MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

Autonomous Institution – UGC, Govt. of India

Department of COMPUTATIONAL INTELLIGENCE

B.TECH (AI&DS)

B.TECH(R-20 Regulation)

(III YEAR – I SEM)

2023-24

AUTOMATA AND COMPILER DESIGN

(R20A1202)

LECTURE NOTES

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12(B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE-Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad–500100, Telangana State, India

Department of COMPUTATIONAL INTELLIGENCE

ARTIFICIAL INTELLIGENCE

&

DATA SCIENCE

AUTOMATA AND COMPILER DESIGN

(R20A1202)

LECTURE NOTES

Prepared by

D CHANDRA SEKHAR REDDY,
ASSOCIATE PROFESSOR

Department of Computational Intelligence

Artificial Intelligence and Data Science

Vision

To be a premier centre for academic excellence and research through innovative interdisciplinary collaborations and making significant contributions to the community, organizations, and society as a whole.

Mission

- To impart cutting-edge Artificial Intelligence technology in accordance with industry norms.
- To instill in students a desire to conduct research in order to tackle challenging technical problems for industry.
- To develop effective graduates who are responsible for their professional growth, leadership qualities and are committed to lifelong learning.

QUALITY POLICY

- To provide sophisticated technical infrastructure and to inspire students to reach their full potential.
- To provide students with a solid academic and research environment for a comprehensive learning experience.
- To provide research development, consulting, testing, and customized training to satisfy specific industrial demands, thereby encouraging self-employment and entrepreneurship among students.

For more information: www.mrcet.ac.in

SYLLABUS

III Year B.Tech. AI&DS - I Sem L/T/P/C 3/-/-/3

(R20A1202) AUTOMATA & COMPILER DESIGN

COURSE OBJECTIVES: -

- To provide an understanding of automata, grammars, language translators.
- To know the various techniques used in compiler construction.
- To be aware of the process of semantic analysis.
- To analyze the code optimization & code generation techniques.

UNIT - I:

Formal Language and Regular Expressions: Languages, Definition Languages regular expressions, Finite Automata – DFA, NFA. Conversion of regular expression to NFA, NFA to DFA. Context Free grammars and parsing, derivation, parse trees, Application of Finite Automata.

UNIT - II:

Introduction To Compiler, Phases of Compilation, ambiguity LL(K) grammars and LL(1) parsing Bottom up parsing handle pruning LR Grammar Parsing, LALR parsing, parsing ambiguous grammars, YACC programming specification.

Semantics: Syntax directed translation, S-attributed and L-attributed grammars,

UNIT - III: Intermediate code – abstract syntax tree, translation of simple statements and control flow statements. **Context Sensitive features** – Chomsky hierarchy of languages and recognizers, type checking, type conversions, equivalence of type expressions, overloading of functions and operations.

UNIT - IV:

Run time storage: Storage organization, storage allocation strategies scope access to nonlocal names, **Code optimization**: Principal sources of optimization, optimization of basic blocks, peephole optimization,

UNIT - V:

Code generation: Machine dependent code generation, object code forms, generic code generation algorithm, Register allocation and assignment. Using DAG representation of Block.

TEXT BOOKS:

1. Introduction to Theory of computation. Sipser, 2nd Edition, Thomson.
2. Compilers Principles, Techniques and Tools Aho, Ullman, Ravisethi, Pearson Education.

REFERENCES:

1. Modern Compiler Construction in C , Andrew W.Appel Cambridge University Press.
2. Compiler Construction, LOUDEN, Thomson.
3. Elements of Compiler Design, A. Meduna, Auerbach Publications, Taylor and Francis Group.
4. Principles of Compiler Design, V. Raghavan, TMH.
5. Engineering a Compiler, K. D. Cooper, L. Torczon, ELSEVIER.
6. Introduction to Formal Languages and Automata Theory and Computation - Kamala Krithivasan and Rama R, Pearson.
7. Modern Compiler Design, D. Grune and others, Wiley-India.
8. A Text book on Automata Theory, S. F. B. Nasir, P. K. Srimani, Cambridge Univ. Press.
9. Automata and Language, A. Meduna, Springer.

COURSE OUTCOMES:

- Understand the necessity and types of different language translators in use.
- Apply the techniques and design different components (phases) of a compiler.
- Ability to implement practical aspects of automata theory.
- Use the tools Lex, Yacc in compiler construction

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
DEPARTMENT OF COMPUTATIONAL INTELLIGENCE

INDEX

UNIT -1

Fundamentals

Symbol – An atomic unit, such as a digit, character, lower-case letter, etc. Sometimesa word.*[Formal language does not deal with the "meaning" of thesymbols.]*

Alphabet – A finite set of symbols, usually denoted byΣ.
Σ ={0, 1}
Σ = {0, a,9, 4}
Σ = {a, b, c,d}

String – A finite length sequence of symbols, presumably from some alphabet. w=0110
y=0aa
x=aabcaa
z = 111

Special string: ε (also denoted by λ)

Concatenation:	wz = 0110111		
Length:	$\|w\| = 4$	$\|\varepsilon\| = 0$	$\|x\| = 6$
Reversal:	$y^R = aa0$		

Some special sets of strings:

Σ^* All strings of symbols fromΣ

Σ^+ $\Sigma^* - \{\varepsilon\}$

Example: $\Sigma = \{0,1\}$

$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001,\ldots\}$

$\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001,\ldots\}$

A **language**is:
A set of strings from some alphabet (finite or infinite). In otherwords,
Any subset L ofΣ^*

Some speciallanguages:
{}The empty set/language, containing nostring.
{ε}A language containing one string, the emptystring.

Examples:
$\Sigma = \{0,1\}$

L = {x | x is in Σ^* and x contains an even number of 0"s}

$\Sigma = \{0, 1, 2,\ldots, 9, .\}$

= {0, 1.5, 9.326,…}

Σ = {a, b, c,…, z, A, B,…, Z}

L = {x | x is in Σ^* and x is a Pascal reserved word}

= {BEGIN, END, IF,…}

Σ = {Pascal reserved words} U { (,), ., :, ;,…} U {Legal Pascal identifiers} L = {x | x is in Σ^* and x is a syntactically correct Pascal program}

Σ = {English words}
L = {x | x is in Σ^* and x is a syntactically correct English sentence}

Regular Expression

- A regular expression is used to specify a language, and it does so precisely.
- Regular expressions are very intuitive.
- Regular expressions are very useful in a variety ofcontexts.
- Given a regular expression, an NFA-ε can be constructed from it automatically.
- Thus, so can an NFA, a DFA, and a corresponding program, all automatically.

Definition:

Let Σ be an alphabet. The regular expressions over Σare:
Ø Represents the empty set {}
E Represents the set{ε}
Represents the set {a}, for any symbol a inΣ
Let r and s be regular expressions that represent the sets R and S, respectively.
r+sRepresents the set RUS (precedence3)
rsRepresents thesetRS (precedence2)
r^* Represents thesetR* (highest precedence)
(r) Represents thesetR (not an op, providesprecedence)
If r is a regular expression, then L(r) is used to denote the correspondinglanguage.

Examples:

Let Σ = {0,1}
(0 +1)* All strings of0‟s and1‟s0(0 +1)* Allstrings of 0‟s and 1‟s, beginning with a0(0 +1)*1 All strings of 0‟s and 1‟s, ending with a1
(0 + 1)*0(0+1)* All strings of 0‟s and 1‟s containing at least one 0 (0 + 1)*0(0 + 1)*0(0+1)* All strings of 0‟s and 1‟s containing at least two
0‟s (0+1)*01*01* All strings of 0‟s and 1‟s containing at least two
0‟s (101*0)* All strings of 0‟s and 1‟s containing an even number of 0‟s
1*(01*01*)* All strings of 0‟s and 1‟s containing an even number of 0‟s
(1*01*0)*1* Allstrings of0‟s and 1‟s containing an even number of0‟s

Identities:

1. Øu = uØ=Ø Multiply by0
2. εu = uε=u Multiply by1
3. Ø* =ε
4. ε* =ε
5. u+v =v+u

6. u + Ø =u

7. u + u = u

8. u* =(u*)*

9. u(v+w) =uv+uw

10. (u+v)w =uw+vw

11. (uv)*u = u(vu)*

12. (u+v)* = (u*+v)*

=u*(u+v)*

=(u+vu*)*
= (u*v*)*

=u*(vu*)*

=(u*v)*u* Finite

State Machines

A finite state machine has a set of states and two functions called the next-state function and the output function

The set of states correspond to all the possible combinations of the internal storage

If there are n bits of storage, there are 2^n possible states

The next state function is a combinational logic function that given the inputs and the current state, determines the next state of the system

The output function produces a set of outputs from the current state and the inputs

- There are two types of finite statemachines
- In a Moore machine, the output only depends on the current state
- While in a Mealy machine, the output depends both the current state and the current input
- We are only going to deal with the Mooremachine.
- These two types are equivalent incapabilities

A Finite State Machine consistsof:

Kstates:S = {s_1, s_2, … ,s_k}, s_1 is initial state Ninputs:I = {i_1, i_2, …,i_n}

Moutputs:O = {o_1, o_2, …,o_m}

Next-state function T(S, I) mapping each current state and input to next state Output Function P(S) specifies output

Finite Automata

- Two types – both describe what are called regular languages
  - Deterministic (DFA) – There is a fixed number of states and we can only be in one state at a time

- Nondeterministic (NFA) –There is a fixed number of states but it can be in multiple states at one time

- While NFA‟s are more expressive than DFA‟s, we will see that adding nondeterminism does not let us define any language that cannot be defined by a DFA.

- One way to think of this is we might write a program using a NFA, but then when it is "compiled" we turn the NFA into an equivalent DFA.

Formal Definition of a Finite Automaton

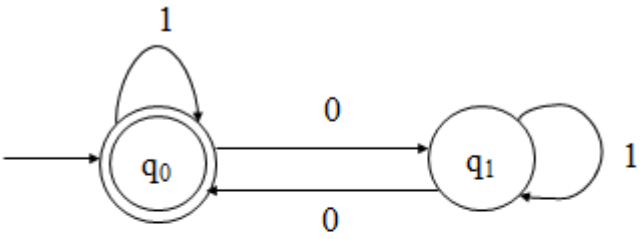
- Finite set of states, typically Q.
- Alphabet of input symbols, typically∑
- One state is the start/initial state, typically q0 // q0 ∈ Q
- Zero or more final/accepting states; the set is typically F. // F⊆Q
- A transition function, typicallyδ. Thisfunction takes a state and input symbol as arguments.

Deterministic Finite Automata (DFA)

- A DFA is a five-tuple: M = (Q, Σ, δ, q0, F)
  Q=A finite set of states
  Σ=A finite input alphabet
  q0=The initial/starting state, q0 is in Q
  F=A set of final/accepting states, which is a subset of Q
  Δ=A transition function, which is a total function from Q x Σ to Q
  δ: (Q x Σ)–>Q δ is defined for any q in Q and s in Σ, and δ(q,s)=q‟is equal to another state q‟ in Q.
  Intuitively, δ(q,s) is the state entered by M after reading symbol s while in state q.

- For Example #1:

$Q = \{q_0, q_1\}$
$\Sigma = \{0, 1\}$
Start state is q_0
$F = \{q_0\}$



δ:

	0	1
q_0	q_1	q_0
q_1	q_0	q_1

- LetM=(Q,Σ,δ,q,F) be a DFA and let w be in Σ*.Then w is *accepted* by M iff

$\delta(q_0, w) = p$ for some state p in F.

- Let M = (Q, Σ, δ, q_0, F) be a DFA. Then the *language accepted* by M is theset:

L(M) = {w | w is in Σ* and $\delta(q_0, w)$ is in F}

- Another equivalent definition:

L(M) = {w | w is in Σ* and w is accepted by M}

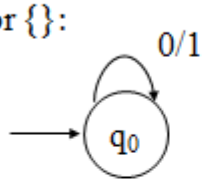
- Let L be a language. Then L is a *regular language* iff there exists a DFA M such that L =L(M).

Notes:

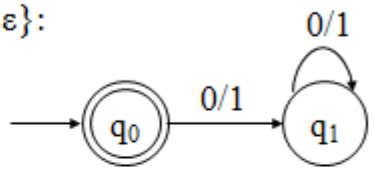
- A DFA M = (Q, Σ, δ,q0,F) partitions the set Σ* into two sets: L(M)and Σ* -L(M).
- If L = L(M) then L is a subset of L(M) and L(M) is a subset of L.
- Similarly, if L(M1) = L(M2) then L(M1) is a subset of L(M2) and L(M2) is a subset of L(M1).
- Some languages are regular, others are not. For example,if L1 = {x | x is a string of 0's and 1's containing an even number of 1's} and L2 = {x | x = 0^n1^n for some n >= 0}then L1 is regular but L2 is not.

- Let Σ = {0, 1}. Give DFAs for {}, {ε}, Σ^*, and Σ^+.

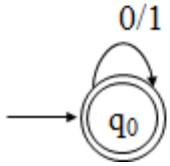
For {}:



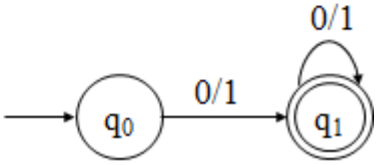
For {ε}:



For Σ^*:



For Σ^+:



Nondeterministic Finite Automata (NFA)

An NFA is a five-tuple: $M = (Q, \Sigma, \delta, q_0, F)$

Q	A finite set of states
Σ	A finite input alphabet
q0	The initial/starting state, q0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A transition function, which is a total function from Q x Σ to 2^Q

δ: (Q x Σ)->$\mathbf{2^Q}$ -2^Q is the power set of Q, the set of all subsets of Q δ(q,s) -The set of all states p such that there is a transition

labeled s from q to p δ(q,s) is a function from Q x S to 2^Q (but not to Q)
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let w be in Σ*. Then w is *accepted* by M iff $\delta(\{q_0\}, w)$ contains at least one state inF.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an NFA. Then the *language accepted* by M is the set: $L(M) = \{w \mid w \text{ is in } \Sigma^* \text{ and } \delta(\{q_0\}, w) \text{ contains at least one state in } F\}$
Another equivalent definition:
L(M) = {w | w is in Σ* and w is accepted by M}

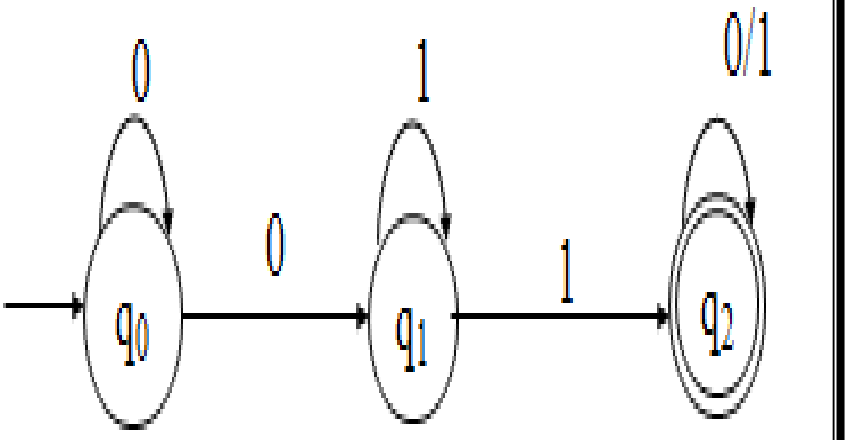
- Example: some 0's followed by some 1's

$Q = \{q_0, q_1, q_2\}$

$\Sigma = \{0, 1\}$

Start state is q_0

$F = \{q_2\}$



δ:

	0	1
q_0	$\{q_0, q_1\}$	$\{\}$
q_1	$\{\}$	$\{q_1, q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

$= \{\epsilon, 0, 00, 1, 11, 111, 01, 10, \ldots\ldots\}$

$= \{ \epsilon$, any combination of 0's, any combination of 1's, any combination of 0 and 1 $\}$

Hence, L. H. S. = R. H. S. is proved.

3.4 RELATIONSHIP BETWEEN FA AND RE

There is a close relationship between a finite automata and the regular expression we can show this relation in below figure.

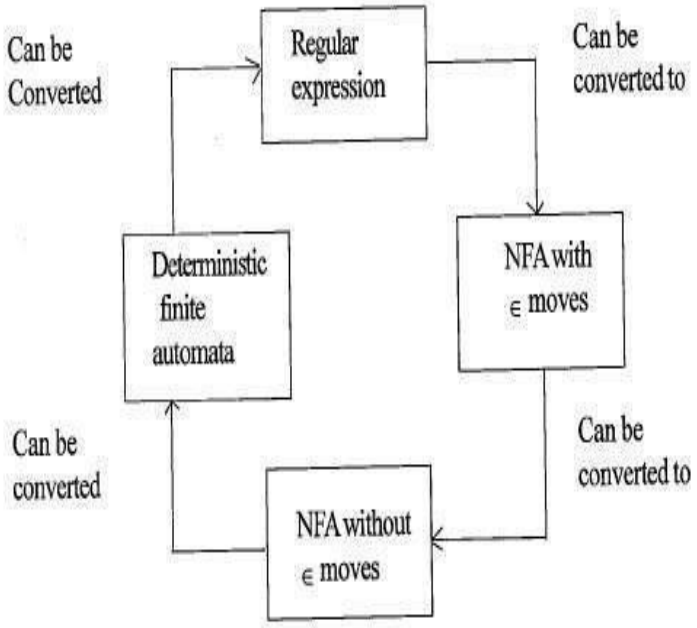


FIGURE : Relationship between FA and regular expression

The above figure shows that it is convenient to convert the regular expression to NFA with ϵ moves. Let us see the theorem based on this conversion.

3.5 CONSTRUCTING FA FOR A GIVEN REs

Theorem : If r be a regular expression then there exists a NFA with ϵ - moves, which accepts $L(r)$.

Proof : First we will discuss the construction of NFA M with ϵ - moves for regular expression r and then we prove that $L(M) = L(r)$.

Let r be the regular expression over the alphabet Σ.

Construction of NFA with ϵ - moves

Case 1 :

(i) $r = \phi$

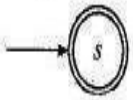
NFA $M = (\{s, f\}, \{\}, \delta, s, \{f\})$ as shown in Figure1 (a)

(No path from initial state s to reach the final state f.)

Figure 1 (a)

(ii) $r = \in$

NFA $M = (\{s\}, \{\}, \delta, s, \{s\})$ as shown in Figure 1 (b)

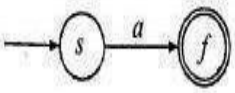


(The initial state s is the final state)

Figure 1 (b)

(iii) $r = a$, for all $a \in \Sigma$,

NFA $M = (\{s, f\}, \Sigma, \delta, s, \{f\})$



(One path is there from initial state s to reach the final state f with label a.)

Figure 1 (c)

Case 2 : $|r| \geq 1$

Let r_1 and r_2 be the two regular expressions over Σ_1, Σ_2 and N_1 and N_2 are two NFA for r_1 and r_2 respectively as shown in Figure 2 (a).

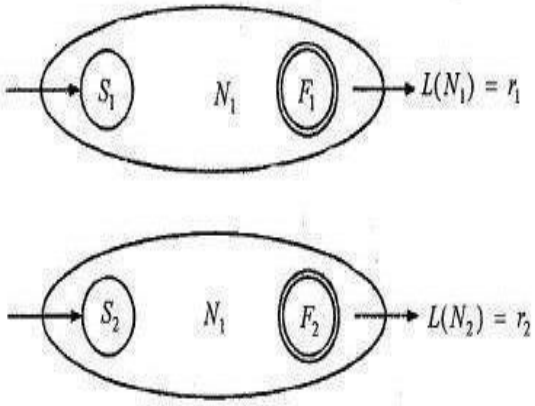
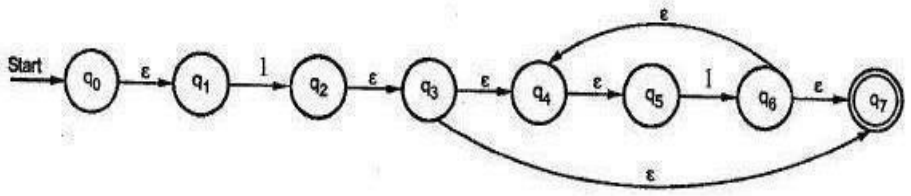


Figure 2 (a) NFA for regular expression r_1 and r_2

The final NFA is



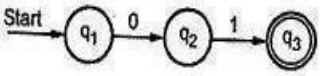
Example 4 : Construct NFA for the r. e. $(01+2^*)0$.

Solution : Let us design NFA for the regular expression by dividing the expression into smaller units

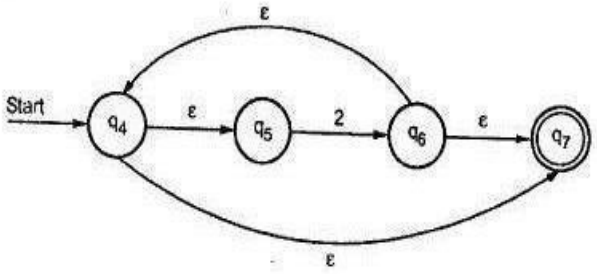
$$r = (r_1 + r_2)r_3$$

where $r_1 = 01$, $r_2 = 2^*$ and $r_3 = 0$

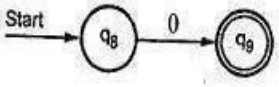
The NFA for r_1 will be



The NFA for r_2 will be



The NFA for r_3 will be



Conversion from NFA to DFA

Suppose there is an NFA N < Q, ∑, q0, δ, F > which recognizes a language L. Then the DFA D < Q‟, ∑, q0, δ‟, F‟ > can be constructed for language L as:

Step 1: Initially Q‟ = ϕ.

Step 2: Add q0 to Q‟.

Step 3: For each state in Q‟, find the possible set of states for each input symbol using transition function of NFA. If this set of states is not in Q‟, add it to Q‟.

Step 4: Final state of DFA will be all states with contain F (final states of NFA)

Example
Consider the following NFA shown in Figure 1.

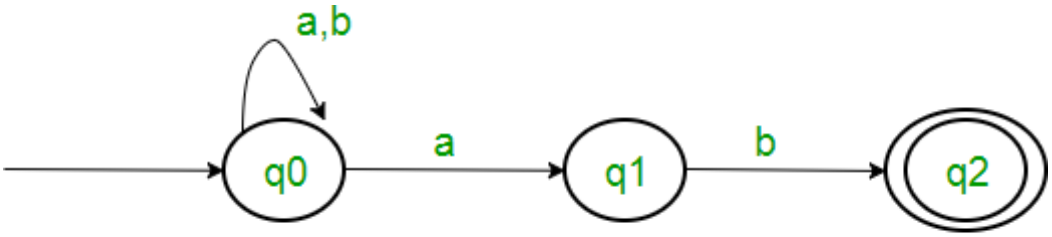


Figure 1

Following are the various parameters for NFA.
Q = { q0, q1, q2 }
∑ = (a, b)
F = { q2 }
δ (Transition Function of NFA)

State	a	b
q0	q0,q1	q0
q1		q2
q2		

Step 1: Q‟ = ϕ Step
2: Q‟ = {q0}
Step 3: For each state in Q‟, find the states for each input symbol.
Currently, state in Q‟ is q0, find moves from q0 on input symbol a and b using transition function of NFA and update the transition table of DFA

δ‟ (Transition Function of DFA)

State	a	b
q0	{q0,q1}	q0

Now { q0, q1 } will be considered as a single state. As its entry is not in Q‟, add it to Q‟. So
Q‟ = { q0, { q0, q1 } }

Now, moves from state { q0, q1 } on different input symbols are not present in transition table of DFA, we will calculate it like:
δ‟ ({ q0, q1 }, a) = δ (q0, a) U δ (q1, a) = { q0, q1 }
δ‟ ({ q0, q1 }, b) = δ (q0, b) U δ (q1, b) = { q0, q2 }
Now we will update the transition table of DFA.

δ‟ (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}

Now { q0, q2 } will be considered as a single state. As its entry is not in Q‟, add it to Q‟. So
Q‟ = { q0, { q0, q1 }, { q0, q2 } }

Now, moves from state {q0, q2} on different input symbols are not present in transition table of DFA, we will calculate it like:
δ‟ ({ q0, q2 }, a) = δ (q0, a) U δ (q2, a) = { q0, q1 }
δ‟ ({ q0, q2 }, b) = δ (q0, b) U δ (q2, b) = { q0 } Now
we will update the transition table of DFA.

δ‟ (Transition Function of DFA)

State	a	B
q0	{q0,q1}	q0
{q0,q1}	{q0,q1}	{q0,q2}
{q0,q2}	{q0,q1}	q0

As there is no new state generated, we are done with the conversion. Final state of DFA will be state which has q2 as its component i.e., { q0, q2 }

Following are the various parameters for DFA.
Q‟ = { q0, { q0, q1 }, { q0, q2 } }
∑ = (a, b)
F = { { q0, q2 } } and transition function δ‟ as shown above. The final DFA for above NFA has been shown in Figure 2.

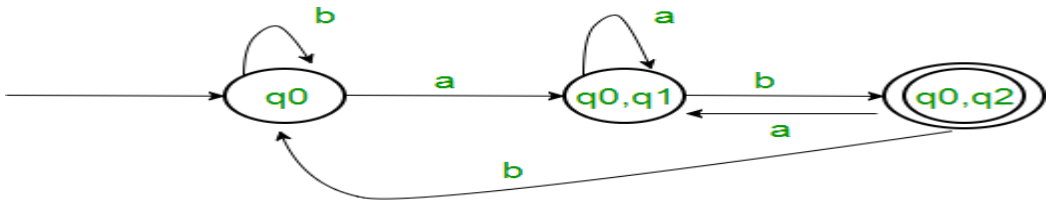


Figure 2

Note : Sometimes, it is not easy to convert regular expression to DFA. First you can convert regular expression to NFA and then NFA to DFA

Application of Finite_state_machine_and regular expression in Lexical analysis: Lexical analysis is the process of reading the source text of a program and converting that source code into a sequence of tokens. The approach of design a finite state machine by using regular expression is so useful to generates token form a given source text program. Since the lexical structure of more or less every programming language can be specified by a regular language, a common way to implement a lexical analysis is to; 1. Specify regular expressions for all of the kinds of tokens in the language. The disjunction of all of the regular expressions thus describes any possible token in the language. 2. Convert the overall regular expression specifying all possible tokens into a deterministic finite automaton (DFA). 3. Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer. To recognize identifiers, numerals, operators, etc., implement a DFA in code. State is an integer variable, δ is a switch statement Upon recognizing a lexeme returns its lexeme, lexical class and restart DFA with next character in source code.

CONTEXT FREE-GRAMMAR

Definition: Context-Free Grammar (CFG) has 4-tuple: G = (V, T, P,S)

Where,

V -A finite set of variables or *non-terminals*
T -A finite set of *terminals* (V and T do not intersect)
P -A finite set of *productions*, each of the form A –>α,
Where A is in V and α is in (V U T)*
Note: that α may be ε.
S -A starting non-terminal (S is inV)

Example :CFG:

G = ({S}, {0, 1}, P, S) P:
S–>0S1or just simply S –> 0S1 |ε S –>ε
Example Derivations:

S	=> 0S1	(1)
S	=> ε	(2)
	=> 01	(2)
S	=> 0S1	(1)
	=> 00S11	(1)
	=> 000S111	(1)
	=> 000111	(2)

- Note that G "generates" the language $\{0^k1^k \mid k>=0\}$

Derivation (or Parse) Tree

- **Definition:** Let G = (V, T, P, S) be a CFG. A tree is a derivation (or parse) tree if:
  - Every vertex has a label from V U T U{ε}
  - The label of the root is S
  - If a vertex with label A has children with labels X1, X2,…, Xn, from left to right, then

synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

Terminals **A –> X1, X2,…, Xn must be a production in P**

The first L stands for "Left-to-right scan of input". The second L stands for "Left-most derivation". The „1"

stands for "1 token of look ahead".

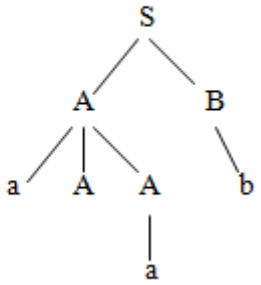
No LL (1) grammar can be ambiguous or left recursive.

LL (1) Grammar:

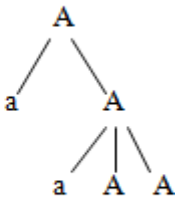
- If a vertex has label ε, then that vertex is a leaf and the only child of its parent
- More Generally, a derivation tree can be defined with any non-terminal as the root.

- **Example:**

S –> AB
A –> aAA
A –> aA
A –> a
B –> bB
B –> b



yield = aAab



yield = aaAA

Notes:

- Root can be any non-terminal
- Leaf nodes can be terminals or non-terminals

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

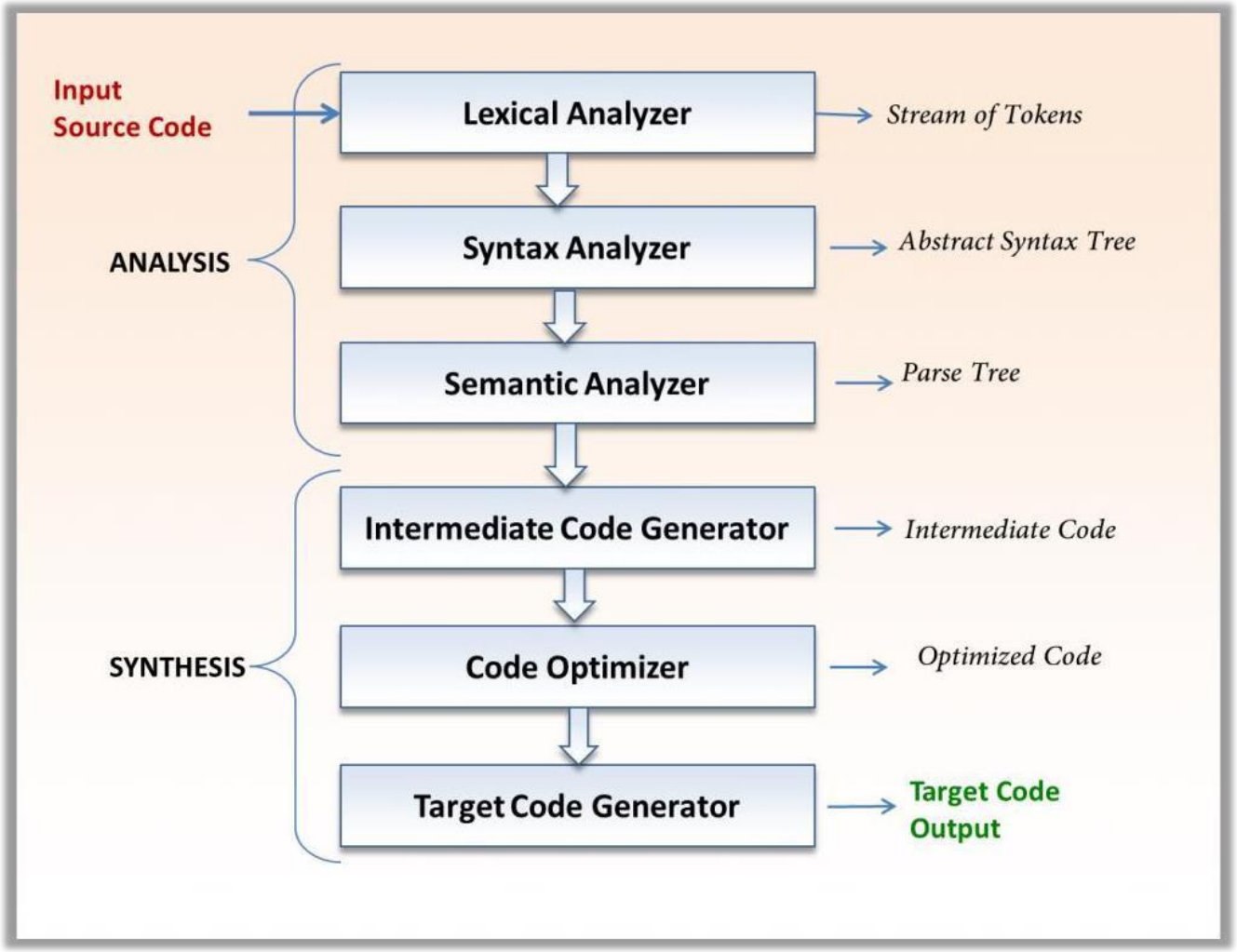
The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of

- A derivation tree with root S shows the productions used to obtain a sentential form.

Phases of a Compilation



LL(k)

LL(k) grammar performs a top-down, leftmost parse after reading the string from left-to-right
Here, kk is the number of look-aheads allowed.
With the knowledge of kk look-aheads, we
calculate FIRSTkFIRSTk and FOLLOWkFOLLOWk where:

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

- FIRSTkFIRSTk: kk terminals that can be at the beginning of a derived non-terminal
- FOLLOWkFOLLOWk: kk terminals that can come *after* a derived non-terminal

The basic idea is to create a lookup table using this information from which the parser can then simply go and check what derivation is to be made given a certain input token.

Now, the following text from here explains strong LL(k)LL(k):
In the general case, the LL(k)LL(k) grammars are quite difficult to parse directly. This is due to the fact that the left context of the parse must be remembered somehow.
Each parsing decision is based both on what is to come as well as on what has already

been seen of the input.

The class of LL(1)LL(1) grammars are so easily parsed because it is strong. The strong LL(kLL(k) grammars are a subset of the LL(k)LL(k) grammars that can be parsed *without* knowledge of the left-context of the parse. That is, each parsing decision is based only on the next k tokens of the input for the current nonterminal that is being expanded.
Formally,

A grammar (G=N,T,P,S)(G=N,T,P,S) is strong if for any two distinct A-productions in the grammar:
A→αA→α
A→βA→β
FIRSTk(αFOLLOWk(A))∩FIRSTk(βFOLLOWk(A))=∅ FIRSTk(αFOLLOWk(A))∩FIRSTk(βFOL LOWk(A))=∅
That looks complicated so we‟ll see it another way. Let‟s take a textbook example to understand, instead, when is some grammar "weak" or when exactly would we need to know the left -context of the parse.

S→aAaS→aAa
S→bAbaS→bAba
A→bA→b
A→ϵA→ϵ
Here, you‟ll notice that for an LL(2)LL(2) instance, baba could result from either of the SSproductions. So the parser needs some left-context to decide whether baba is produced by S→aAaS→aAa or S→bAbaS→bAba.
Such a grammar is therefore "weak" as opposed to being a strong LL(k)LL(k) grammar.

BOTTOM UPPARSING:

Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

(The point of parsing is to construct a derivation. A derivation consists of a series of rewrite steps)

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow - - - \Rightarrow r_{n-}$

$\leftarrow$

$_1 \Rightarrow r_n \Rightarrow$ sentence Bottom-up

Assuming the production $A \rightarrow \beta$, to reduce r_i r_{i-1} match some RHS β against r_i then replace β with its corresponding LHS, A.In terms of the parse tree, this is working from leaves to root.

Example – 1:

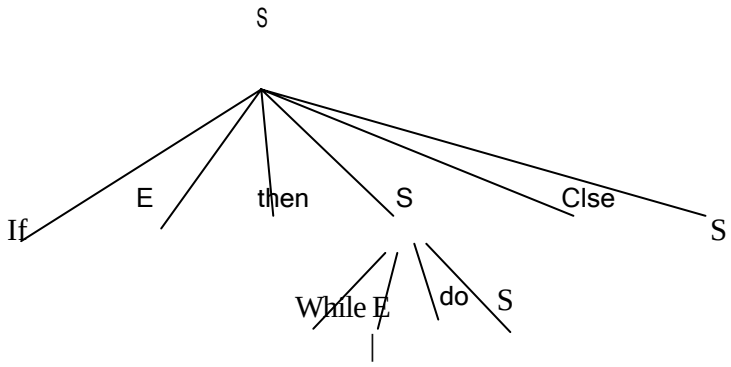
S→if E then S else S/while E do S/

print E→ true/ False/id

Input: if id then while true do print else print.

Parse tree:

Basicidea: Given input string a, "reduce" it to the goal (start) symbol, by looking for substring that match production RHS.



$\Rightarrow$ **if E then S elseS**
lm

$\Rightarrow$ **if id then S elseS**
lm

$\Rightarrow$ **if id then while E do S elseS**
lm

$\Rightarrow$ **if id then while true do S elseS**
lm

$\Rightarrow$ **if id then while true do print elseS**
lm

$\Rightarrow_{lm}$ **if id then while true do print elseprint**

$\Leftarrow_{rm}$ **if E then while true do print elseprint**

$\Leftarrow_{rm}$ **if E then while E do print elseprint**

$\Leftarrow_{rm}$ **if E then while E do S elseprint**

$\Leftarrow_{rm}$ **if E then S elseprint**

$\Leftarrow_{rm}$ **if E then S elseS**

$\Leftarrow_{rm}$ **S**

HANDLE PRUNING:

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example:

E→E+E/E*E/(E)/id

Right-sentential form	Handle	Reducing production
a+b*c	A	E→id
E+b*c	B	E→id

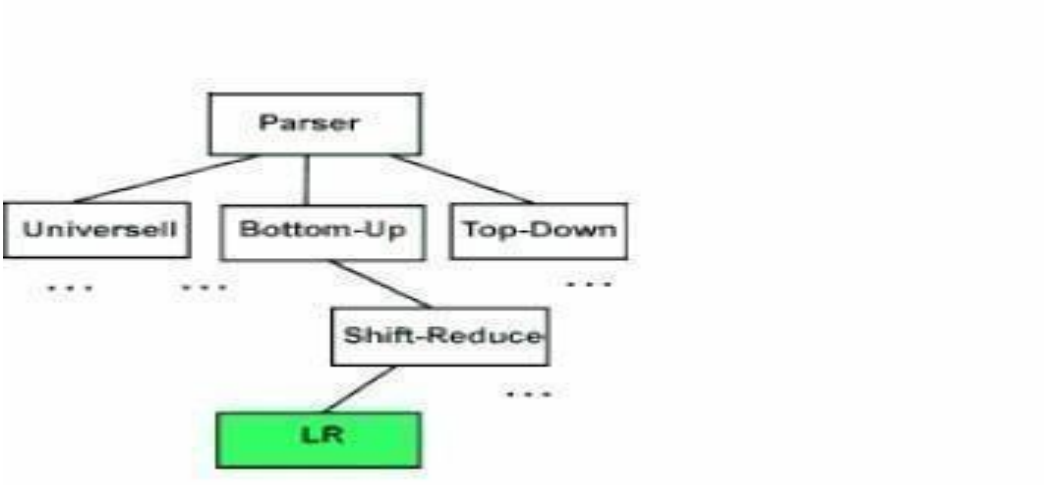
E+E*C	C	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step. We can use parser-generators that compute the handles for us

LR PARSINGINTRODUCTION:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a

right most derivation in reverse.



WHY LR-PARSING:

1. LRparsers can be constructed to recognize virtually all programming-language constructs for which context-free grammar scan be written.

2. TheLRparsing method is the most general non-backtracking shift- reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.

3. The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.

4,AnLR parser can detect a syntactic error as soon as it is possible to do so on a left- to-right scan of theinput.

The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

LR-PARSERS:

LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are k=0 and k=1. LR(1) is of practical relevance.

„L'stands for "Left-to-right" scan of input.

„R" stands for "Rightmost derivation (in reverse)".

K'standsfornumber of input symbols of look-a-head that are used in making parsing decisions.When (K) is omitted, „K'is assumed to be 1.

LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar.

A grammar is LR(1) if, given a right-most derivation

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 - - - r_{n-1} \Rightarrow r_n \Rightarrow$ sentence.

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i.

Parser accepts input when stack contains only the start symbol and no remaining input symbol areleft.

LR(0)item:(no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.

Ex– 1: $S^{I} \rightarrow .S\$$

$S \rightarrow .$

x $S \rightarrow .(L)$

Ex-2: $A \rightarrow XYZ$ generates 4LR(0) items

$A \rightarrow .XYZ$

$A \rightarrow X.$

YZ $A \rightarrow XY.$

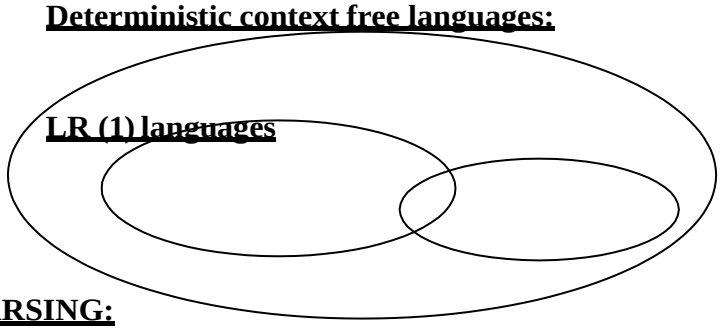
Z $A \rightarrow XYZ.$

$A \rightarrow XY.Z$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z.

→ **LR(0) items play a key role in the SLR(1) table construction algorithm.**

→ **LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms. LR parsers have more information available than LL parsers when choosing a production:**

- LR knows everything derived from RHS plus,K 'lookahead symbols.
- **LL just knows„K'lookahead symbols into whatsderived from RHS.**
- **Deterministic context free languages:**
- **LR (1) languages**



LALR PARSING:

Example:

Construct C={I0,I1, ... ,In} The collection of sets of LR(1)items

For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (clus them into a single term)

I_0 →same asprevious
I_1 → “
I_2 → “
I_{36} – Clubbing item I3 and I6 into one I36 item.
C →cC,c/d/$
C→cC,c/d/$
C→d,c/d/$
I_5→some as previous
I_{47}→C→d,c/d/$
I_{89}→C→cC, c/d/$

LALR Parsing table construction:

State	Action			Goto	
	c	d	$	S	C
I_o	S_{36}	S_{47}		1	2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
89	r_2	r_2	r_2		

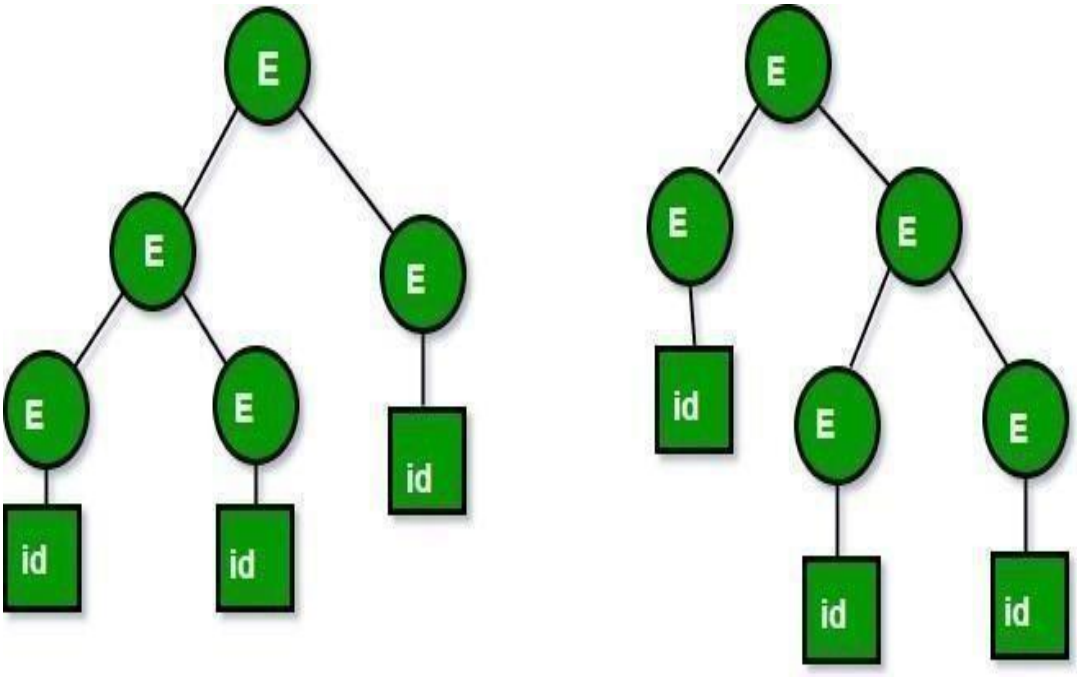
Ambiguous grammar:

A CFG is said to ambiguous if there exists more than one derivation tree for the given input string i.e., more than one **L**eft **M**ost **D**erivation **T**ree (LMDT) or **R**ight **M**ost **D**erivation **T**ree (RMDT).

Definition: G = (V,T,P,S) is a CFG is said to be ambiguous if and only if there exist a string in T* that has more than on parse tree.
where V is a finite set of variables.
T is a finite set of terminals.
P is a finite set of productions of the form, A -> α, where A is a variable and α ∈ (V ∪ T)* S is a designated variable called the start symbol.

For Example:

1. Let us consider this grammar : **E ->E+E|id**
We can create 2 parse tree from this grammar to obtain a string **id+id+id** :
The following are the 2 parse trees generated by left most derivation:



Both the above parse trees are derived from same grammar rules but both parse trees are different. Hence the grammar is ambiguous.

YACC PROGRAMMING

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language. Historically, they are also called compiler-compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 look ahead token) parser generator. YACC was originally designed for being complemented by Lex.

Input File:
YACC input file is divided in three parts.

```
/* definitions */
 ....


%%
/* rules */
....
%%


/* auxiliary routines */
....
```

Input File: Definition Part:

- The definition part includes information about the tokens used in the syntax definition:
- %token NUMBER

```
%token ID
```

- Yacc automatically assigns numbers for tokens, but it can be overridden by
```
%token NUMBER 621
```
- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.
- The definition part can include C code external to the definition of the parser and variable declarations, within **%{**and **%}** in the first column.
- It can also include the specification of the starting symbol in the grammar:
```
%start nonterminal
```
- The rules part contains grammar definition in a modified BNF form.
- Actions is C code in { } and can be embedded inside (Translation schemes).

Input File: Auxiliary Routines Part:

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part.
- It can also contain the main() function definition if the parser is going to be run as a program.
- The main() function must call the function yyparse().

Input File:

- If yylex() is not defined in the auxiliary routines sections, then it should be included:
```
#include "lex.yy.c"
```
- YACC input file generally finishes with:
```
.y
```

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the **main()** definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function **intyyparse()**
- If called with the **–d** option in the command line, Yacc produces as output a header file **y.tab.h** with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v** option, Yacc produces as output a file **y.output** containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

Semantics

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming languageconstructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semantic rules

Definition of (syntax Directed definition) SDD:

- SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)

Where a is an attributes obtained from the function f.

A syntax-directed definition is a generalization of a context-free grammar in which:

- Each grammar symbol is associated with a set of attributes.
- This setof attributes for a grammar symbol is partitioned into two subsets called synthesized and inherited attributes of that grammar symbol.
- Each production rule is associated with a set of semanticrules.
- Semantic rules set up dependencies between attributes which can be represented by a dependency graph.
- This dependency graph determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing avalue.

The two attributes for non terminalare :

The two attributes for non terminalare :

Synthesized attribute (S-attribute) :(↑)

An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node

Inherited attribute:(↑,→)

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of thatnode.

- The attribute can be string, a number, a type, a, memory location or anything else.
- The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.Terminals can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an Annotated parse tree.
- The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Ex1:1) Synthesized Attributes : Ex: Consider the CFG :

S→ EN

E→E+T

E→E-T

E→ T

T→ T*F

T→T/F

T→F

F→(E)

F→digit N→;

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production

Production rule	Semantic actions
S→EN	S.val=E.val
E→E1+T	E.val =E1.val +T.val
E→E1-T	E.val = E1.val –T.val
E→T	E.val=T.val
T→T*F	T.val = T.val *F.val
T→T|F	T.val =T.val | F.val
F→ (E)	F.val=E.val
T→F	T.val=F.val
F→digit	F.val =digit.lexval
N→;	can be ignored by lexical Analyzer as;I is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute "Val".

The taken digit has synthesized attribute "lexval".

In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.

Following steps are followed to Compute S attributed definition

Write the SDD using the appropriate semantic actions for corresponding production rule of the given Grammar.

The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom up manner.

The value obtained at the node is supposed to be final output.

L-attributed SDT

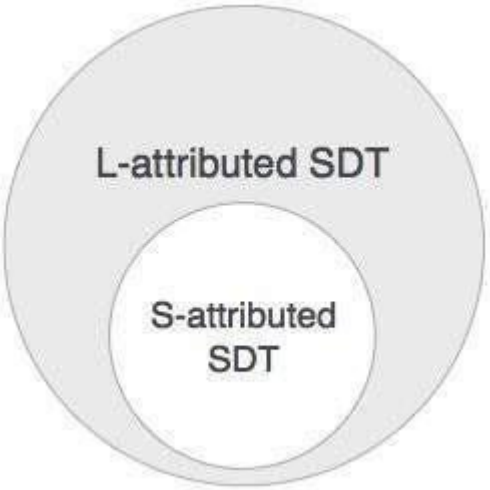
This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. As in the following production

S → ABC

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.

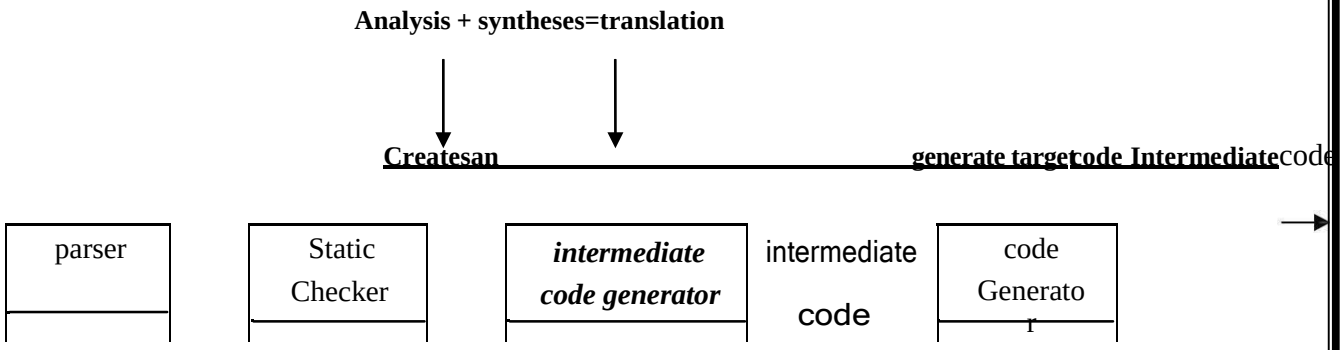
Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.



We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions

Intermediate Code

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an object module.



In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.

We assume that the source program has already been parsed and statically checked..the various intermediate code forms are:

a) Polishnotation
b) Abstract syntax trees(or)syntaxtrees
c) Quadruples
d) Triples three address code
e) Indirecttriples
f) Abstract machinecode(or)pseudocopde

postfix notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. the postfix (or postfix polish)notation for the same expression places the operator at the right end, asab+.

In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 is indicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfixexpression.

Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming language constructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semantic rules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)
Where a is an attributes obtained from the function f.

- A syntax-directed definition is a generalization of a context-free grammar in which:
- Each grammar symbol is associated with a set of attributes.

This set of attributes for a grammar symbol is partitioned into two subsets called synthesized and inherited attributes of that grammar symbol.

- Each production rule is associated with a set of semantic rules.

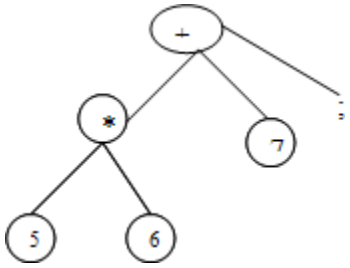
- Semantic rules set up dependencies between attributes which can be represented by a dependency graph.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an Annotated parse tree.

- The process of computing the attributes values at the nodes is called annotating(or decorating) of the parse tree.Of course, the order of these computations depends on the dependency graph induced by the

Syntax tree:



Annotated parse tree :

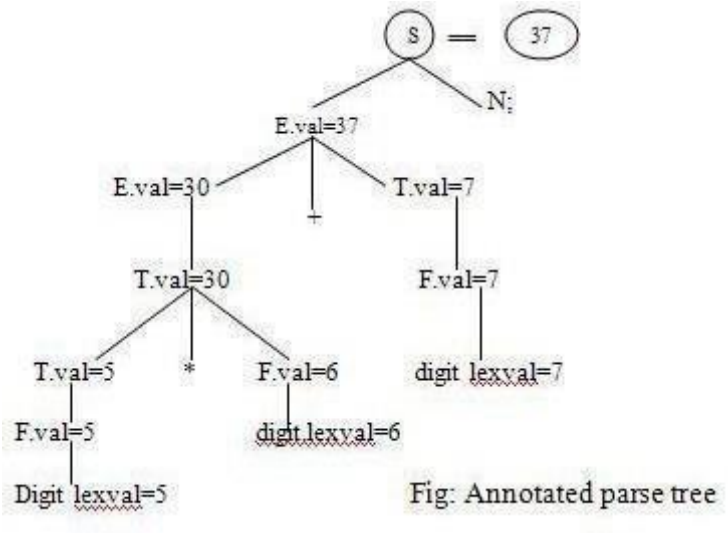


Fig: Annotated parse tree

ASSIGNMENT STATEMENTS

Suppose that the context in which an assignment appears is given by the following grammar. P M D

M ε
D D ; D | **id** : T | **proc id** ; N D ; S
N ε

Nonterminal P becomes the new start symbol when these productions are added to those in the translation scheme shown below.

Translation scheme to produce three-address code for assignments

S →id := E { p : = lookup (**id**.name);
ifp≠nil **then**
emit(p ' : =' E.place)
elseerror }

E → E1 + E2 { E.place : = newtemp;

emit(E.place ': =' E1.place ' + ' E2.place) }

E → E1 * E2 { E.place : = newtemp;
emit(E.place ': =' E1.place ' * ' E2.place) }

E →- E1 { E.place : = newtemp;
emit (E.place ': =' 'uminus' E1.place) }

E →(E1) { E.place : = E1.place }

E → id { p : = lookup (**id**.name);
ifp≠nil **then**
E.place : = p
elseerror }

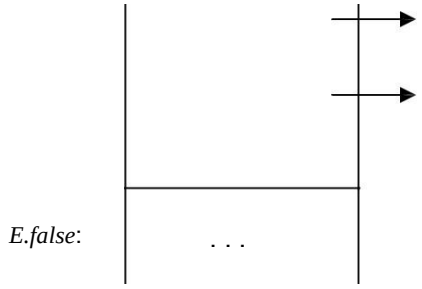
Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of if- then, if-then-else, and while-do statements such as those generated by the following grammar:

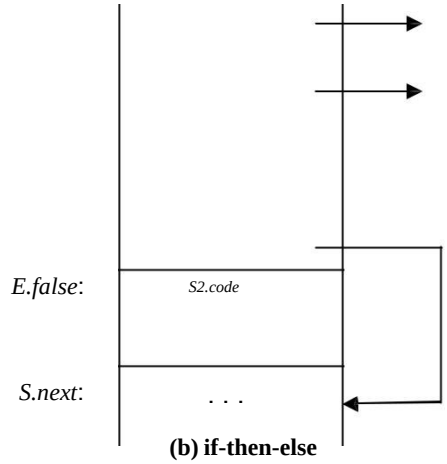
S **if** E **then** S1
if E **then** S1 **else**
| S2
| **while** E **do** S1

In each of these productions, *E* is the Boolean expression to be translated. In the translation, we assume that a three-address statement can be symbolically labeled, and that the function *newlabel* returns a new symbolic label each time it is called.

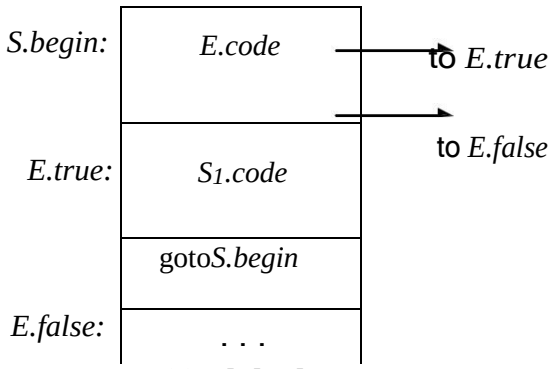
- E.true is the label to which control flows if E is true, and E.false is the label to which control flows if E is false.
- The semantic rules for translating a flow-of-control statement S allow control to flow from the translation S.code to the three-address instruction immediately following S.code.
- S.nextis a label that is attached to the first three-address instruction to be executed after the code for **Code for if-then , if-then-else, and while-do statements**



(a) if-then



(b) if-then-else



(c) while-do

PRODUCTION	SEMANTIC RULES
→	
S **if** *E* **then** S_1	*E.true : = newlabel;* *E.false : = S.next;* S_1*.next : = S.next;* *S.code : = E.code || gen(E.true „:") ||* S_1*.code*
→	
S **if** *E* **then** S_1**else** S_2	*E.true : = newlabel;* *E.false : = newlabel;* S_1*.next : = S.next;* S_2*.next : = S.next;* *S.code : = E.code || gen(E.true „:") ||* S_1*.code || gen(„***goto***" S.next) || gen(E.false „:") ||* S_2*.code*
→	
S **while***E* **do** S_1	*S.begin : = newlabel;* *E.true : = newlabel;* *E.false : = S.next;* S_1*.next : = S.begin;* *S.code : = gen(S.begin „:")|| E.code || gen(E.true „:") ||* S_1*.code || gen(„***goto***" S.begin)*

Unit-III

According to Chomsky hierarchy, grammars are divided of 4 types:

Type 0 known as unrestricted grammar.
Type 1 known as context sensitive grammar.
Type 2 known as context free grammar.
Type 3 Regular Grammar.

Type 0: Unrestricted Grammar:
In Type 0
Type-0 grammars include all formal grammars. Type 0 grammar language are recognized by turing machine. These languages are also known as the Recursively Enumerable languages.

Grammar Production in the form of

alpha \to \beta where alpha is (V + T)* V (V + T)* V
: Variables
T : Terminals.
beta is (V + T)*.
In type 0 there must be at least one variable on Left side of production.

For example,
Sab –>ba
A –> S.
Here, Variables are S, A and Terminals a, b.

Type 1: Context Sensitive Grammar)
Type-1 grammars generate the context-sensitive languages. The language generated by the grammar are recognized by the Linear Bound Automata
In Type 1
I. First of all Type 1 grammar should be Type 0.
II. Grammar Production in the form of

alpha \to \beta
alpha| <= |\beta|
i.e count of symbol in \alpha is less than or equal to \beta

For Example,
S –> AB
AB –>abc
B –> b

Type 2: Context Free Grammar:
Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Pushdown automata. Type-2 grammars generate the context-free languages.

In Type 2,
1. First of all it should be Type 1.
2. Left hand side of production can have only one variable.

alpha= 1.

Their is no restriction on \beta.

For example,
S –> AB
A –> a
B –> b

Type 3: Regular Grammar:
Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite state automaton.

Type 3 is most restricted form of grammar.
Type 3 should be in the given form only :

V –> VT* / T*.
(or)
V –> T*V /T*

for example :
S –> ab.

TypeChecking:

- A compiler has to do semantic checks in addition to syntactic checks.
- Semantic Checks
- Static –done during compilation
- Dynamic –done during run-time
- Type checking is one of these static checkingoperations.
- we may not do all type checking at compile-time.
- Some systems also use dynamic type checking too.
- A type system is a collection of rules for assigning type expressions to the parts of a program.
- A type checker implements a type system.
- A sound type system eliminates run-time type checking for type errors.
- A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.
- In practice, some of type checking operations is done at run-time (so, most of the programming languages are not strongly typed).
- –Ex: int x[100]; … x[i] most of the compilers cannot guarantee that i will be between 0 and 99

Type Expression:

- The type of a language construct is denoted by a type expression.

- **A type expression can be:**

–A basic type

- **a primitive data type such as integer, real, char, Boolean, …**

- type-error to signal a type error

- void: notype

A type name

- **a name can be used to denote a type expression.**
- **A type constructor applies to other type expressions.**
- arrays: If T is a type expression, then array (I,T)is a type expression where I denotes index range. Ex: array(0..99,int)
- products: If T1and T2 are type expressions, then their Cartesian product T1 x T2 is a type expression. Ex: int xint
- pointers: If T is a type expression, then pointer (T) is a type expression. Ex: pointer(int)
- functions: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression D→R where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

Type Checking of Statements:

S->d=E	**{ if (id.type=E.type Then S.type=void else S.type=type- error**
S ->if Ethen S1	{ if (E.type=boolean then S.type=S1.type else S.type=type-error}
S->while EdoS1	{ if (E.type=boolean then S.type=S1.type else S.type=type-error}

Type Checking of Functions:

E->E1(E2)	**{ if (E2.type=s and E1.type=s□ t)thenE.type=t else E.type=type-error}**

Ex: intf(double x, char y) { ... }

f: double x char->int

argumenttypes **return type**

Structural Equivalence of Type Expressions:

•How do we know that two type expressions are equal?
•As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

Structural Equivalence Algorithm (sequin):

if (s and t are same basic types) then return true
else if (s=array(s1,s2) and t=array(t1,t2))

then return (sequiv(s1,t1)

andsequiv(s2,t2))

else if(s= s1 x s2and t = t1 x t2)

then return (sequiv(s1,t1)

and sequiv(s2,t2))

else if (s=pointer(s1) and t=pointer(t1)) then return (sequiv(s1,t1))

else if (s = s1 □ s2and t = t1 □ t2) then return (sequiv(s1,t1) and sequiv(s2,t2))

else return false

Names for Type Expressions:

In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

type link= ↑cell;? p,q,r,s have same

types ? varp,q :link;

varr,s : ↑cell

•How do we treat type names?
Get equivalent type expression for a type name (then use structural equivalence), or
Treat a type name as a basic type

Overloading of Functions and Operators

AN OVERLOADED OPERATOR may have different meanings depending upon its context.
Normally overloading is resolved by the types of the arguments,
but sometimes this is not possible and an expression can have a set of possible types.
Example 2 In the previous section we were resolving overloading of binary arithmetic operators by looking at the the types of the arguments. Indeed we had two possibles types, say $ \mathbb {Z}$ and $ \mathbb {R}$, with a natural coercion due to the inclusion
$\displaystyle \mbox{${\mathbb Z}$}$ $\displaystyle \subseteq$ $\displaystyle \mbox{${\mathbb R}$}$ (2)
But what could we do if we had the three types $ \mathbb {Z}$, $ \mbox{${\mathbb Z}$}$/p$ \mbox{${\mathbb Z}$}$ and $ \mbox{${\mathbb Z}$}$/m$ \mbox{${\mathbb Z}$}$ for two different integers m and p?
There is no natural coercion between $ \mbox{${\mathbb Z}$}$/p$ \mbox{${\mathbb Z}$}$ and $ \mbox{${\mathbb Z}$}$/m$ \mbox{${\mathbb Z}$}$.
So the type of an expression like 1 + 2 and consequently the signature of + may also depend on

what is done with 1 + 2.

SET OF POSSIBLE TYPES FOR A SUBEXPRESSION. The first step in resolving the overloading of operators and functions occuring in an expression E' is to determine the possible types for E'.

For simplicity, we restrict here to unary functions.
We assign to each subexpression E of E' a synthesized attribute E.types which is the set of possible types for E.

These attributes can be computed by the following translation scheme.

E' $ \longmapsto$ E	{ E'.types :=	E.types }
E $ \longmapsto$ $ \bf id$	{ E.types :=	lookup(id.entry) }
E $ \longmapsto$ E1[E2]	{ E.types :=	{t | ($ \exists$ s $ \in$ E2.types) | (s $ \rightarrow$ t) $ \in$ E1.types} }

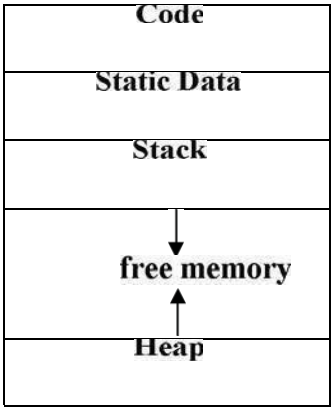
NARROWING THE SET OF POSSIBLE TYPES.

The second step in resolving the overloading of operators and functions in the expression E' consists of determining if a unique type can be assigned to each subexpression E of E' and generating the code for evaluating each subexpression E of E'.
This is done by assigning an inherited attribute E.unique to each subexpression E of E' such that either E can be assigned a unique type and E.unique is this type,or E cannot be assigned a unique type and E.unique is $ \bf type_error$.assigning a synthesized attribute E.code which is the target code for evaluating Eand executing the following translation scheme (where the attributes E.types have already been computed).

Unit-IV

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

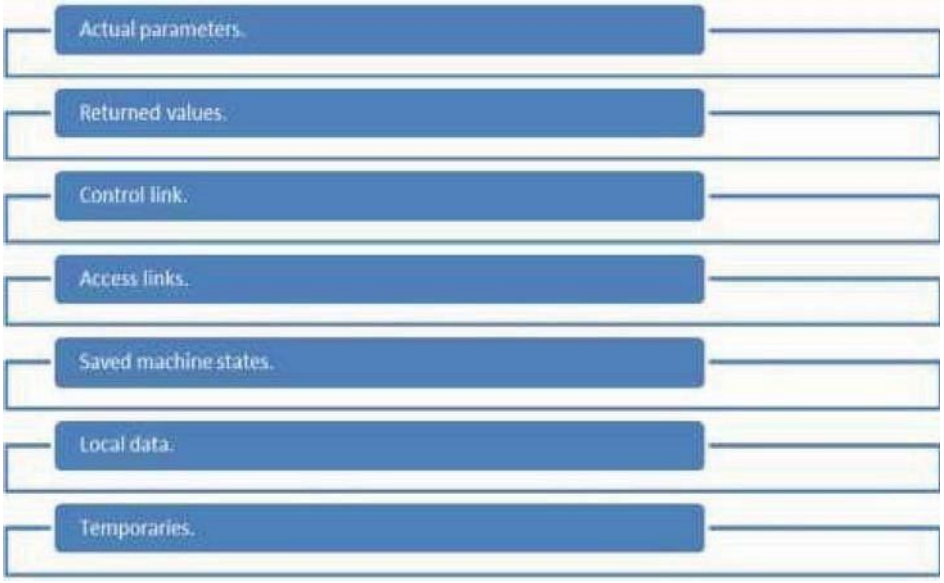


Run-time storage comes in blocks, where a byte is the smallest unit of addressable bytes and given the address of first byte.

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

Space for the return value of the called functions, if any. Again, not all called procedures efficiency.

- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation –** lays out storage for all data objects at compile time
2. **Stack allocation –** manages the run-time storage as a stack.
3. **Heap allocation –** allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.

- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

Scope access to non local names

Scope rules define the visibility rules for names in a programming language. What if you have references to a variable named k in different parts of the program? Do these refer to the same variable or to different ones?

Most languages, including Algol, Ada, C, Pascal, Scheme, and Haskell, are statically scoped. A block defines a new scope. Variables can be declared in that scope, and aren't visible from the outside. However, variables outside the scope -- in enclosing scopes -- are visible unless they are overridden. In Algol, Pascal, Haskell, and Scheme (but not C or Ada) these scope rules also apply to the names of functions and procedures.

Static scoping is also sometimes called lexical scoping.

Simple Static Scoping Example

```
 begin
integer m, n;

procedure hardy;
   begin
   print("in hardy -- n = ", n);
   end;

procedure laurel(n: integer);
   begin
   print("in laurel -- m = ", m);
   print("in laurel -- n = ", n);
   hardy;
   end;

m := 50;
n := 100;
print("in main program -- n = ", n);
laurel(1);
hardy;
end;
```

The output is:

```
in main program -- n = 100
in laurel -- m = 50
in laurel -- n = 1
in hardy -- n = 100    /* note that here hardy is called from laurel */
in hardy -- n = 100    /* here hardy is called from the main program */
```

11.5

in hardy -- n = 1 ;; NOTE DIFFERENCE -- here hardy is called from laurel in hardy -- n = 100 ;; here hardy is called from the main program

Static Scoping with Nested Procedures

```
begin
integer m, n;

procedure laurel(n: integer);
   begin
```

```
    procedure hardy;
       begin
       print("in hardy -- n = ", n);
       end;

    print("in laurel -- m = ", m);
    print("in laurel -- n = ", n);
    hardy;
    end;

m := 50;
n := 100;
print("in main program -- n = ", n);
laurel(1);


/* can't call hardy from the main program any more */
   end;
```

The output is:

```
in main program -- n = 100 in
laurel -- m = 50
in laurel -- n = 1 in
hardy -- n = 1
```

Parameters

Various parameter passing methods:

- call by value: passing r-values
- call by reference: passing l-values
- call by copy-restore: hybrid between by-value and by-reference; AKA Call by *Value-Result*
- call by name: passing via name substitution; is rarely used; first designed for Algol-60

Call by Value

- Each actual argument is evaluated before call. On entry, the resulting value is copied and bound to the formal parameter; which behaves just like a local variable
- Advantages:
  - Simple; easy to understand!
- Formal parameters can be used as local variables, Updating them doesn't affect actuals in calling procedure:

```
double hyp( double a, double b ) {
  a = a * a;
  b = b * b;
  return sqrt( a + b ); // use built-in sqrt()
} //end hyp
```

Problem with Call by Value

Can be inefficient if value is large:

```
typedef struct { double a1, a2, ..., a10; } vector;
double dotp( vector v, vector w ) {
   return v.a1 * w.a1 + v.a2 * w.a2 + ... + v.a10 * w.a10;
} //end dotp
vector v1, v2;
double d = dotp( v1, v2 ); // copy 20 doubles
```

Cannot affect calling environment directly:

```
void swap( int i, int j ) {
   int temp;
   temp = i; i = j; j = temp;
} //end swap
swap( a[p], a[q] ); // has no effect!
```

Can return result only through (the single) return value

Call by Reference

Implemented by passing address of actual parameter:

- On entry, the formal is bound to the address, providing a reference to actual parameter from within the subroutine
- If actual argument doesn't have an l-value (e.g., "2 + 3"), then either:
  - Forbid in language, i.e. treat as an error; compiler catches this
  - evaluate it into a temporary location and pass its address; but what will be the modified value of actual after the call?
- Advantages:
  - No more large copying
  - Actual parameter can be updated — Now swap, etc., work fine!

Call by Copy-Restore

- Each actual argument is evaluated to a value before call
- On entry, value is bound to formal parameter just like a local
- Updating formal parameters doesn't affect actuals in calling procedure during execution
- Upon exit, the final contents of formals are copied into the actuals
- Thus, behaves like call by reference in most "normal" situations, but may give different results when concurrency or aliasing are involved:

```
type t is record a, b: integer; end record;
r : t;
procedure foo( s : in out t ) is
  begin r.a := 2; s.a := s.a + 3;
end foo;
r.a := 1;
foo( r );
print( r.a ); -- what's the value of r.a?
```

Call by Name

- First introduced in Algol-60, first implemented via “thunk” by Ingerman
- Idea derived from substitution or macro-expansion. In C and some other languages, macros are used to define simple functions. The user interface of a macro resembles that of a function.

```
#define max( a, b ) ((a)>(b) ? (a) : (b))
v = max( x, y );
```

- When a macro is invoked, each formal parameter is literally replaced by corresponding actual parameter; string substitution:

```
max( x+1, y-2 ) => ( (x+1) > (y-2) ? (x+1) : (y-2))
```

- However, macros and functions are fundamentally different:
  - Macros are invoked at compile-time (or pre-compile-time), functions are called at runtime
  - Macros can not contain recursions

Call by Name

- First introduced in Algol-60, first implemented via “thunk” by Ingerman
- Idea derived from substitution or macro-expansion. In C and some other languages, macros are used to define simple functions. The user interface of a macro resembles that of a function.

```
#define max( a, b ) ((a)>(b) ? (a) : (b))
v = max( x, y );
```

- When a macro is invoked, each formal parameter is literally replaced by corresponding actual parameter; string substitution:

```
max( x+1, y-2 ) => ( (x+1) > (y-2) ? (x+1) : (y-2))
```

- However, macros and functions are fundamentally different:
  - Macros are invoked at compile-time (or pre-compile-time), functions are called at runtime
  - Macros can not contain recursions

Language facilities for dynamic storage allocation

The data under program control can be allocated dynamically.this allocation is done from heap.
Explicit allocation is the kind of memory allocation that can be with the help of some procedures.
Implicit allocation is a kind of allocation that can be done automatically for storing the results of expression evaluation**.**
Garbage is a ammount of memory which gets alloacted dynamically but is unreachable memory.
Dangling reference is a kind of complication that occurs due to explicit dealloaction of memory.

CODEOPTIMIZATION

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. Compilers that apply code- improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are
Machine independent optimizations:
Machine dependant optimizations:

Machine independent 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:
Machine dependant optimizations are based on register allocation and utilization of special machine- instruction sequences.

The criteria for code improvement transformations:

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.
- 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 such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.
- A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.
- The transformation must be worth the effort. It does not make sense for a compiler writer

- to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.
- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flowanalysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs sucha
    - control flow graph
    - Callgraph
- Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) ina computer program

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations
    - Common sub expressione limination,
    - Copy propagation,
    - Dead-code elimination,and
    - Constant folding, are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.
- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can usethe previously computedvalue.

Forexample
t1: =4*i
t2: =a [t1]
t3: =4*j
t4:=4*i
t5:=n
t 6: =b [t 4] +t 5

The above code can be optimized using the common sub-expression eliminationas t1:=4*i
t2:=a
[t1]t3:=4*j
t5:=n
t6: =b [t1] +t5
The common sub expression t 4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

CopyPropagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This maynot appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example: x=Pi;

…… A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r; Here the

variable x is eliminated

Dead-CodeEliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating deadcode.

Example: i=0;
if(i=1)
{
a=b+5;
}

Here,,,ifstatement is dead codebecausethis condition will never get satisfied.

Constant folding:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

- One advantage of copy propagation is that it often turns the copy statement into deadcode.

Forexample,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

Loop Optimizations

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outsidethatloop.
- Three techniques are important for loopoptimization:
  codemotion, which moves code outside aloop;

Induction -variable elimination, which we apply to replace variables from innerloop.
Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition

CodeMotion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note

that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:
while (i<= limit-2) /* statement does not change Limit*/ Code motion
will result in the equivalent of
t= limit-2;
while (i<=t) /* statement does not change limit or t */

Induction Variables

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.
- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

LOOPS IN FLOWGRAPH

- A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

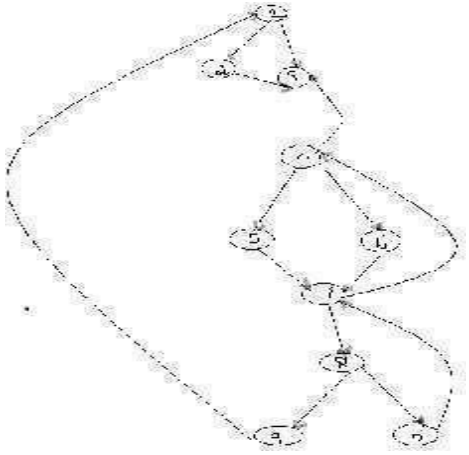
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

- In the flow graph below,
- Initial node,node1 dominates every node. *node 2 dominates itself
- node 3 dominates all but 1 and 2. *node 4 dominates all but 1,2 and 3.

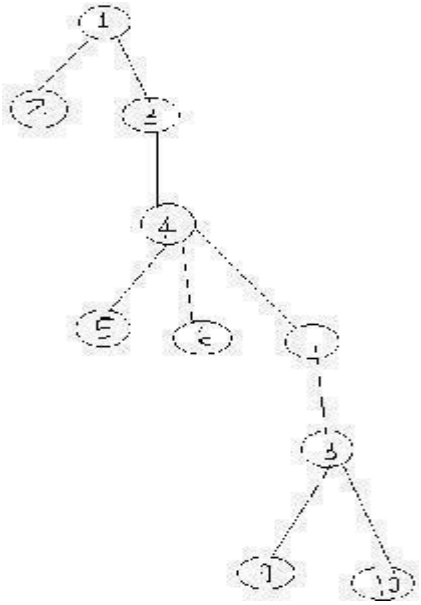
- node 5 and 6 dominates only themselves,since flow of control can skip around either by going through the other.
- node 7 dominates 7,8 ,9 and 10. *node 8 dominates 8,9 and 10.

- node 9 and 10 dominates only themselves



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.

- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in thetree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node ton.
- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d domm.



D(1)={1}

D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

NaturalLoop

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
- The properties of loops are
  - A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to theloop.
  - There must be at least one wayto iterate the loop(i.e.)at least one path back to the header.

One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as backedge.

Example:

In the above graph,

7 → 4 4 DOM7

0 →7 7 DOM10

4 → 3

8 → 3

9 →1

- The above edges will form loop in flowgraph.
- Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of theloop.

Algorithm: Constructing the natural loop of a back edge.
Input: A flow graph G and a back edge n→d

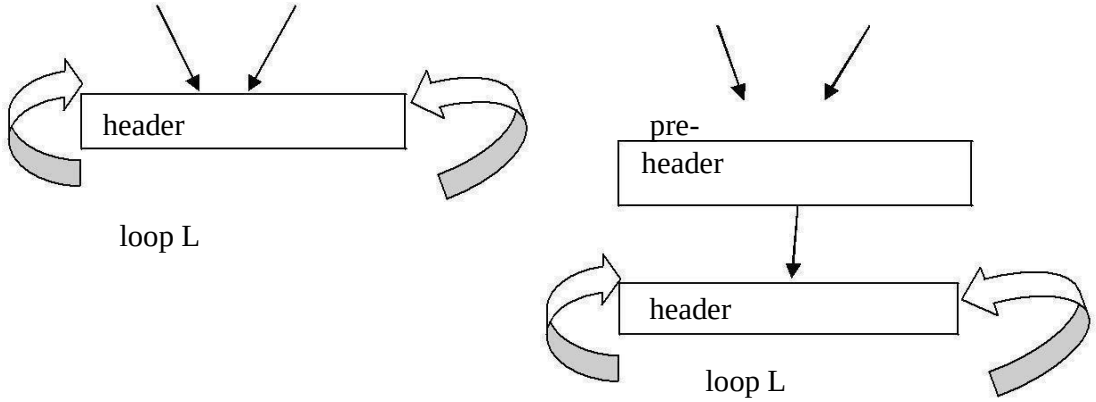
LOOP:

- If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no otherloop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as asingleloop.

Pre-Headers:

- Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called thepreheater.

- The pre -header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are notchanged.
- Initially the pre-header is empty, but transformations on L may place statements in it.



Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible. The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

***Definition*:**

- A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.
- The forward edges from an acyclic graph in which every node can be reached from initial node of G.
- The back edges consist only of edges where heads dominate their stails. Example: The above flow graph is reducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.
- The remaining edges are forwardedges.
- If the forward edges form an acyclic graph, then we can say the flow graph reducible.
- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.
- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a backedge

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions

by a shorter or faster sequence, whenever possible.

- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
    - We shall give the following examples of program transformations that are characteristic of peephole optimizations:

    - Redundant-instructionselimination
    - Flow-of-control optimizations
    - Algebraics implifications
    - Use of machine idioms
    - Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOVR0,a

(2) MOVa,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

INTRODUCTION TO GLOBAL DATAFLOWANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flowgraph.
- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data- flow information can be collected by setting up and solving systems of equations of the form:

$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."

- The details of how data-flow equations areset and solved depend on threefactors.
- The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].
- Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Data-flow analysis of structuredprograms:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the

followingsyntax.

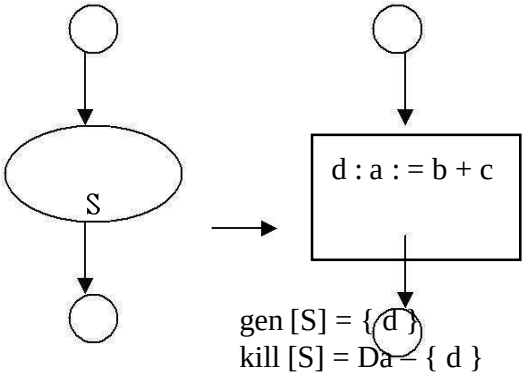
⟶**Sid: = E| S; S | if E then S else S | do S while**

⟶**E Eid + id|id**

- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.
- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for somethat enter the header.
- The Portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.
- We say that the beginning points of the dummy blocks at the entry and exit of a statement"s region are the beginning and end points, respectively, of the statement.The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statementsS.
- gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end ofS.

Consider the following data-flow equations for reaching definitions: i

)

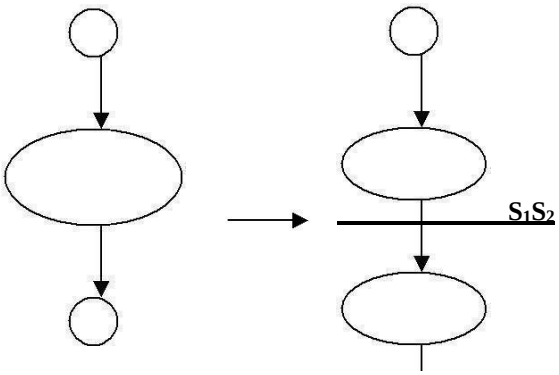


gen [S] = { d }
kill [S] = Da – { d }
out [S] = gen [S] U (in[S] – kill[S])

Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. ThusGen[S]={d}
On the other hand, d "kills" all other definitions of a, so we write Kill[S] = Da –{d}

Where, Da is the set of all definitions in the program for variable a.

ii



gen[S]=gen[S2] U(gen[S1]-kill[S2])
Kill[S] = kill[S2] U (kill[S1] – gen[S2])
in [S1] = in [S] in [S2] =
out [S1] out
[S] = out [S2]

- Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, wewrite

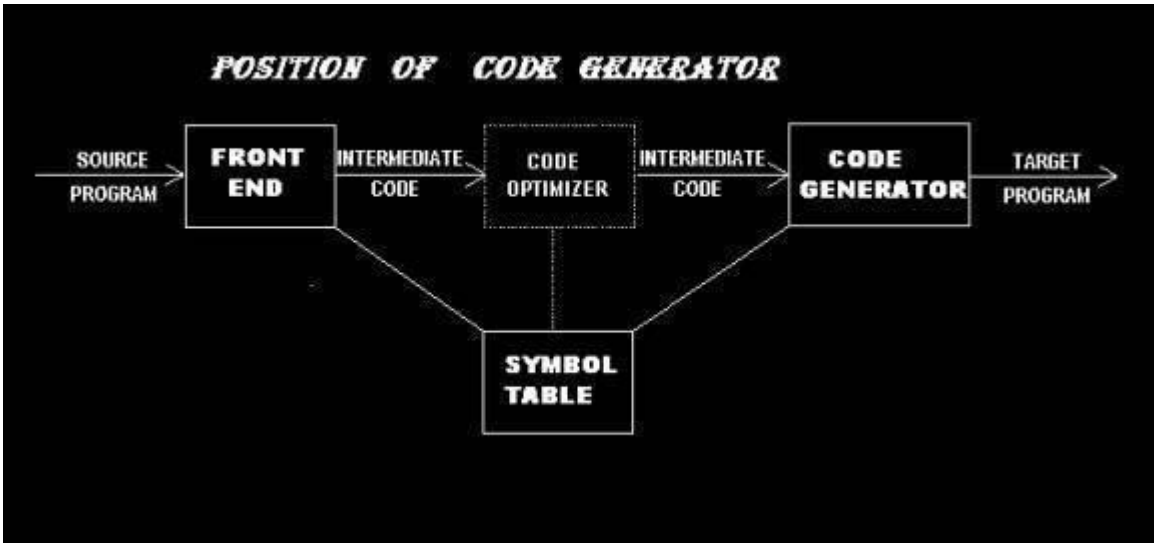
gen[S]=gen[S2] U (gen[S1]-kill[S2])
Similar reasoning applies to the killing of a definition, so we have Kill[S]
= kill[S2] U (kill[S1] –gen[S2])

Unit-V

OBJECT CODE GENERATION:

The final phase in our compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.



Object code forms

Absolute coding is a method of computer programming where the writer uses absolute instead of indirect addressing. For example, in an assembly language, the programmer may enter an exact memory address for data storage instead of an indirect address that a higher programming language may use.

After compilation of the source code, the object code is generated, which not only contains machine level instructions but also information about hardware registers, memory address of some segment of the run-time memory (RAM), information about system resources, read-write permissions ..etc.

Now there could be two ways for allocating run-time memory:-

1. The programmer decides which segment of the memory would be used during running of the programme.For doing this he passes memory address symbols or hex symbols in the programme.
2. Dynamic allocation by the linker, allocation of memory to the subroutines wherever, linker finds spaceforthem. This is something called **relocatable machine code**. i.e it doesn't have a static memory address for running.

Assembly Language is a low-level programming language. It helps in understanding the programming language tomachine code. In computer, there is assembler that helps in converting the assembly code into machine code executable. Assembly language is designed to understand the instruction and provide to machine language for further processing. It mainly depends on the architecture of the system whether it is the operating system or computer architecture.

Assembly Language mainly consists of mnemonic processor instructions or data, and other statements or instructions. It is produced with the help of compiling the high-level language source code like C, C++. Assembly Language helps in fine-tuning the program.

ISSUES IN THE DESIGN OF A CODE GENERATOR

While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation, and evaluation order are inherent in almost all code generation problems.

INPUT TO THE CODE GENERATOR

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation.

There are several choices for the intermediate language, including: linear representations such as postfix notation, three address representations such as quadruples, virtual machine representations such as syntax trees and dags.

We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.). We also assume that the necessary type checking has take place, so type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.

TARGET PROGRAMS

The output of the code generator is the target program. The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed quickly. A number of "student-job" compilers, such as WATFIV and PL/C, produce absolute code.

Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled programsegments.

Producing an assembly language program as output makes the process of code generation somewhat easier .We can generate symbolic instructions and use the macro facilities of the assembler to help generate code .The price paid is the assembly step after code generation.

Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must uses several passes.

A code-generation algorithm:

The 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, perform the following actions:

Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

Consult the address descriptor for y to determine y, the current location of y. Prefer the register for y if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y , L to place a copy of y in L.

Generate the instruction OP z , L where z is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

If the current values of y or z 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 of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

t : = a – b

u : = a – c

v : = t + u

d : = v + u

with d live at the end.

Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a – b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a – c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0

d : = v + u	ADD R_1, R_0
	
	MOV R_0, d
		R
		0 contains d
			d in R_0
			d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements a : = b [i] and a [i] : = b

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments a : = *p and *p : = a

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

REGISTER ALLOCATION

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During **register allocation,** we select the set of variables that will reside in registers at a point in theprogram.

2. During a subsequent **register assignment** phase, we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require register pairs (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

M x,y

where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form D x,y

where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient.

Now consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c). Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which "a' is to be loaded depends on what will ultimately happen to e.

t := a+b	**t := a +b**
t := t* c	**t := t +c**
t := t/ d	**t := t / d**

(a) fig. 2 Two three address code sequences

L	**R1, a**	**L**	**R0, a**
A	**R1, b**	**A**	**R0, b**
M	**R0, c**	**A**	**R0, c**
D	**R0, d**	**SRDA**	**R0, 32**
ST	**R1, t**	**D**	**R0, d STR1,t**
(a)		**(b)**	

THE DAG REPRESENTATION FOR BASIC BLOCKS

A DAG for a basic block is a directed acyclic graph with the following labels on nodes:

Leaves are labeled by unique identifiers, either variable names or constants.

Interior nodes are labeled by an operator symbol.

Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

DAGs are useful data structures for implementing transformations on basic blocks.

It gives a picture of how the value computed by a statement is used in subsequent statements.

It provides a good way of determining common sub - expressions

Input: A basic block

Output: A DAG for the basic block containing the following information:

A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.

For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

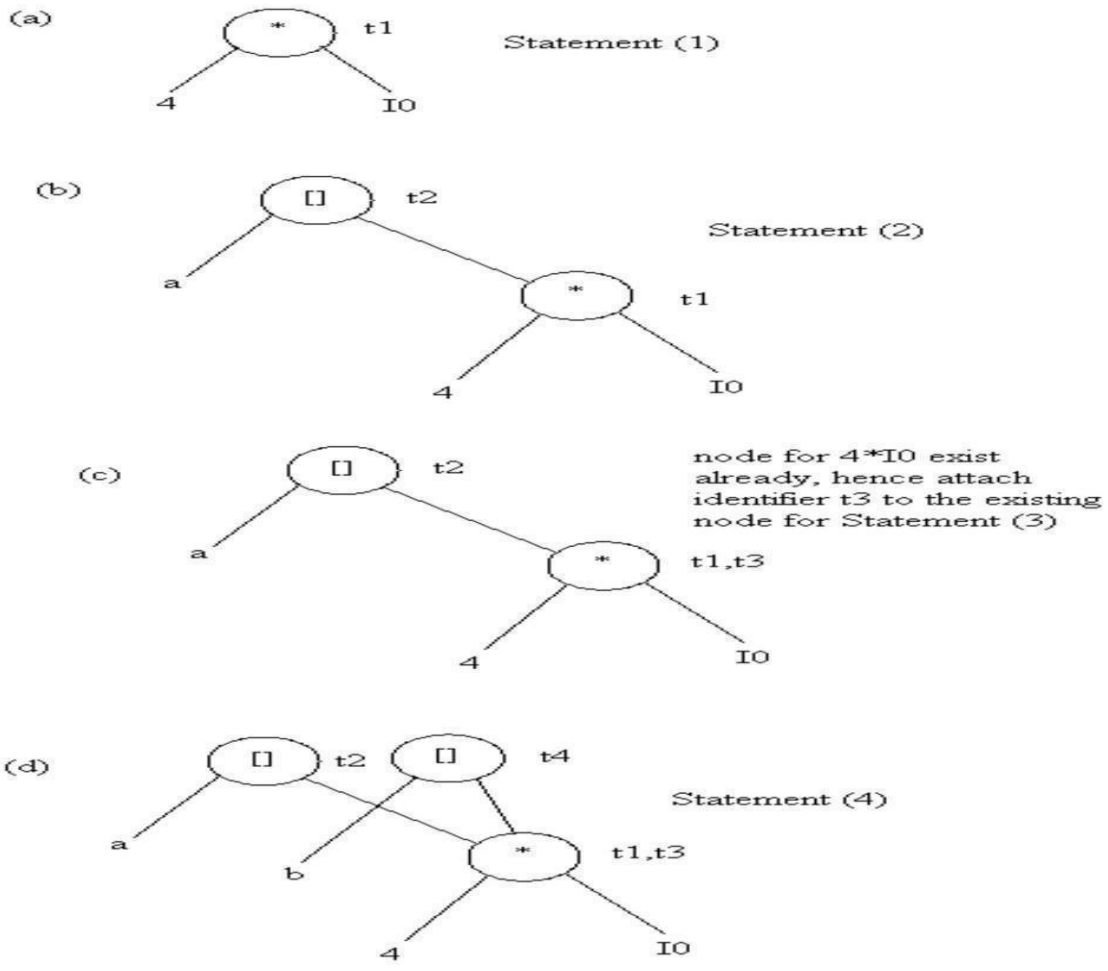
Step 3: Delete x from the list of identifiers for node(x). Append x to the listof attached identifiers for the

node n found in step 2 and set node(x) to n.

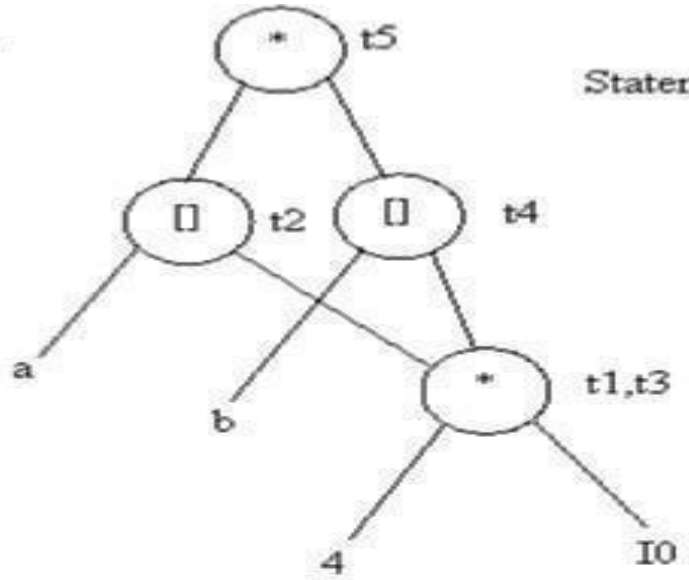
Example: Consider the block of three- address statements:

$t_1 := 4* i$
$t_2 := a[t_1]$
$t_3 := 4* i$
$t_4 := b[t_3]$
$t_5 := t_2*t_4$
$t_6 := prod+t_5$
$prod := t_6$
$t_7 := i+1$
$i := t_7$
if i<=20 goto (1)

Stages in DAG Construction

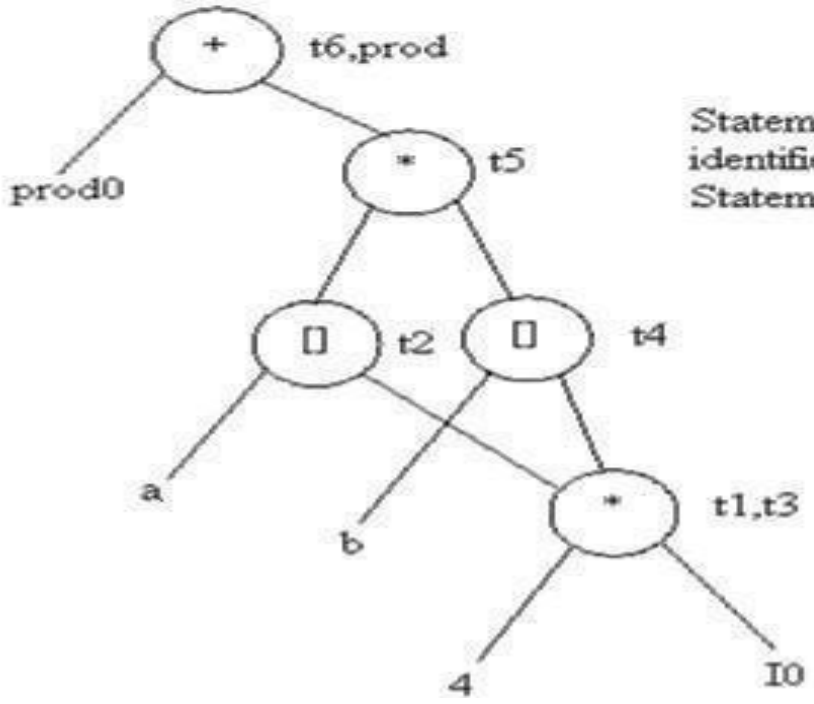


(e)



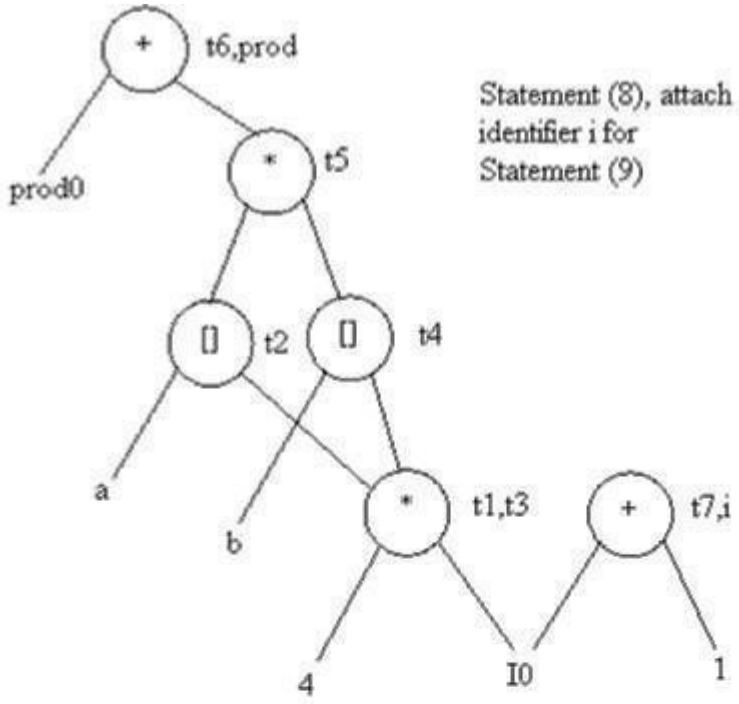
Statement (5)

(f)



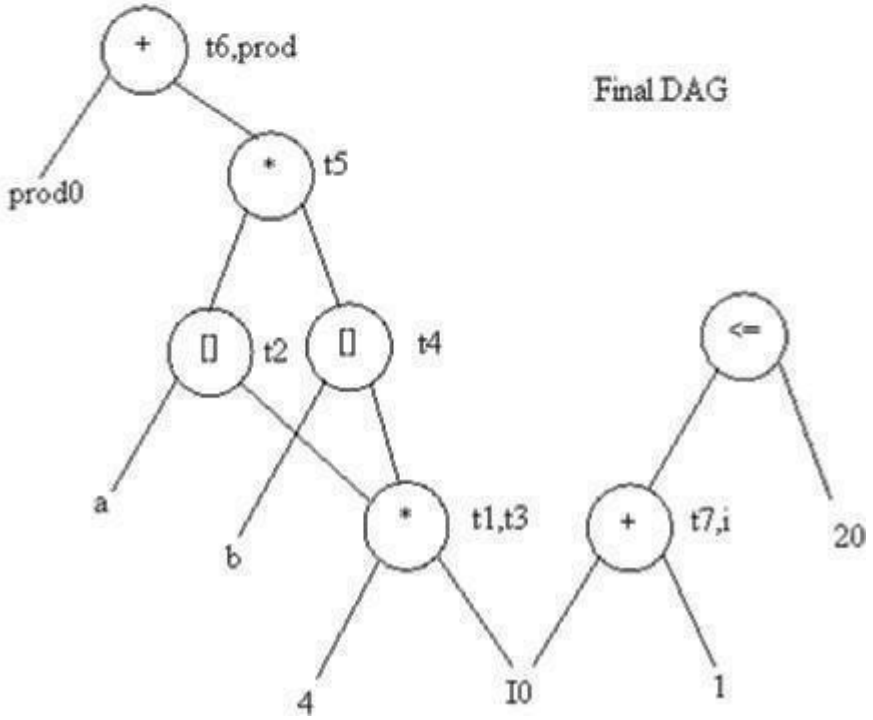
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can 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.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions MOV R_0 , t_1 and MOV t_1 , R_1 have been saved.