

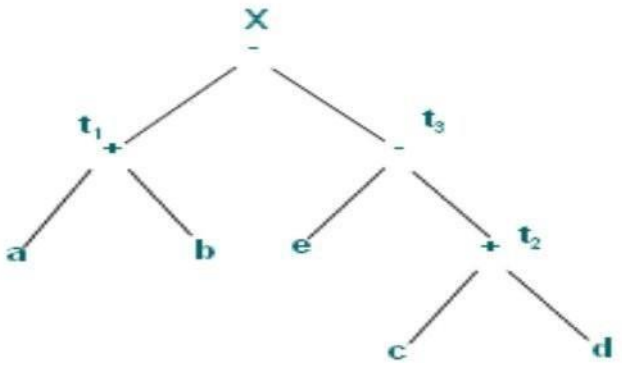
We see how to generate code for a basic block from its DAG representation. The advantage of doing so is that from a DAG we can more easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples. If the DAG is a tree, we can generate code that we can prove is optimalunder such criteria as program length or the fewest number of temporaries used. The algorithm for optimal code generation from a tree is also useful when the intermediate code is a parse tree.

Rearranging order of the code

Consider following basicblock :

t 1 = a + bt
2 = c + d t
3 = e -t 2

X = t 1 -t 3



and its **DAG** given here.

Here, we briefly consider how the order in which computations are done can affect the cost of resulting object code. Consider the basic block and its corresponding DAG representationas shown in the slide.

Rearranging order .

Three adress code for the DAG (assuming only two registers are available)

Rearranging the code

$ast_2 = c + d$

$t_3 = e - t\ 2$

$t_1 = a + b$

MOV a, R_0		$X = t_1 - t_3$
ADD b, R_0		gives
MOV c, R_1		MOV c, R_0
ADD d, R_1		ADD d, R_0
MOV R_0 , t_1	Register spilling	MOV e, R_1
MOV e, R_0		SUB R_0 , R_1
SUB R_1 , R_0		MOV a, R_0
MOV t_1 , R_1	Register reloading	ADD b, R_0
SUB R_0 , R_1		SUB R_1 , R_0
MOV R_1 , X		MOV R_1 , X

If we generate code for the three-address statements using the code generation algorithm described before, we get the code sequence as shown (assuming two registers R0 and R1 are available, and only X is live on exit). On the other hand suppose we rearranged the order of the statements so that the computation of t 1 occurs immediately before that of X as:

t_2 = c +
dt_3 = e -t
$2t_1$ = a +
b X = t_1
$-t_3$

Then, using the code generation algorithm, we get the new code sequence as shown (again only R0 and R1 are available). By performing the computation in this order, we have been able tosave two instructions; MOV R0, t 1 (which stores the value of R0 in memory location t 1) and MOV t 1 , R1 (which reloads the value of t 1 in the register R1).

CODE GENERATION:

The code generator generates target code for a sequence of three-address statement. It considers each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact, if possible. The code-generation uses descriptors to keep track of register contents and addresses for names.

3. A register descriptor keeps track of what is currently in each register. It is consulted

whenever a new register is needed. We assume that initially the register descriptor shows that all registers are empty. (If registers are assigned across blocks, this would not be the case). As the code generation for the block progresses, each register will hold the value of zero or more names atany given time.

4. An address descriptor keeps track of the location (or locations) where the current value of the name can be found at run time. The location might be a register, a stack location, a memory address, or some set of these, since when copied, a value also stays where it was. This information can be stored in the symbol table and is used to determine the accessing method fora name.

CODE GENERATION ALGORITHM :

for each X = Y op Z do

- Invoke a function getreg to determine location L where X must be stored. Usually L is a register.
- Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate

  Mov Y', L

- Generat

  eop Z', L

Again prefer a register for Z. Update address descriptor of X to indicate X is in L. If L is a register update its descriptor to indicate that it contains X and remove X from all other register descriptors.

. If current value of Y and/or Z has no next use and are dead on exit from block and are inregisters, change register descriptor to indicate that they no longer contain Y and/or Z.

The code generation algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x := y op z we perform the following actions:

5. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored. L will usually be a register, but it could also be a memory location. We shall describe getreg shortly.

6. Consult the address descriptor for u to determine y', (one of) the current location(s) of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of u is not already in L, generate the instruction MOV y', L to place a copy of y in L.

7. Generate the instruction OP z', L where z' is a current location of z. Again, prefer a register to a memory location if z is in both. Update the address descriptor to indicate thatx is in location L. If L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptors.

8. If the current values of y and/or y have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution ofx := y op z, those registers no longer will contain y and/or z, respectively.

FUNCTION getreg:

5. If Y is in register (that holds no other values) and Y is not live and has no next use after
   X = Y op Z
   then return register of Y for L.
6. Failing (1) return an empty register
7. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.
8. Else select memory location X as L

The function **getreg** returns the location L to hold the value of x for the assignment x := y op z.

5. If the name y is in a register that holds the value of no other names (recall that copy instructions such as x := y could cause a register to hold the value of two or more variables

simultaneously), and y is not live and has no next use after execution of x := y op z, then returnthe register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

6. Failing (1), return an empty register for L if there is one.

7. Failing (2), if x has a next use in the block, or op is an operator such as indexing, that requires a register, find an occupied register R. Store the value of R into memory location (by MOV R, M) if it is not already in the proper memory location M, update the address descriptor M, and return R. If R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable occupied register might be one whose datum is referenced furthest in the future, or one whose value is also in memory.

8. If x is not used in the block, or no suitable occupied register can be found, select the memory location of x as L.

Example :

Stmt	code	reg desc	addr desc
t_1 =a-b	mov a,R_0 sub b,R_0	R_0 contains t_1	t_1 in R_0
t_2 =a-c	mov a,R_1 sub c,R_1	R_0 contains t_1 R_1 contains t_2	t_1 in R_0 t_2 in R_1
t_3 =t_1 +t_2	add R_1 ,R_0	R_0contains t_3 R_1 contains t_2	t_3 in R_0 t_2 in R_1
d=t_3 +t_2	add R_1 ,R_0 mov R_0 ,d	R_0contains d	d in R_0 d in R_0 and memory

For example, the assignment d := (a - b) + (a - c) + (a - c) might be translated into the followingthree- address code sequence:

t_1 = a -

bt_2 = a

- c

t_3 = t_1 +

t_2d = t_3 +

t_2

The code generation algorithm that we discussed would produce the code sequence as shown. Shown alongside are the values of the register and address descriptors as code generationprogresses.

DAG for Register allocation:

DAG (Directed Acyclic Graphs) are useful data structures for implementing transformations on basic blocks. A DAG gives a picture of how the value computed by a statement in a basic block is used in subsequent statements of the block. Constructing a DAG from three-address statements is a good way of determining common sub-expressions (expressions computed more than once) within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

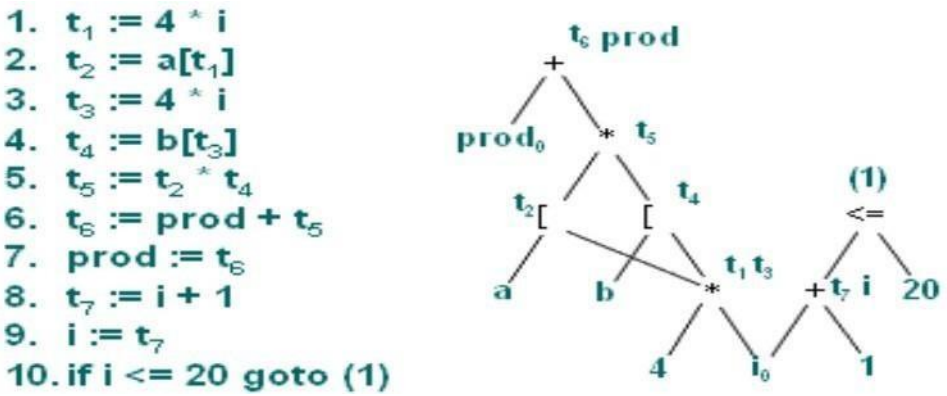
A DAG for a basic block is a directed cyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants. From the operator applied to a name we determine whether the l-value or r-value of a name is needed; most leaves represent r- values. The leaves represent initial values of names, and we subscript them with 0 to avoid confusion with labels denoting "current" values of names as in (3) below.

2. Interior nodes are labeled by an operator symbol.

3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values, and the identifiers labeling a node are deemed to have that value.

DAG representation Example:



For example, the slide shows a three-address code. The corresponding DAG is shown. We observe that each node of the DAG represents a formula in terms of the leaves, that is, the values possessed by variables and constants upon entering the block. For example, the node labeled t 4 represents the formula

b[4 * i]

that is, the value of the word whose address is 4*i bytes offset from address b, which is theintended value of t 4 .

Code Generation from DAG

S_1= 4 * i	$S_1 = 4 * i$
S_2 = addr(A)-4	$S_2 = addr(A)-4$
$S_3 = S_2 [S_1]$	$S_3 = S_2 [S_1]$

$S_4 = 4 * i$

S_5 = addr(B)-4	$S_5 = addr(B)-4$
$S_6 = S_5 [S_4]$	$S_6 = S_5 [S_4]$
$S_7 = S_3 * S_6$	$S_7 = S_3 * S_6$

$S_8 = prod+S_7$

prod = S_8	prod = prod + S_7

$S_9 = I+1$

I = S_9	I = I + 1
If I <= 20 goto (1)	If I <= 20 goto (1)

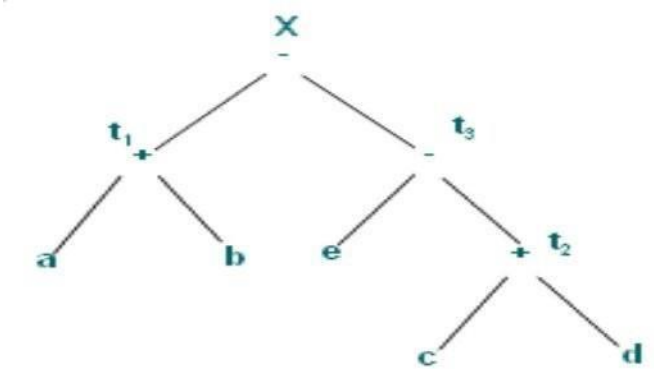
We see how to generate code for a basic block from its DAG representation. The advantage of doing so is that from a DAG we can more easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples. If the DAG is a tree, we can generate code that we can prove is optimalunder such criteria as program length or the fewest number of temporaries used. The algorithm for optimal code generation from a tree is also useful when the intermediate code is a parse tree.

Rearranging order of the code

Consider following basicblock :

t 1 = a + bt
2 = c + d t
3 = e -t 2

X = t 1 -t 3



and its **DAG** given here.

Here, we briefly consider how the order in which computations are done can affect the cost of resulting object code. Consider the basic block and its corresponding DAG representationas shown in the slide.

Rearranging order .

Three adress code for the DAG (assuming only two registers are available)

Rearranging the code

$ast_2 = c + d$

$t_3 = e - t\ 2$

$t_1 = a + b$

MOV a, R_0		$X = t_1 - t_3$
ADD b, R_0		gives
MOV c, R_1		MOV c, R_0
ADD d, R_1		ADD d, R_0
MOV R_0 , t_1	Register spilling	MOV e, R_1
MOV e, R_0		SUB R_0 , R_1
SUB R_1 , R_0		MOV a, R_0
MOV t_1 , R_1	Register reloading	ADD b, R_0
SUB R_0 , R_1		SUB R_1 , R_0
MOV R_1 , X		MOV R_1 , X

If we generate code for the three-address statements using the code generation algorithm described before, we get the code sequence as shown (assuming two registers R0 and R1 are available, and only X is live on exit). On the other hand suppose we rearranged the order of the statements so that the computation of t 1 occurs immediately before that of X as:

$t_2 = c + d$
$t_3 = e - t\ 2$
$t_1 = a + b$
$X = t_1 - t_3$

Then, using the code generation algorithm, we get the new code sequence as shown (again only R0 and R1 are available). By performing the computation in this order, we have been able to save two instructions; MOV R0, t 1 (which stores the value of R0 in memory location t 1) and MOV t 1 , R1 (which reloads the value of t 1 in the register R1).