


Compiler Design

[Syllabus: BPSC CS: Introduction to compiliary. Basic issues, logical analysis, hexical analysis, syntax analysis. Semantic analysis, type checking, run-time environments, code generation, code optimization and language theory.]

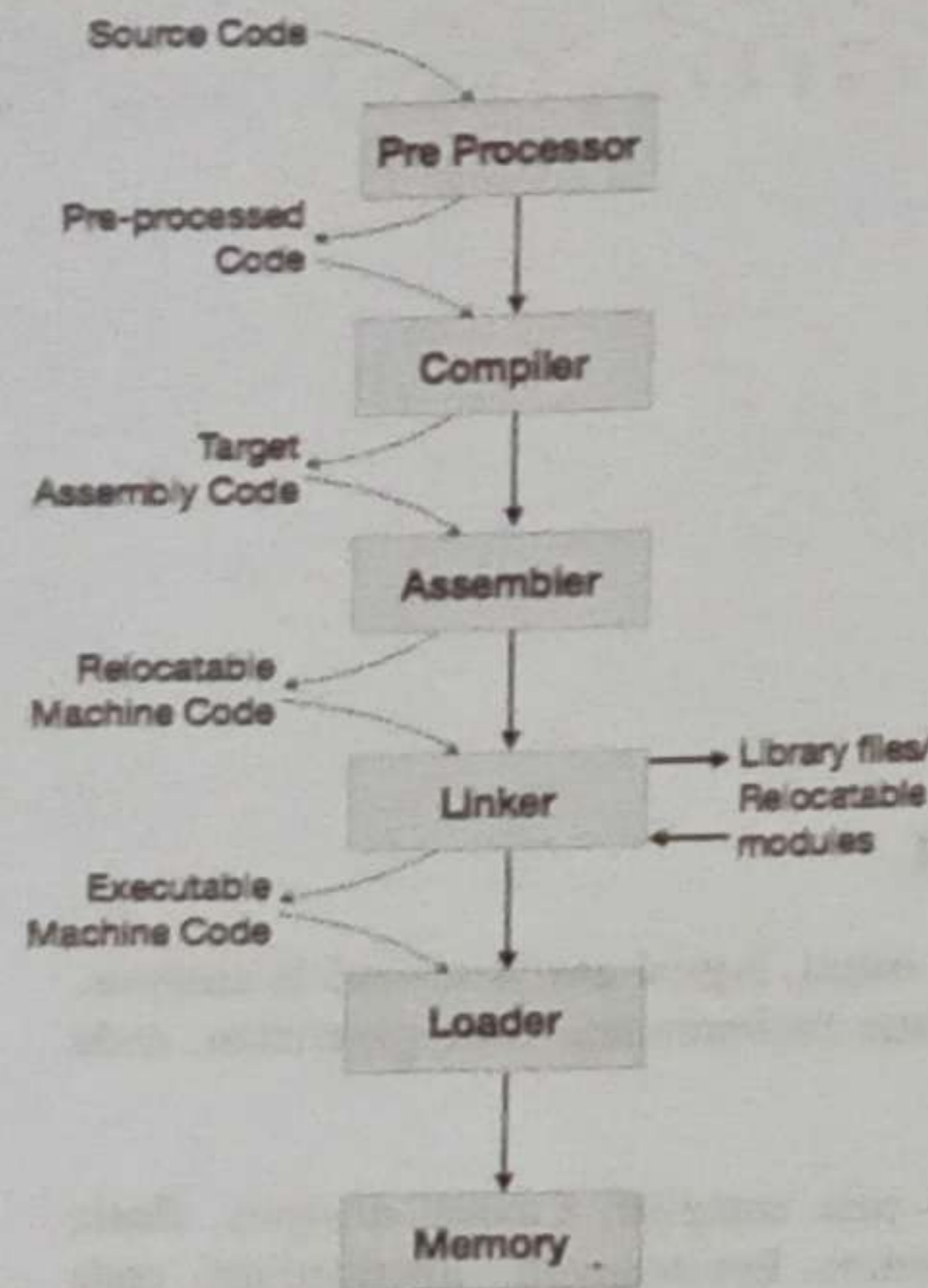
NTRCA CS: Introduction to compiler, A simple one pass compiler, Lexical analysis, Basic parsing technique, Syntax Directed Translation, Runtime Environment, Intermediate code generation, Code generation, Code optimization.]



Language Processing System

আমরা জানি যে একটি কম্পিউটার সিস্টেম সফটওয়্যার ও হার্ডওয়্যারের সমন্বয়ে গঠিত। কম্পিউটার হার্ডওয়্যার যে ভাষা বুঝে সেটি কিন্তু মানুষ বুঝে না। তাই আমরা প্রোগ্রাম লিখি মানুষের বোধ্য ভাষা হাইলেভেল ভাষায়। পরবর্তীতে এই হাই লেভেল ভাষা বিভিন্ন টুলস ও অপারেটিং সিস্টেমের বিভিন্ন কম্পোনেন্ট ব্যবহার করে মেশিন রিভেবল ভাষাতে রূপান্তরিত করে। এই প্রসেসটিকে আমরা ল্যাংগুয়েজ প্রোসেসিং সিস্টেম হিসেবে চিনি।

Language Processing System steps:



Compiler: কম্পাইলার হলো এক ধরনের অনুবাদক যা হাইলেভেল ভাষায় লিখিত প্রোগ্রামকে মেশিন ভাষায় রূপান্তর করে।

অ্যাসেম্বলার : অ্যাসেম্বলি ভাষায় লিখিত প্রোগ্রামকে মেশিন ভাষায় অনুবাদ করার জন্য যে অনুবাদক প্রোগ্রাম ব্যবহার করা হয়,তাকে অ্যাসেম্বলার বলা হয়। এটি অ্যাসেম্বলি ভাষায় লিখিত প্রোগ্রামকে যান্ত্রিক ভাষায় রূপান্তর করে। অর্থাৎ নেমোনিক কোডকে মেশিন ভাষায় অনুবাদ করে।

Preprocessor :

Preprocessor এমন একটা প্রোগ্রাম যা কোন প্রোগ্রামকে কম্পাইলার process করার পূর্বেই process করে। এটি macro-processing, augmentation, file inclusion, language extension, etc নিয়ে কাজ করে।

ইন্টারপ্রেটার ইন্টারপ্রেটার এক ধরনের অনুবাদক প্রোগ্রাম যা উচ্চস্তরের ভাষায় লিখিত প্রোগ্রামকে লাইন বা লাইন পড়ে এবং তা মেশিন ভাষায় রূপান্তর করে। তবে কম্পাইলার প্রথমে সোর্স প্রোগ্রামকে অবজেক্ট প্রোগ্রামে রূপান্তর করে এবং সর্বশেষ ফলাফল প্রদান করে। কিন্তু

ইন্টারপ্রেটার সোর্স প্রোগ্রামকে অবজেক্ট প্রোগ্রামে রূপান্তর করে না, ইন্টারপ্রেটার লাইন নির্বাহ করে এবং তাৎক্ষণিক ফলাফল প্রদর্শন করে।

Linker:

লিঙ্কার হল এমন সফটওয়্যার যা অতিরিক্ত ফাইলের সাথে যেমন মেজার ফাইলগুলির সাথে অবজেক্ট কোডকে লিঙ্ক করে এবং .exe এক্সিকিউশন সহ একটি এক্সিকিউটেবল ফাইল তৈরি করে। প্রোগ্রামটি অর্জনিত ফাংশনগুলি ব্যবহার করতে পারে। অর্জনিত ফাংশনগুলির কার্যকরিতা header ফাইলগুলিতে থাকে।

Loader:

লোডার অপারেটিং সিস্টেমের একটি অংশ যা প্রোগ্রাম এবং লাইব্রেরিগুলি লোড করার জন্য দায়বদ্ধ। লোডার লিঙ্কারের দ্বারা তৈরিকৃত এক্সিকিউটেবল ফাইলটিকে মূল স্মৃতিতে লোড করে। এটি মূল স্মৃতিতে এক্সিকিউটেবল মডিউল মেমরির স্থান বরাদ্দ করে।

Cross-compiler:

ক্রস কম্পাইলার হল একটি কম্পাইলার যা চলমান সিস্টেমে থেকে অন্য প্লটফর্মের জন্য এক্সিকিউটেবল কোড তৈরি করতে সক্ষম। ক্রস কম্পাইলারের একটি প্রাথমিক উদাহরণ এইমিকো (AIMICO), যা ইউনিট্যাক ২ তে এক্সিকিউট হত এবং আইবিএম ৭০৫ কম্পিউটারে জন্য এসেম্বলি কোড তৈরি করত।

Source-to-source Compiler

একটি কম্পাইলার যা একটি প্রোগ্রামিং ভাষার সোর্স কোড নেয় এবং এটিকে অন্য প্রোগ্রামিং ভাষার সোর্স কোডে অনুবাদ করে তাকে সোর্স থেকে সোর্স কম্পাইলার বলে।

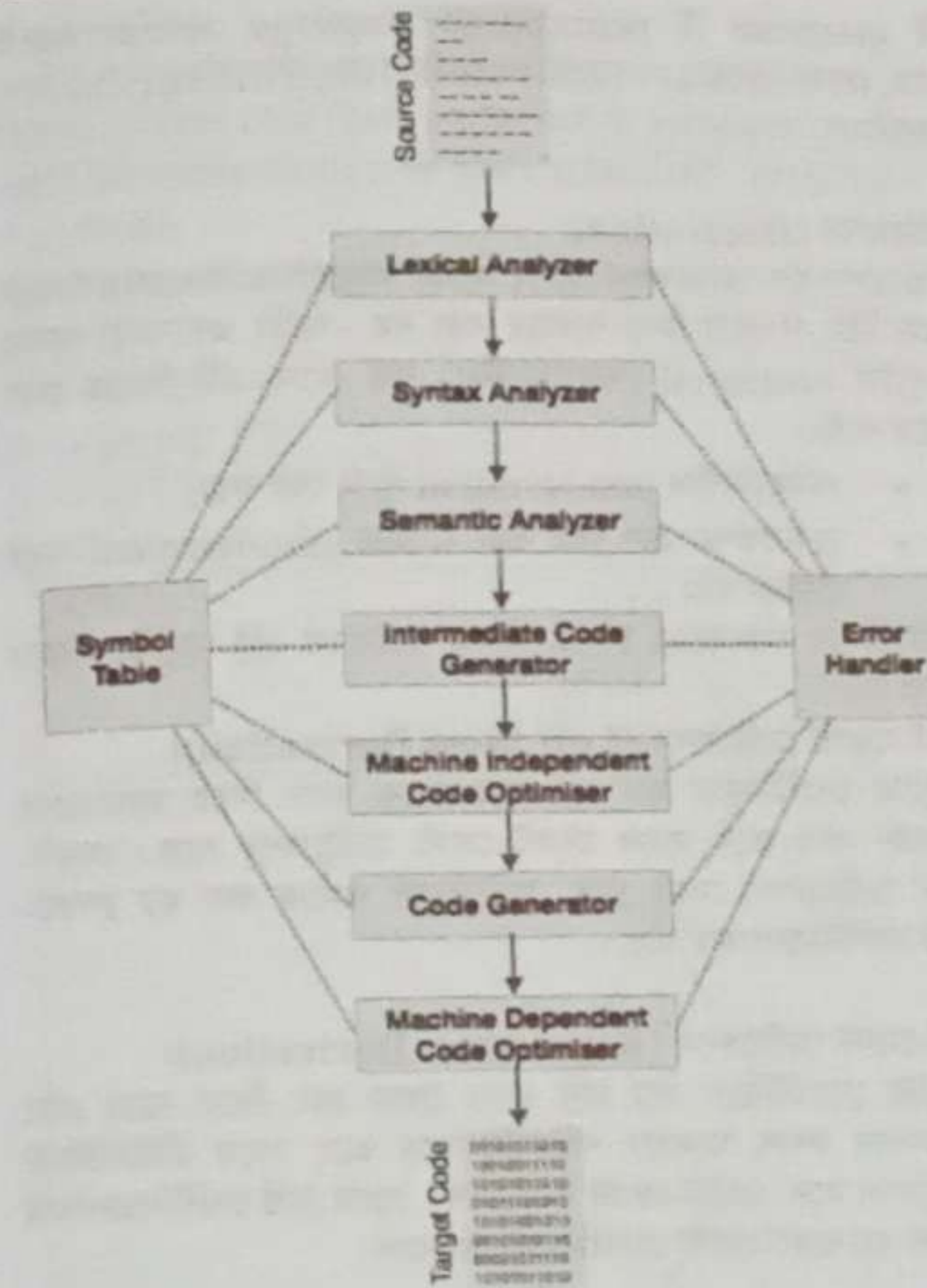
Phases Of Compiler

Compilation প্রক্রিয়া হল বিভিন্ন phase এর একটি ধারাবাহিক ধাপ। যেখানে প্রতিটি phase তার আগের ধাপ থেকে ইনপুট নেয়, উক্ত

phase এ সোর্স প্রোগ্রামের নিজস্ব representation থাকে এবং কম্পাইলারের পরবর্তী phase এ তার আউটপুট পাঠায়। চলুন একটি কম্পাইলারের phase গুলি দেখি।

There are six different phases available in the compiler

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation



Final Overview:

Phase Name	Operation
Lexical Analysis	Divided into Tokens
Syntax Analysis	Parse Tree
Semantic Analysis	Parse Tree which is semantically verified
Intermediate Code Generation	Three Address Code
Code Optimization	Reduce the size of Code
Code Generation	Target Machine Code

Lexical Analysis

প্রশ্ন ১: Lexical Analysis কি? সংক্ষেপে বর্ণনা কর। [NTRC-2014]

Lexical Analysis: এটিকে ক্যানিংও বলা হয়। এটি অনেকগুলো character কে সম্মত করে অর্থপূর্ণ ইউনিট তৈরি করে যাকে টোকেন বলে। এই ফেজ কম্পাইলার সোর্স কোডটিকে পড়ে এবং বিভিন্ন টোকেনে বিভক্ত করে।

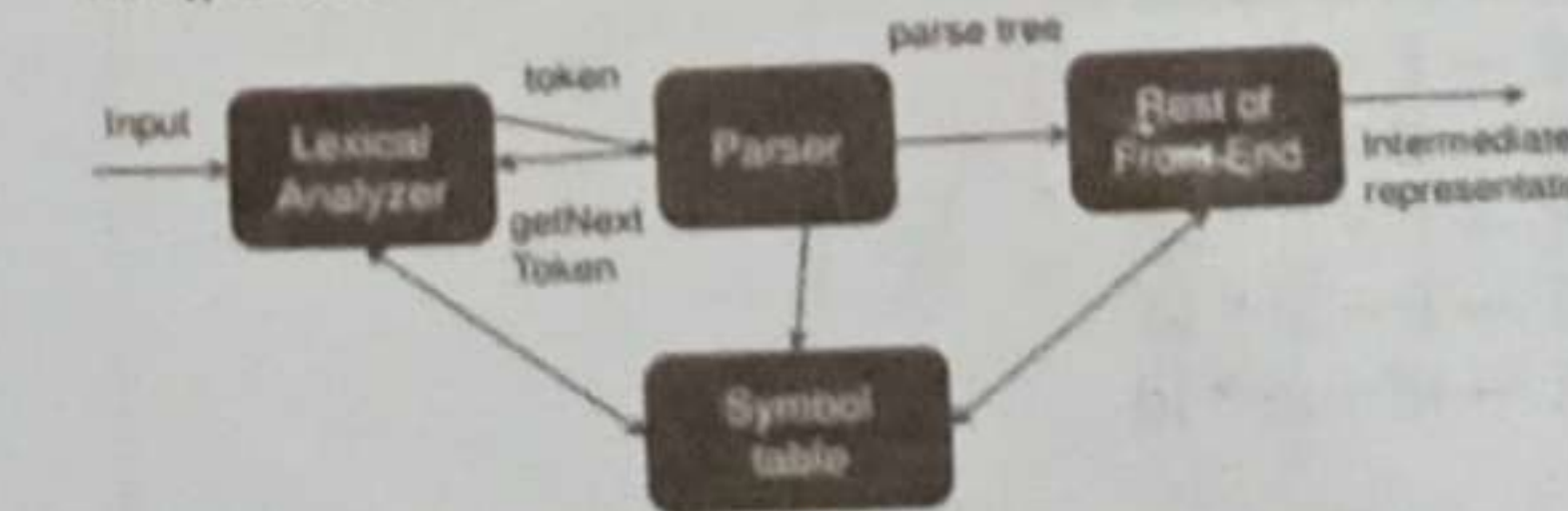


Fig: Lexical Analyzer Architecture

Lexical Analysis General Form:

(token-name, attribute-value)

- token-name -> an abstract symbol
- attribute-value -> entry in the symbol table for this token.

Lexical Analysis উদাহরণ:

position = initial + rate * 60

<id,1><=> <id,2><+><id,3><*><60>

• **Position:** টোকেন হিসেবে সনাক্ত করি এবং যেটি symbol table এর ১ম এ আছে নাম দেয়া হল id, সুতরাং (id,1)

Assignment symbol(=): এটিও একটি টোকেন (=)

• **Initial:** টোকেন হিসেবে সনাক্ত করি এবং যেটি symbol table এর ২য় তে আছে নাম দেয়া হল id, সুতরাং (id,2)

• **+:** এটিও একটি টোকেন (+).

• **Rate:** টোকেন হিসেবে সনাক্ত করি এবং যেটি symbol table এর ৩য় তে আছে নাম দেয়া হল id, সুতরাং (id,3)

• *****: এটিও একটি টোকেন (*)

• **60:** এটিও একটি টোকেন (60)

What's a lexeme?

একটি lexeme হল অক্ষরের ক্রম (sequence) যা একটি টোকেনের ম্যাচিং প্যাটার্ন অনুসারে source প্রোগ্রামে অন্তর্ভুক্ত করা হয়। এটি আসলে টোকেনের একটি উদাহরণ।

What's a token?

কম্পাইলার ডিজাইনে টোকেনগুলি হল অক্ষরের ক্রম যা source প্রোগ্রামে তথ্যের একটি ইউনিটকে প্রতিনিধিত্ব (represent) করে।

Example: Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens.

Sum = oldsum — value /100;

Token	Lexeme
IDENT	SUM
ASSIGN_OP	=
IDENT	oldsum
SUBTRACT_OP	-
IDENT	Value
DIVISION_OP	/
INT_LIT	100
SEMICOLON	;

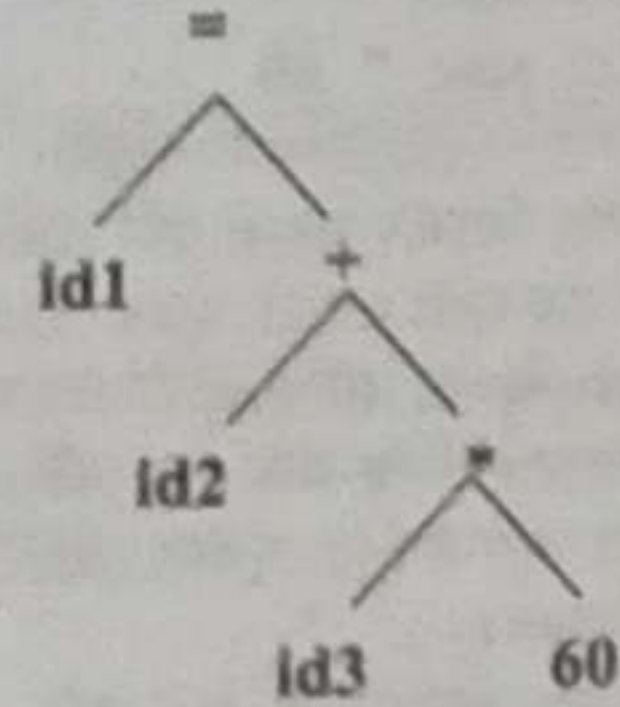
Syntax Analysis

Syntax Analysis কে Parsing ও বলা হয়। Syntax Analysis হল কম্পাইলার ডিজাইন প্রক্রিয়ার দ্বিতীয় ধাপ যেখানে formal grammar এর rules এবং structure নিশ্চিতকরণের জন্য প্রদত্ত ইনপুট স্ট্রিংটি পরীক্ষা করা হয়। Tree এর মাধ্যমে token form arrange করাকে Syntax Tree বলে।

Consider the example

• position = initial + rate * 60
(id1 = position, id2 = initial, id3 = rate)

Syntax tree:



Context-Free Grammar

একটি context-free grammar এ চারটি component বিদ্যমান।

১। **Non-terminals** এর একটি সেট (V) non-terminal গুলো হল syntactic variable, যা string এর set চিহ্নিত করে। non-terminal গুলো string গুলোর set সমূহ define করে, যা grammar দ্বারা উৎপন্ন language define করতে সাহায্য করে।

২। **Token** সমূহের একটি সেট (Σ) ইহা terminal symbol (Σ) নামে পরিচিত। terminal symbol গুলো হল basic বা মৌলিক symbol, যা হতে string সমূহ উৎপন্ন হয়।

৩। **Production** সমূহের একটি সেট (P) একটি grammar এর production সমূহ কার্যপদ্ধতি নির্ধারণ করে যাতে terminal এবং non-terminal গুলো একত্রে string তৈরি করতে পারে। প্রতিটি production একটি non-terminal নিয়ে গঠিত, একে production এর left side বলা হয় এবং, token এবং/অথবা non-terminal সমূহের একটি sequence বা ধারাকে, production এর right side বলা হয়।

৪। **Start symbol (S)** Start symbol থেকে production শুরু হয়। production এর right side দ্বারা উৎপন্ন string এর non-terminal গুলো start symbol হতে একের পর এক replace করে, এই non-terminal এর জন্য।

Example

Palindrome language এর problem নিয়ে আলোচনা করা যাক, যা Regular Expression দ্বারা describe করা যায় না। যাতে $L = \{w \mid w = w^R\}$ একটি regular language নয়। কিন্তু ইহা CFG দ্বারা describe করা যায়। তা নিম্নে দেখানো হল।

$G = (V, \Sigma, P, S)$

Where:

$V = \{Q, Z, N\}$

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

$P = \{Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \epsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1\}$

$S = \{Q\}$

এই grammar টি palindrome language accept করে।
পারে, যেমন- 1001, 11100111, 00100, 1010101, 1111 ইত্যাদি।

ডেরিভেশন (Derivation):

ডেরিভেশন হল প্রোডাকশন রুলের একটি সিকুয়েন্স যা সাধারণত ইনপুট থেকে ফিউ পাওয়ার জন্য ব্যবহার করা হয়। পার্সিং এর সময় ডেরিভেশন ইনপুটের sentential (সংবেদনশীল) ফর্ম থেকে ২টি সিদ্ধান্ত গ্রহণ করতে পারি:-

- পরিবর্তনশীল non-terminal খুঁজে বের করা।
- প্রোডাকশন রুল বের করা যা দ্বারা non-terminal সমূহ স্থানান্তর হবে।

মন টার্মিনাল পরিবর্তনের ক্ষেত্রে আমরা সাধারণত ২টি পদ্ধতি ব্যবহার করতে পারি।

লেফট-মোস্ট ডেরিভেশন (Left Most Derivation):

ইনপুটের সেন্টেন্সিয়াল ফর্ম যদি বাম থেকে ডানে দিকে ছ্যান এবং পরিবর্তন করে তবে তাকে লেফট-মোস্ট ডেরিভেশন বলে। লেফট-মোস্ট ডেরিভেশন থেকে প্রাপ্ত সেন্টেন্সিয়াল ফর্মকে বলা হয় লেফট-মোস্ট সেন্টেন্সিয়াল ফর্ম বলে।

রাইট-মোস্ট ডেরিভেশন (Right Most Derivation):

ইনপুটের সেন্টেন্সিয়াল ফর্ম যদি ডানে থেকে বাম দিকে ছ্যান এবং প্রোডাকশন রুলস অনুসারে পরিবর্তন করে তবে তাকে রাইট-মোস্ট ডেরিভেশন বলে। রাইট-মোস্ট ডেরিভেশন থেকে প্রাপ্ত সেন্টেন্সিয়াল ফর্মকে বলা হয় রাইট-মোস্ট সেন্টেন্সিয়াল ফর্ম বলে।

উদাহরন:

প্রোডাকশন রুলস:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

ইনপুট ফিউ: $id + id * id$

লেফট-মোস্ট ডেরিভেশন:

$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

উল্লেখ্য যে, লেফট-মোস্ট ডেরিভেশনের ক্ষেত্রে বাম দিকের ন টার্মিনাল সমূহ সবার আগে প্রসেস হবে।

রাইট-মোস্ট ডেরিভেশন:

$E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$

Parse Tree

পার্স টি হল ডেরিভেশন এর গ্রাফিক্যাল উপস্থাপন। ইনপুট সিদ্ধ থেকে ক্রমান্বয়ে টার্মিনাল নোড পাওয়া যায় তা পার্স টি এর মাধ্যমে দেখা যায়। একটি উদাহরণের মাধ্যমে দেখা যাক। আমরা একটি ইনপুট $(a + b * c)$ নিয়েছি।

যার লেফট-মোস্ট ডেরিভেশন হল:

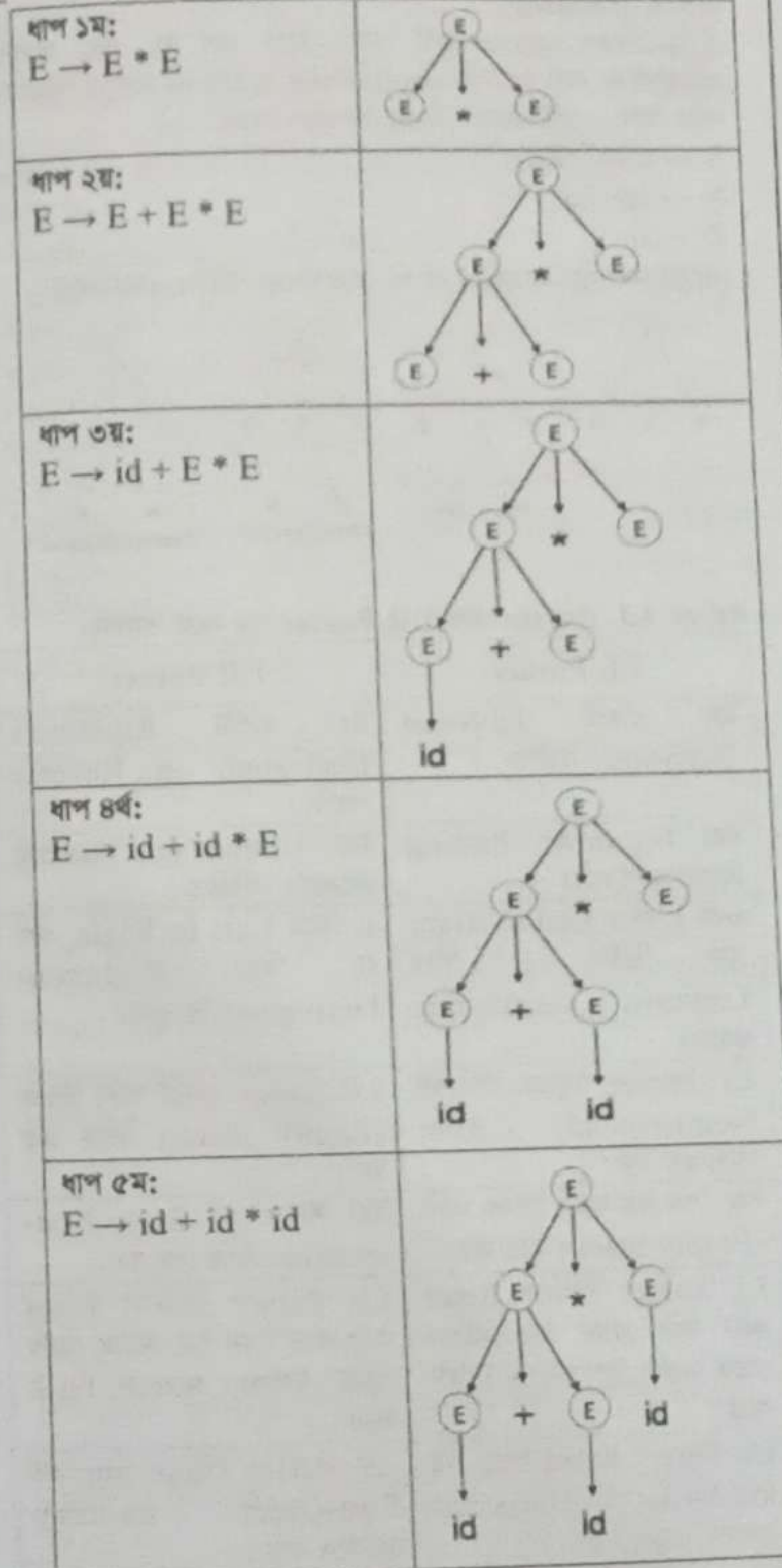
$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



parse tree এ:

- সমস্ত leaf নোডগুলো টার্মিনাল হিসাবে ব্যবহৃত হয়।
- সমস্ত অভ্যন্তরীণ নোডগুলো non-টার্মিনাল হিসাবে ব্যবহৃত হয়।
- In-order ট্রাভারসালে মূল ইনপুট সিদ্ধি দেওয়া হয়।

parse tree সাধারণত সংযুক্ততা এবং অপারেটরদের প্রাধান্য দিয়ে থাকে। প্রথমে sub-tree তে ট্রাভার্স করে, আর sub-tree এর অপারেটরের আধিকার অনুযায়ী পরবর্তী প্যারেন্ট নোডগুলিতে ট্রাভার্স করে থাকে।

Ambiguity (অনিচ্ছয়তা বা অস্পষ্টতা)

যদি grammar G এ কমপক্ষে একটি সিদ্ধির জন্য একাধিক parse tree (left বা right ডেরিভেশন) থাকে তাকে ambiguous বলা হয়।

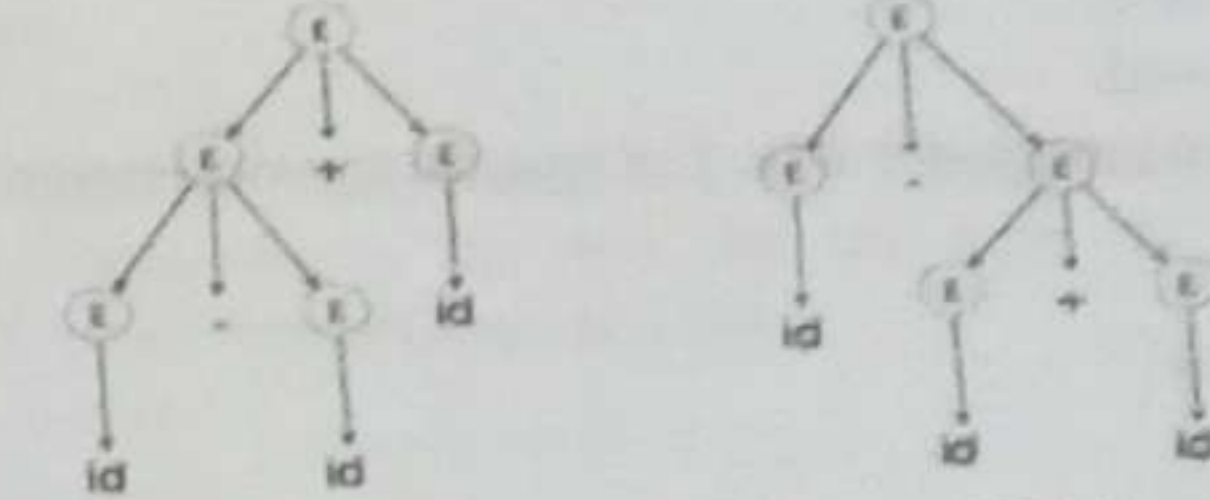
Example

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

$id + id - id$ এই string টির জন্য উপরের grammar দুইটি parse tree তৈরি করতে পারে।



ambiguous grammar দ্বারা উৎপন্ন language টি inherently ambiguous বলে মনে করা হয়।

compiler তৈরির জন্য grammar এ ambiguity থাকা ভাল নয়। কোন পদ্ধতিই স্বয়ংক্রিয়ভাবে ambiguity সনাক্ত করতে এবং মুছে ফেলাতে পারে না, তবে এটি re-writing মাধ্যমে পুরো grammar হতে remove করা যায়।

Ambiguity বিভিন্ন ভাবে দূর করা যায়

১। associativity এবং precedence প্রয়োগ করে।

২। left recursion এবং left factoring বাদ দিয়ে ব্যাকরণটি পুনরায় লিখে।

Example: Check " $a+a*a$ " whether the grammar 'a' with production rules: $x \rightarrow x+x \mid x*x \mid x \mid a$ is Ambiguous or not. (একটি grammar 'a' কি Ambiguous কিনা যাচাই কর, যার Production: $x \rightarrow x+x \mid x*x \mid x \mid a$).

Solution:

Division 1: (Left Most Derivation)

$x \rightarrow x \rightarrow x+x \rightarrow a+x \rightarrow a+x*x \rightarrow a+a*x \rightarrow a+a*a$

Derivation 2: (Right most derivation)

$x \rightarrow x \rightarrow x*x \rightarrow x+x*x \rightarrow a+x*x \rightarrow a+a*x \rightarrow a+a*a$

'a*a*a' has two left most derivation. Thus, the expression is ambiguous.

Parsing এর প্রকারভেদ/ Rules/ পদ্ধতি
Syntax analyzers production rules মাধ্যমে context-free grammar এর সংজ্ঞায়িত করে থাকে। production rules এর প্রয়োগের উপর ভিত্তি করে পার্সিংকে দুই ভাগে ভাগ করে যায়। যথাঃ

1. Top-down Parsing
2. Bottom-up Parsing.

Top-down Parsing

parser যখন শুরু symbol থেকে parse tree তৈরি করা শুরু করে এবং শুরুর symbol টিকে ইনপুটে রূপান্তরিত করার চেষ্টা করে, তখন তাকে top-down parsing বলা হয়।

Example:

$S \rightarrow E$
 $E \rightarrow E + T$
 $E \rightarrow E * T$
 $E \rightarrow T$
 $T \rightarrow id$

Input string: $a + b * c$, Let us start top-down parsing

S
 E
 $E + T$
 $E + c$
 $E * T + c$
 $E * b + c$
 $T * b + c$
 $a + b * c$

Bottom-up Parsing

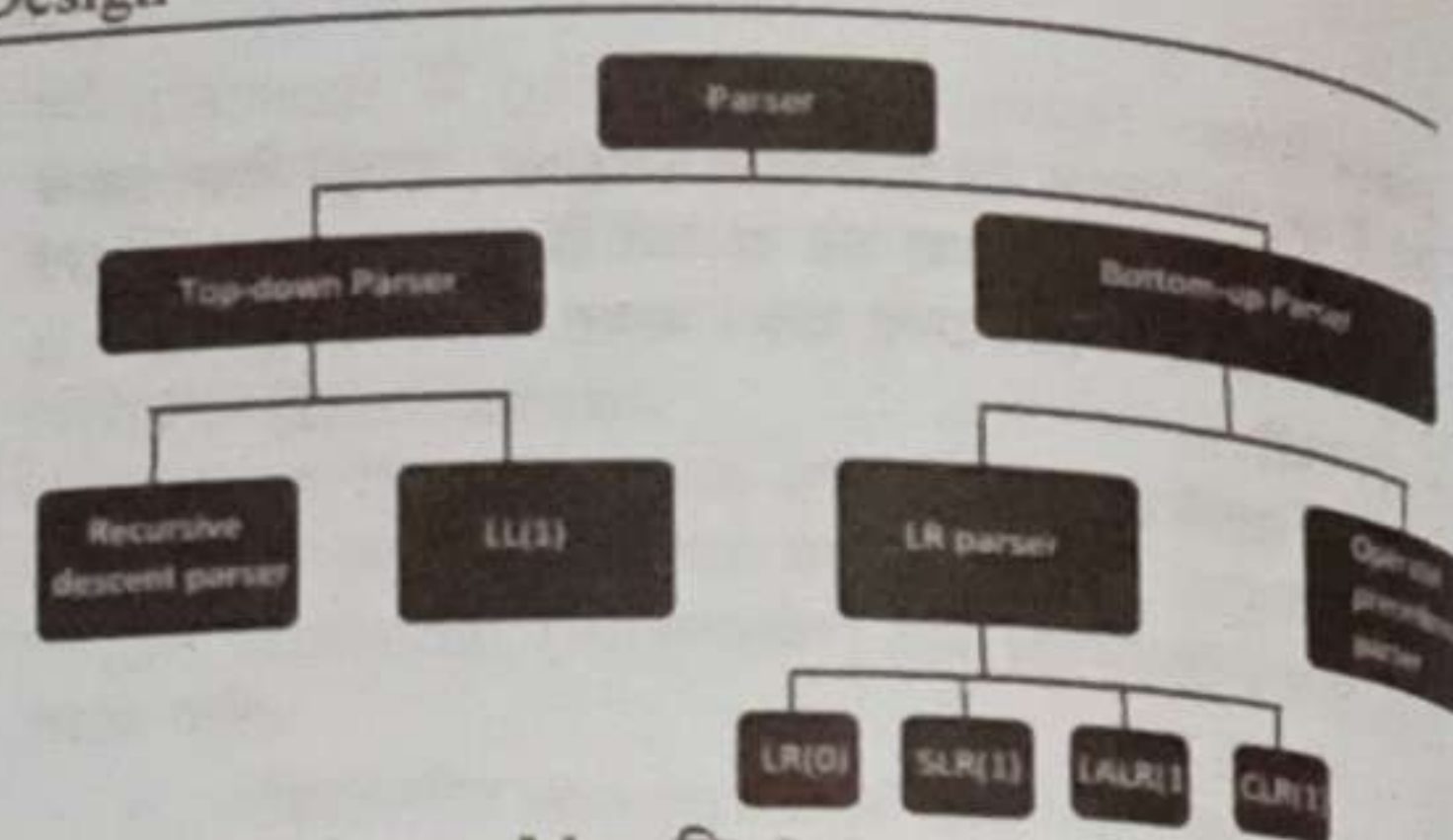
parser যখন শেষ symbol থেকে parse tree তৈরি করা শুরু করে এবং শেষ symbol টিকে ইনপুটে রূপান্তরিত করার চেষ্টা করে, তখন তাকে bottom-up parsing বলা হয়।

Example:

$S \rightarrow E$
 $E \rightarrow E + T$
 $E \rightarrow E * T$
 $E \rightarrow T$
 $T \rightarrow id$

Input string: $a + b * c$, Let us start bottom-up parsing

$a + b * c$
 $T + b * c$
 $E + b * c$
 $E + T * c$
 $E * c$
 $E * T$
 E
 S



প্রশ্ন ২: Back-tracking কি? [NTRC-2011]

Back-tracking

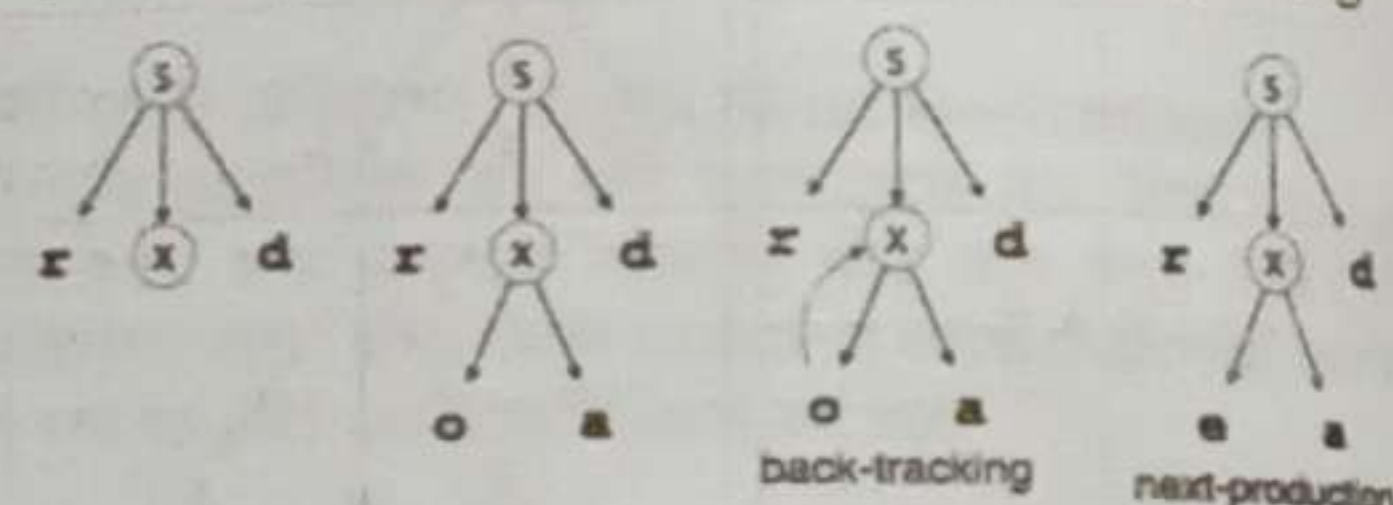
Top-down parser রুট নোড থেকে শুরু হয় এবং অনেক প্রতিস্থাপনের করা হয় যখন production rules এর ইনপুট স্ট্রিংয়ের সাথে মিলে। এটি বুঝতে, নীচের উদাহরণ দেখাও:

$S \rightarrow rXd \mid rZd$

$X \rightarrow oa \mid ea$

$Z \rightarrow ai$

input string: read, Let us start top-down parsing



প্রশ্ন ৩: LL Parser এবং LR Parser এর মধ্যে পার্থক্য:

LL Parser	LR Parser
ইহা একটি Leftmost Derivation পদ্ধতি।	ইহা একটি Rightmost Derivation in Reverse পদ্ধতি।
ইহা Top-down Parsing হিসেবেও পরিচিত।	ইহা Bottom-up Parsing হিসেবেও পরিচিত।
প্রথম L দিয়ে Left to Right এবং দ্বিতীয় L দিয়ে Leftmost Derivation বুঝায়।	L দিয়ে Left to Right এবং R দিয়ে Rightmost Derivation কে বুঝায়।
LL Parser শুধুমাত্র স্ট্যাকের Non-terminal Root দিয়ে শুরু হয়।	LR Parser একটি খালি স্ট্যাক (Empty Stack) দিয়ে শুরু হয়।
ইহা শেষ হয় যখন স্ট্যাক খালি (Empty Stack) হয়ে যায়।	ইহা স্ট্যাক এর Root Non-terminal দিয়ে শেষ হয়।
LL Parser টার্মিনাল Read করে যখন যখন ইহা স্ট্যাক থেকে একটা উপাদানকে POP করে।	LR Parser টার্মিনাল Read করে যখন যখন ইহা স্ট্যাক থেকে একটা উপাদান শুধুমাত্র Push করে।
LL Parser Parse tree এর Pre-Order traversal ব্যবহার করে।	LR Parser Parse tree এর Post-Order traversal ব্যবহার করে।
LL Parser Non-	LR Parser Non-terminal

terminal কে	Expands করে।	কে Reduces করে।
LL Parser write এর ক্ষেত্রে সহজ কিন্তু কম Powerful এবং LL(1) এর মতো অনেক রকমের (Flavours) হয়।	LR Parser অনেক Powerful এবং LR(0), SR(1), LALR(1), LR(1) এর মতো ইত্যাদি রকমের (Flavours) হয়।	

Semantic Analysis

১. Semantic Analysis কি? কাজ কি?

Semantic Analysis: Semantic analysis হচ্ছে text থেকে অর্থ বের করার প্রক্রিয়া। semantic এনালাইজার semantic ত্রুটির জন্য source program পরীক্ষা করে এবং code generation ধাপের জন্য ডাটা টাইপ তথ্য সংগ্রহ করে। semantic analysis একটি গুরুত্বপূর্ণ অংশ হল type checking। একপ্রকার ডেটা টাইপ থেকে অন্যপ্রকার ডেটা টাইপে রূপান্তর: উদাহরণ:

- position = initial + rate * 60
- position, initial এবং rate floating-point numbers হিসাবে ঘোষণা করা হয়েছে এবং এখানে lexeme 60 নিজেই একটি পূর্ণসংখ্যা (integer) গঠন করে।
- এই পূর্ণসংখ্যা (integer) একটি floating-point numbers এ রূপান্তরিত (Converted) হতে পারে।
- আউটপুট (Output) - int to float অপারেটরের জন্য আলাদা একটা নোড (Node) আছে। যা স্পষ্টভাবে integer এর যুক্তিকে (argument) কে একটি floating-point numbers এ রূপান্তরিত করে।

২. Explain Semantic Error in a context of Compiler.

Ans: কম্পাইলারের semantic analysis phase এ Semantic errors কম্পাইল-টাইমে সনাক্ত করা হয়। এর মধ্যে অপারেটর এবং অপারেন্ডের মধ্যে type mismatch ত্রুটিও রয়েছে।

৩. What is handle pruning in compiler design?

Ans: Handle Pruning হল shift-and-reduce parsing - এ ব্যবহৃত একটি সাধারণ পদ্ধতি। Handle হল একটি substring যা production এর body'র সাথে matches করে।

৪. Example of Handle

Consider the following CFG

$D \rightarrow TL;$
 $T \rightarrow int$
 $T \rightarrow float$
 $T \rightarrow char$
 $L \rightarrow L, id$
 $L \rightarrow id$

The parsing table for the input string int id,id; is shown on the table

Input String	Handle	Action
int id,id;	int	Reduce $T \rightarrow int$
T id,id;	id	Reduce $L \rightarrow id$

T L,id;	L, id	Reduce $L \rightarrow L, id$
TL;	TL;	Reduce $D \rightarrow TL$
D	-	Accepted

Intermediate Code Generation

Intermediate Code Generation:

- একটি Simple Machine-Independent Intermediate ভাষায় অনুবাদ করে।
- Low Level Language এ রূপান্তরিত করে।
- আউটপুট (Output) - Three-Address Code অনুসারে আসে।

Three-Address Code: Intermediate code generator তার Predecessor Phase (পূর্বসূরী পর্যায়), Semantic Analyzer, একটি টীকাযুক্ত সিনট্যাক্স ট্রি (Annotated Syntax Tree) আকারে ইনপুট গ্রহণ করে। সেই Syntax Tree টিকে তখন একটি Linear উপস্থাপনে (A Linear Representation) রূপান্তর করে। যেমন: Postfix Notation।

Intermediate code টি Machine-Independent Code হতে পারে। অতএব, কোড জেনারেটর সীমাহীন সংখ্যক মেমরি স্টোরেজ (রেজিস্টার) ধরে রাখে কোড উৎপন্ন করার জন্য।

উদাহরণ:

$a = b + c * d;$

Intermediate Code Generator এই Expression টিকে Sub-Expression এ ভাগ করার জন্য চেষ্টা করবে এবং তখন নিম্নোক্ত কোডটি উৎপন্ন (Generate) হবে:

$r1 = c * d;$
 $r2 = b + r1;$
 $r3 = r2 + r1;$
 $a = r3$

এখানে r ট্যাগেট প্রোগ্রাম এর মধ্যে রেজিস্টার হিসেবে ব্যবহৃত হয়েছে। একটি Three-Address Code এর কমপক্ষে তিনটি three address locations আছে এই Expression সমাধান (Calculate) করার জন্য। একটি Three-Address Code কে তিনটি উপায়ে প্রকাশ করা যায়: Quadruples, Triples এবং Indirect Triples.

উদাহরণ: position = initial + rate * 60

- t1 = int to float (60)
- t2 = id3 * t1
- t3 = id2 + t2
- id1 = t3

Code Optimization

প্রশ্ন ৪: Code Optimization কি? [NTRC-2017]

Code Optimization:

- Intermediate code কে Improve (উন্নতি) করার চেষ্টা করে আরও ভাল ট্যাগেট কোড (Target Code) তৈরি করার লক্ষ্যে।
- ইহা form এর মধ্যে সংক্ষিপ্ত ক্রমানুসারে থাকে।

Intermediate Code Generation phase এর Code Optimization নিম্নরূপ:

- t1 = id3 * 60.0
- id1 = id2 + t1

Code Generation

এই Code Generation কি Code Generation কাজ বর্ণনা কর। [NTRC-2017]

Code Generation: Code জেনারেটর three-address statement এর জন্য উচিত কোড produce করতে ব্যবহার করা হয়। এটি three address statement এর অপারেটর সংরক্ষণের জন্য প্রক্সিটার ব্যবহার করে। code optimization phase থেকে নিষ্কাশিত three address statement পাওয়া যায় code generation phase এর মাধ্যমে:

- MOVF ID3, R2
- MULF #60.0, R2
- MOVF id2, R1
- ADDF R2, R1
- MOVF R1, id1

নিম্নে কম্পাইলারের সকল ধাপের কাজ সংক্ষেপে দেয়া হল:

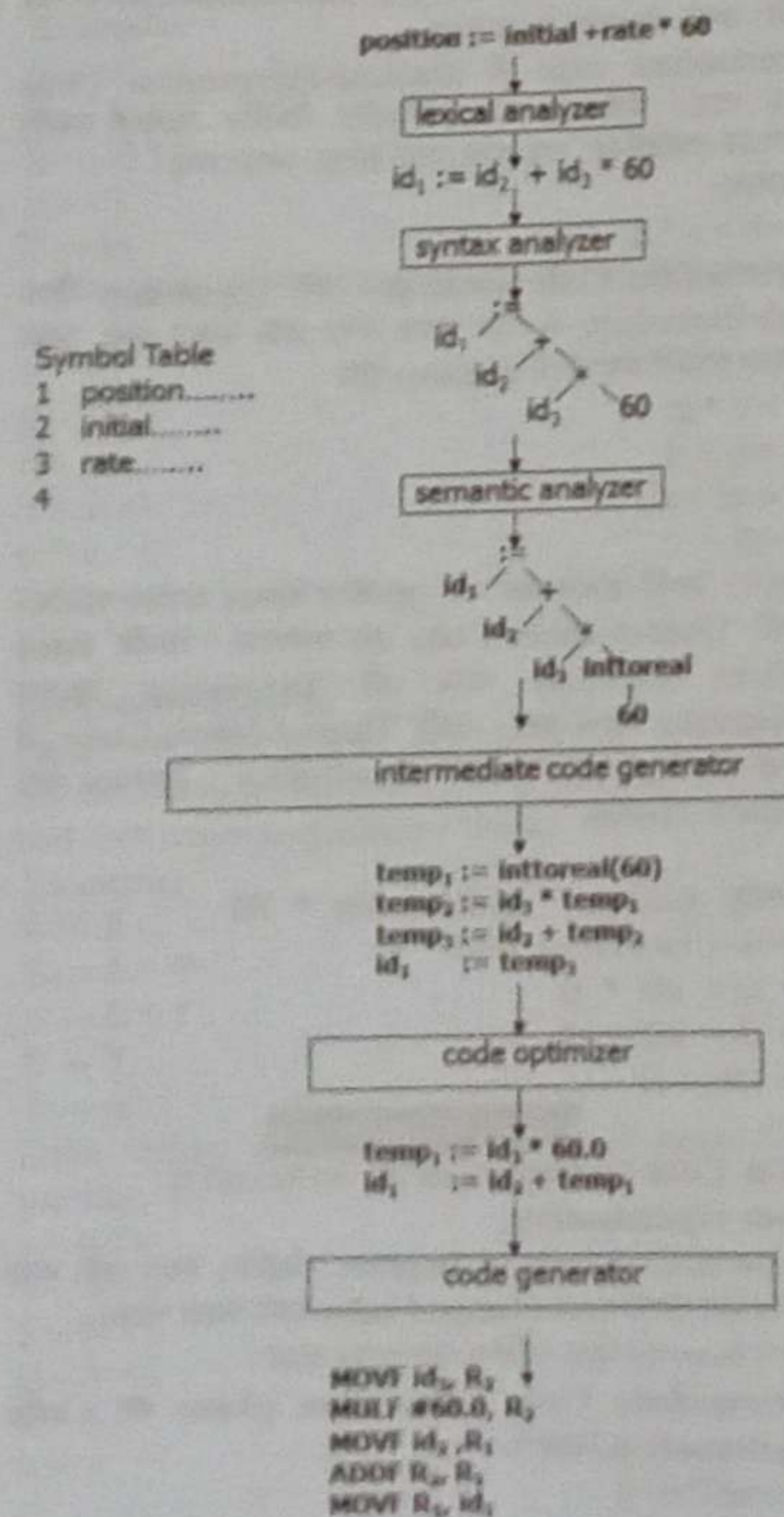


Fig: compilation stages

এই Symbol Table কি Symbol Table এর কাজ বর্ণনা কর। [NTRC-2013, 2014]

Symbol table হল একটি তথ্যভাণ্ডার যা কম্পাইলার দ্বারা তৈরি এবং রক্ষণাবেক্ষণ করা হয় যাতে বিভিন্ন এনটিটি জেনারেটরের নাম, ফাংশন নাম, অবজেক্ট, ক্লাস, ইটারেবল ইত্যাদি সম্পর্কে তথ্য এবং তাদের বৈশিষ্ট্য সংরক্ষণ করা যায়। Symbol table কম্পাইলারের analysis এবং synthesis উভয় অংশের জন্য ব্যবহৃত হয়। এই তথ্য ক্রমাগত সংগ্রহ করা হয় এবং কম্পাইলারের বিভিন্ন পর্যায়ে ব্যবহৃত হয়।

Use of a Symbol Table:

- a. Symbol table information is used by the analysis and synthesis phases.
- b. To verify that used identifiers have been defined (declared).
- c. To verify that expressions and assignments are semantically correct — type checking.
- d. To generate intermediate or target code.

Components of Symbol Table:

symbol table এর দুটি প্রধান উপাদান থাকবে:

1. Name Table: নাম টেবিল ব্যবহার করা হয় প্রোগ্রামের বিভিন্ন নামকে uniquely identify করতে।
2. Entity Table: Entity Table এ প্রোগ্রামের প্রতিটি entity একটি unique এন্ট্রি দ্বারা represent করা হয়।

Symbol Table এর কাজ

প্রধানত দুটি কাজ করে থাকে যথা:

- insert (name) প্রতি নামের জন্য একটি করে এন্ট্রি করে।
- lookup (name) টেবিল অনুসন্ধান করে নামের প্রসঙ্গের occurrence খুঁজি বের করে। [insert এর চেয়ে lookup অনেক বেশি ঘটে থাকে]

এছাড়া Symbol Table এর কাজ

- initializeScope (level)
- finalizeScope (level)

Given sample program to compile: lets see how symbol table is generated

```

defproc myproc (int A, float B){
    int D, E;
    D = 0;
    E = A;
    if (E>5){
        print D
    }
}
defproc myproc (int A, float B){
    int D, E;
    int D = 0;

```

```

E = A/round(B);
if (E>5){
    print D
}
}

```

Symbol Table:

Symb	Token	Dtype	Init?
defproc myproc (int A, float B){			
int D, E;			
int D = 0;			
E = A/round(B);			
if (E>5){			
print D			
}			
}			

Event: identifier = procedure name
Action: Add name to symbol name

Symbol Table:

Symb	Token	Dtype	Init?
myproc	id	procname	-
defproc myproc (int A, float B){			
int D, E;			
int D = 0;			
E = A/round(B);			
if (E>5){			
print D			
}			

Event: identifier = variable declaration, function arg
Action: Add name to symbol name as initialized

Symbol Table:

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
defproc myproc (int A, float B){			
int D, E;			
int D = 0;			
E = A/round(B);			
if (E>5){			
print D			
}			

Event: identifier = variable declaration, function arg
Action: Add name to symbol name as initialized

Symbol Table:

Symb	Token	Dtype	Init?
myproc	id	procname	-

A	id	int	yes
B	id	float	yes

```

defproc myproc (int A, float B){
    int D, E;
    int D = 0;
    E = A/round(B);
    if (E>5){
        print D
    }
}

```

Event: identifier = variable declaration, function arg
Check: Already in symbol table? if so, fail
Else: Add name to symbol name, not initialized

Symbol Table:

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes
D	id	int	no

Thus, the symbol table shows:

- How far the given program is compiled and stored in the table.
- Wntire program is transformed into a table.

Explain the functionality of pushdown automata.

উচ্চ pushdown automata হলো কয়েকটি গ্রামার ইম্প্লিমেন্ট করার একটি পদ্ধতি। pushdown automata সাধারণত ইনপুট স্ট্রিং কে বাম থেকে ডানে রিড করে। যা প্রতিটি step এ, ইনপুট স্ট্রিং, বর্তমান অবস্থা এবং স্ট্যাকের টপ সিলে দ্বারা একটি table কে indexing করে transition choose করে অর্থাৎ প্রতি ট্রানজিশনে স্ট্যাকের টপ ডালু হতে রিড করে। একটি pushdown automaton কোন transition সম্পাদন করার আগে হিসাবে স্ট্যাকটিকে পরিচালনা করতে পারে।

