http://www.rajanaryal.com.np/

http://hutrajshrestha.com.np/

Visit site For bsc csit syllabus/old question/notes/solution

You can follow the site for more notes.

Our Social Links

My Facebook Read my blog Follow me on twitter

Follow me on instagram My youtube channel My Facebook

Note : if you found any mistake on solution then please send us a message .

(You don't need to write long answer. Some of our answer is very long for better understanding so summarize yourself)

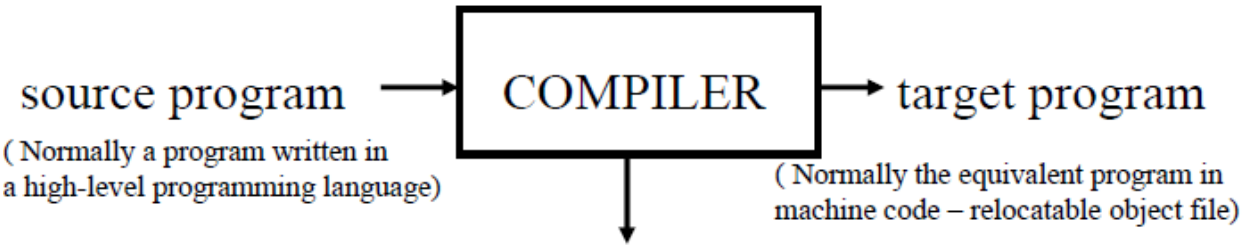
Bachelor of Computer Science and Information Technology

Course Title: Compiler Design and Construction

2071 (II)

1.) Define the compiler. Explain the phases of compiler.

Ans: A compiler is a program that takes a program written in a *source language* and translates it into an equivalent low level program in a *target language*.



Phases of a Compiler

There are two major parts of a compiler: Analysis and Synthesis

– In analysis phase, an intermediate representation is created from the given source program.

This phase (Source code analysis phase) is mainly divided into following three parts:

· Lexical Analyzer
· Syntax Analyzer and
· Semantic Analyzer

– In synthesis phase, the equivalent target program is created from this intermediate representation. This phase is divided into following three parts:

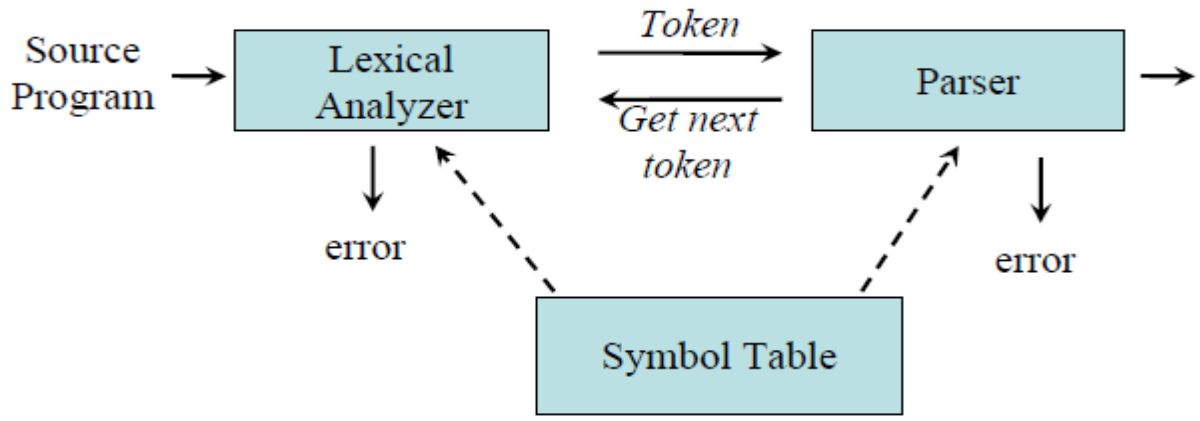
· Intermediate Code Generator
· Code Optimizer and
· Final Code Generator

Lexical Analyzer

Lexical Analyzer reads the source program in character by character ways and returns the *tokens* of the source program.

Normally a lexical analyzer doesn't return a list of tokens, it returns a token only when the parser asks a token from it.

Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.



Example:

newval := oldval + 12

tokens :	newval	identifier
	:=	assignment operator
	Oldval	identifier
	+	add operator
	12	a number

Put information about identifiers into the symbol table.

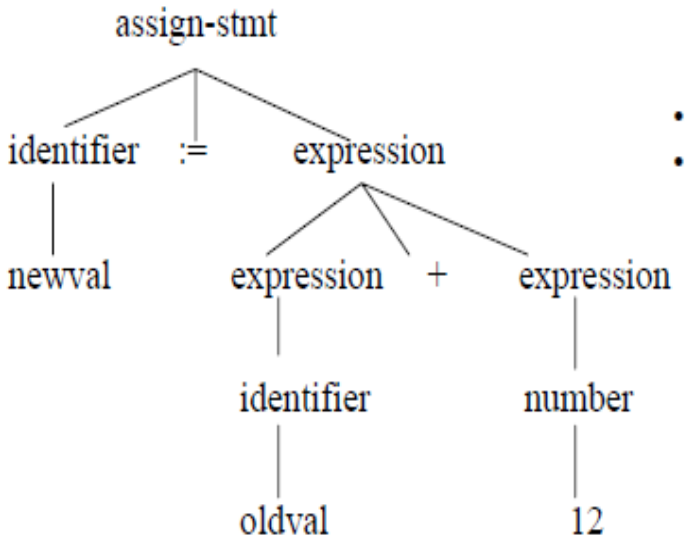
Regular expressions are used to describe tokens (lexical constructs).

A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Syntax Analyzer

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program. Syntax analyzer is also called the **parser.** Its job is to analyze the source program based on the definition of its syntax. It works in lock-step with the lexical analyzer and is responsible for creating a parse-tree of the source code.

Ex: newval := oldval + 12



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a **context free grammar** (CFG). The rules in a CFG are mostly recursive. A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.

– If it satisfies, the syntax analyzer creates a parse tree for the given program.

Semantic Analyzer

A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.

– Type-checking is an important part of semantic analyzer.

Ex: newval := oldval + 12

• The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Synthesis phase

Intermediate Code Generation

An intermediate language is often used by many compiler for analyzing and optimizing the source program. The intermediate language should have two important properties:

– It should be simple and easy to produce.

– It should be easy to translate to the target program

A compiler may produce an explicit intermediate codes representing the source program. These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is close to the level of machine codes.

Ex:

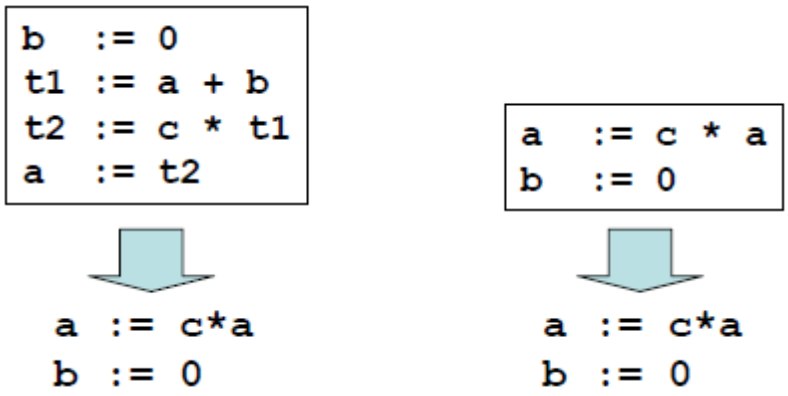
newval := oldval * fact + 12
id1:= id2 * id3 + 12
temp1 = intTofloat(12)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1 = temp3

Code Optimization

The process of removing unnecessary part of a code is known as code optimization. Due to code optimization process it decreases the time and space complexity of the program. i.e Detection of redundant function calls
Detection of loop invariants Common sub-expression elimination Dead code detection and elimination

Ex:

temp1 = intTofloat (12) temp1 = id2 * id3
temp2 = id2 * id3 id1 = temp1 + 12
temp3 = temp1 + temp2
id1 = temp3



Code Generation

This involves the translation of optimized intermediate code into the target language
The target code is normally is a relocatable object file containing the machine or assembly codes.

Ex:

(Assume that we have an architecture with instructions whose at least one of its operands is a machine register)
MOVE id2, R1

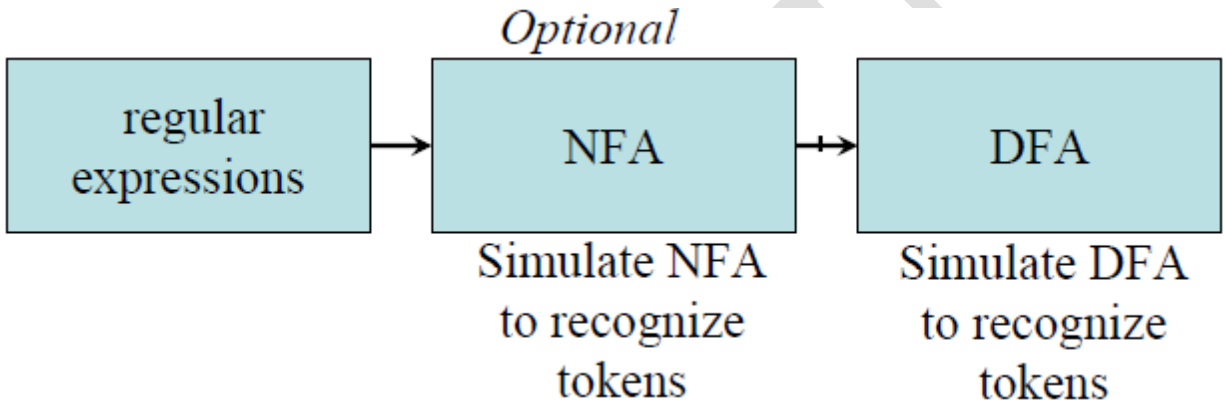
MULT id3, R1
ADD #1, R1
MOVE R1, id1

2.) Design a lexical analyzer generator and explain it.

Ans: First, we define regular expressions for tokens; then we convert them into a DFA to get a lexical analyzer for our tokens.

Algorithm1:
Regular Expression → NFA → DFA (two steps: first to NFA, then to DFA)

Algorithm2:
Regular Expression → DFA (directly convert a regular expression into a DFA)



Subset Construction Algorithm

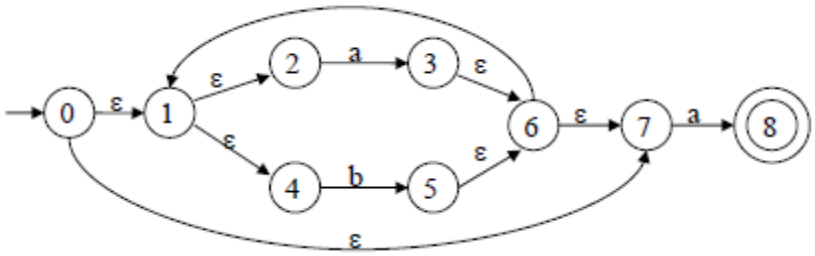
put ε-*closure*(*s*0) as an unmarked state in to *Dstates*
while there is an unmarked state *T* in *Dstates* **do**
mark *T*
for each input symbol *a* ∈ Σ **do**
U = ε-*closure*(*move*(*T*,*a*))
if *U* is not in *Dstates* **then**
add *U* as an unmarked state to *Dstates*
end if
Dtran[*T*,*a*] = *U*
end do
end do

The algorithm produces:

Dstates: Dstates is the set of states of the new DFA consisting of sets of states of the NFA

Dtran: Dtran is the transition table of the new DFA

Subset Construction Example (NFA to DFA)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into *Dstates* as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into *Dstates*

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into *Dstates*

Dtran[S_0,a] ← S_1 *Dtran*[S_0,b] ← S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

Dtran [S_1,a] ← S_1 *Dtran* [S_1,b] ← S_2

⇓ mark S_2

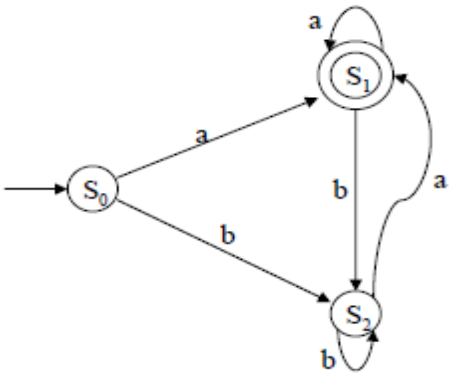
ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

Dtran[S_2,a] ← S_1 *Dtran*[S_2,b] ← S_2

S0 is the start state of DFA since 0 is a member of S0= {0, 1, 2, 4, 7}

S1 is an accepting state of DFA since 8 is a member of S1 = {1, 2, 3, 4, 6, 7, 8}



this is final DFA

Conversion from RE to DFA Example1

Note: - the start state of DFA is firstpos (root)
The accepting states of DFA are all states containing the position of #
For the RE --- (a | b) * a
Its augmented regular expression is;
The syntax tree is:
Now we calculate followpos ,
followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

S_1=*firstpos(root)*={1,2,3}

mark S_1

for a: *followpos(1)* $\cup$ *followpos(3)*={1,2,3,4}=S_2 $move(S_1,a)=S_2$

for b: *followpos(2)*={1,2,3}=S_1 $move(S_1,b)=S_1$

mark S_2

for a: *followpos(1)* $\cup$ *followpos(3)*={1,2,3,4}=S_2 $move(S_2,a)=S_2$

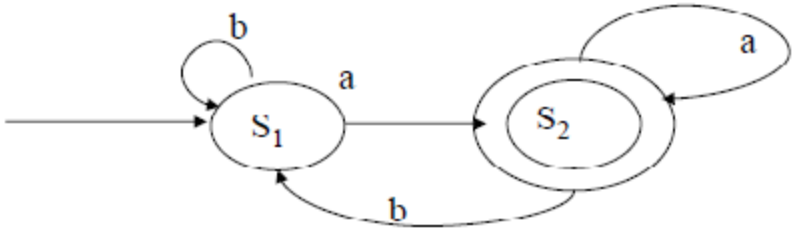
for b: *followpos(2)*={1,2,3}=S_1 $move(S_2,b)=S_1$

Now

start state: S_1

accepting states: {S_2}

Note:- Accepting states=states containing position of i.e. 4.



3.) Differentiate between top-down parsing and bottom-up parsing.

Ans:

Top-Down	Bottom-Up
LL(k) parsers are Top-Down Parsers	LR(k) Parsers are Bottom-Up Parsers
LL(1) is Deterministic	LR(k) Grammars is exactly the set of Deterministic Context-Free Grammars
The way you are most likely familiar with how to parsing grammars	LR(k), for some k, is also LR(1)

- Bottom-up more powerful than top-down;
  - Can process more powerful grammar than LL, will explain later.
- Bottom-up parsers are too hard to write by hand
  - but JavaCUP (and yacc) generates parser from spec;
- Bottom up parser uses right most derivation
  - Top down uses left most derivation;
- Less grammar translation is required, hence the grammar looks more natural;
- Intuition: bottom-up parse postpones decisions about which production rule to apply until it has more data than was available to top-down.

5.) Explain the dynamic programming code generation algorithm with example.

Ans:

6.) What do you mean by code optimization? Explain the basic blocks and their optimization.

Ans: The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives:

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

There are two types of basic block optimizations. They are :

- Ø Structure-Preserving Transformations
- Ø Algebraic Transformations

Structure-Preserving Transformations:
The primary Structure-Preserving Transformation on basic blocks are:

- Ø Common sub-expression elimination
- Ø Dead code elimination
- Ø Renaming of temporary variables
- Ø Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced.

Example:

a: =b+c
b: =a-d
c: =b+c
d: =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d
Basic block can be transformed to

a: = b+c
b: = a-d
c: = a
d: = b

7.) What are the generic issues in the design of code generators? Explain.

Ans:

Input to the code generator:

The input to the code generator is intermediate representation together with the information in the symbol table. What type of input postfix, three-address, dag or tree?

Target Program:

Which one is the out put of code generator: Absolute machine code (executable code), Relocatable machine code (object files for linker), Assembly language (facilitates debugging), Byte code forms for interpreters (e.g. JVM)

Target Machine:

Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.

Instruction Selection:

Instruction selection is important to obtain efficient code.

Register Allocation:

Proper utilization of registers improve code efficiency

Choice of Evaluation order:

The order of computation effect the efficiency of target code.

8.) What are the compiler construction tools? Explain.

Ans: Some commonly used compiler-construction tools. Include:

1. Parser generators.
2. Scanner generators.
3. Syntax-directed translation engines.
4. Automatic code generators.
5. Data-flow analysis engines.
6. Compiler-construction toolkits.

Parser Generators

Input: Grammatical description of a programming language
Output: Syntax analyzers.

Parser generator takes the grammatical description of a programming language and produces a syntax analyzer.

Scanner Generators

Input: Regular expression description of the tokens of a language
Output: Lexical analyzers.
Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language.

Syntax-directed Translation Engines

Input: Parse tree.
Output: Intermediate code.
Syntax-directed translation engines produce collections of routines that walk a parse tree and generates intermediate code.

Automatic Code Generators

Input: Intermediate language.
Output: Machine language.
Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

Data-flow Analysis Engines

Data-flow analysis engine gathers the information, that is, the values transmitted from one part of a program to each of the other parts. Data-flow analysis is a key part of code optimization.

Compiler Construction Toolkits

The toolkits provide integrated set of routines for various phases of compiler. Compiler construction toolkits provide an integrated set of routines for construction of phases of compiler.

9.) Explain the principle sources of code optimization with example.

Ans: We distinguish local transformations—involving only statements in a single basic block—from global transformations. A basic block computes a set of expressions: A number of transformations can be applied to a basic block without changing the expressions computed by the block.

1. Common Sub expressions elimination;

2. Copy Propagation;

3. Dead-Code elimination;

4. Constant Folding.

Common Subexpressions Elimination

- Frequently a program will include calculations of the same value.
- An occurrence of an expression E is called a *common subexpression* if E was previously computed, and the values of variables in E have no changed since the previous computation.
- **Example.** Consider the basic block B_5. The assignments to both t_7 and t_{10} have common subexpressions and can be eliminated.
  After local common subexpression elimination, B_5 is transformed as:

  $t_6 := 4 * i$
  $x := a[t_6]$
  $t_8 := 4 * j$
  $t_9 := a[t_8]$
  $a[t_6] := t_9$
  $a[t_8] := x$
  goto B_2

Common Subexpressions Elimination (Cont.)

- **Example (Cont.)** After local elimination, B_5 still evaluates $4 * i$ and $4 * j$ which are common subexpressions.
- $4 * j$ is evaluated in B_3 by t_4. Then, the statements
  $$t_8 := 4 * j;\ t_9 := a[t_8];\ a[t_8] := x$$
  can be replaced by
  $$t_9 := a[t_4];\ a[t_4] := x$$
- Now, $a[t_4]$ is also a common subexpression, computed in B_3 by t_5. Then, the statements
  $$t_9 := a[t_4];\ a[t_6] := t_9$$
  can be replaced by
  $$a[t_6] := t_5.$$
- Analogously, the value of x is the same as the value assigned to t_3 in block B_3; while t_6 can be eliminated and replaced by t_2.

Dead-Code Elimination

- A variable is *live* at a point in a program if its value can be used subsequently, otherwise it is *dead*.
- A piece of code is *dead* if data computed is never used elsewhere.
- Dead-Code may appear as the result of previous transformation.
- Dead-Code works well together with Copy Propagation.
- **Example.** Considering the Block B_5 after Copy Propagation we can see that `x` is never reused all over the code. Thus, `x` is a dead variable and we can eliminate the assignment `x` := $\texttt{t}_3$ from B_5.

Copy Propagation

- **Copy Propagation Rule**: Given the *copy statement* `x := y` use `y` for `x` whenever possible after the copy statement.
- Copy Propagation applied to Block B_5 yields:

  $x := t_3$
  $a[t_2] := t_5$
  $a[t_4] := t_3$
  goto B_2

- This transformation together with Dead-Code Elimination (see next slide) will give us the opportunity to eliminate the assignment `x` := t_3 altogether.

10.) Differentiate between C compiler and Pascal compiler.

Ans:

C Language	Pascal Language
C language was found by **Dennis Ritchie** in **1972**.	Pascal language was found by **Niklaus Wirth** in **1969**. Name of this language is kept Pascal in the honor of 'one of the great french mathematician & philosopher named "**Blaise Pascal**"'.
C language is influenced by **ALGOL 68, BCPL, Assembly , Fortran, PL/I**. etc.	This Language was influenced by **ALGOL 60.**
In C language, **semicolon (;)** is used as **statement terminator**.	In Pascal language, semicolon (;) is used as **statement separator**.
/* comment */, // comment, are used for comments.	**{comment}** – curly braces and **(* comment *)** are used for comments.
return, break and continue can be used in C.	return, **break and continue** is not used in Pascal.

C language is **case-sensitive.**	Pascal language is **not** case sensitive.
C does not have **Boolean** data type but have **relational operators**.	Pascal have **Boolean** data type.
To define **constant** in C, #define is used.For ex : #define PI=3.14 ;	To define **constant** in Pascal, **constant** is used.For ex : constant PI = 3.14
Variable declaration in c :For ex : int x; int x , y ;	**Variable** declaration in PascalFor ex : var x : integer ; var x, y : integer;
Pointer variable declaration in C :For ex : int *x;	Pointer variable declaration in Pascal :For ex : x:^integer
In C language, no such specific **keyword** is used in beginning or ending but **"{"** and **"}"** are used for block of statements.	In Pascal, program starts with the keyword '**program**' and then follows **begin** and **end** keywords.

Tribhuvan University
Institute of Science and Technology
Bachelor of Computer Science and Information Technology
Course Title: Compiler Design and Construction
2071

1.) Explain the various phases of compiler in detail with practical example.

Ans: View in 2071(II) q no. 1

2.) Explain about design of lexical analyzer generator with its suitable diagram.

Ans: View in 2071(II) q no. 2

3.) What are the problem with top down parsers ? Explain the LR parsing algorithm.

Ans: **Problems with the Top-Down Parser**

1. Only judges grammatically

2. Stops when it finds a single derivation.

3. No semantic knowledge employed.

4. No way to rank the derivations.

5. Problems with left-recursive rules.

6. Problems with ungrammatical sentences.

LR PARSER

In computer science, **LR parsers** are a type of bottom-up parser that efficiently handle deterministic context-free languages in guaranteed linear time. The **LALR** parsers and the **SLR** parsers are common variants of LR parsers. LR parsers are often mechanically generated from a formal grammar for the language by a parser generator tool. They are widely used for the processing of computer languages.

The name **LR** is an initialism. The **L** means that the parser reads input text in one direction without backing up; that direction is typically **L**eft to right within each line, and top to bottom across the lines of the full input file. (This is true for most parsers.) The **R** means that the parser produces a **R**ightmost derivation in reverse: it does a bottom-up parse - not a top-down LL parse or ad-hoc parse. The name LR is often followed by a numeric qualifier, as in **LR(1)** or sometimes **LR(*k*)**. To avoid backtracking or guessing, the LR parser is allowed to peek ahead at *k*lookahead input symbols before deciding how to parse earlier symbols. Typically *k* is 1 and is not mentioned. The name LR is often preceded by other qualifiers, as in **SLR** and **LALR**.

LR parsers are deterministic; they produce a single correct parse without guesswork or backtracking, in linear time. This is ideal for computer languages, but LR parsers are not suited for human languages which need more flexible but inevitably slower methods. Some methods which can parse arbitrary context-free grammars (e.g., Cocke-Younger-Kasami, Earley, GLR) have worst-case performance of $O(n^3)$ time. Other methods which backtrack or yield multiple parses may even take exponential time when they guess badly.

The above properties of **L**, **R**, and **k** are actually shared by all shift-reduce parsers, including precedence parsers. But by convention, the LR name stands for the form of parsing invented by Donald Knuth, and excludes the earlier, less powerful precedence methods (for example Operator-precedence parser).LR parsers can handle a larger range of languages

and grammars than precedence parsers or top-down LL parsing.This is because the LR parser waits until it has seen an entire instance of some grammar pattern before committing to what it has found. An LL parser has to decide or guess what it is seeing much sooner, when it has only seen the leftmost input symbol of that pattern. LR is also better at error reporting. It detects syntax errors as early in the input stream as possible.

5.) What are the different issues in the design of code generator ? Explain with example about the optimization of basic blocks.

Ans: **Input to the code generator:**
The input to the code generator is intermediate representation together with the information in the symbol table. What type of input postfix, three-address, dag or tree?

Target Program:
Which one is the out put of code generator: Absolute machine code (executable code), Relocatable machine code (object files for linker), Assembly language (facilitates debugging), Byte code forms for interpreters (e.g. JVM)

Target Machine:
Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.

Instruction Selection:
Instruction selection is important to obtain efficient code.

Register Allocation:
Proper utilization of registers improve code efficiency

Choice of Evaluation order:
The order of computation effect the efficiency of target code.

There are two types of basic block optimizations. They are:

- Ø Structure-Preserving Transformations
- Ø Algebraic Transformations

Structure-Preserving Transformations:
The primary Structure-Preserving Transformation on basic blocks are:

- Ø Common sub-expression elimination
- Ø Dead code elimination
- Ø Renaming of temporary variables
- Ø Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced.

Example:

a: =b+c
b: =a-d
c: =b+c
d: =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d
Basic block can be transformed to

a: = b+c
b: = a-d
c: = a
d: = b

6.) What are the main issues involved in designing lexical analyzer? Mention the various error recovery strategies for a lexical analyzer.

Ans:

7.) Define a context free grammar. What are the component of context free grammar? Explain.

Ans:

In formal language theory, a **context-free grammar** (**CFG**) is a certain type of formal grammar: a set of production rules that describe all possible strings in a given formal language. Production rules are simple replacements. For example, the rule

$$A \to \alpha$$

replaces A with α. There can be multiple replacement rules for any given value. For example,

$$A \to \alpha$$

$$A \to \beta$$

means that A can be replaced with either α or β.

A context-free grammar (CFG) is a set of recursive rewriting rules (or *productions*) used to generate patterns of strings.

A CFG consists of the following components:

- a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
- a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
- a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

8.) What are the various issues of code generator ? Explain the benefits of intermediate code generation.

Ans:

: **Input to the code generator:**

The input to the code generator is intermediate representation together with the information in the symbol table. What type of input postfix, three-address, dag or tree?

Target Program:

Which one is the out put of code generator: Absolute machine code (executable code), Relocatable machine code (object files for linker), Assembly language (facilitates debugging), Byte code forms for interpreters (e.g. JVM)

Target Machine:

Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.

Instruction Selection:

Instruction selection is important to obtain efficient code.

Register Allocation:

Proper utilization of registers improve code efficiency

Choice of Evaluation order:

The order of computation effect the efficiency of target code.

9.) Explain the peephole organization. Write a three address code for the expression

r; = 7*3+9.

Ans: A statement-by-statement code-generations strategy often produces 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 be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristics of peephole optimizations:

Redundant-instructions elimination
Flow-of-control optimizations
Algebraic simplifications
Use of machine idioms
Unreachable

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R0,a
(2) MOV a,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).

Flows-Of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1
….
L1: gotoL2 (d)

by the sequence

goto L2
….
L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1

….

L1: goto L2 (e)

can be replaced by

If a < b goto L2

….

L1: goto L2

Ø Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

L1: if a < b goto L2 (f) L3:

may be replaced by

If a < b goto L2

goto L3

…….

L3:

While the number of instructions in(e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) in execution time

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 or

x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Use of Machine Idioms:

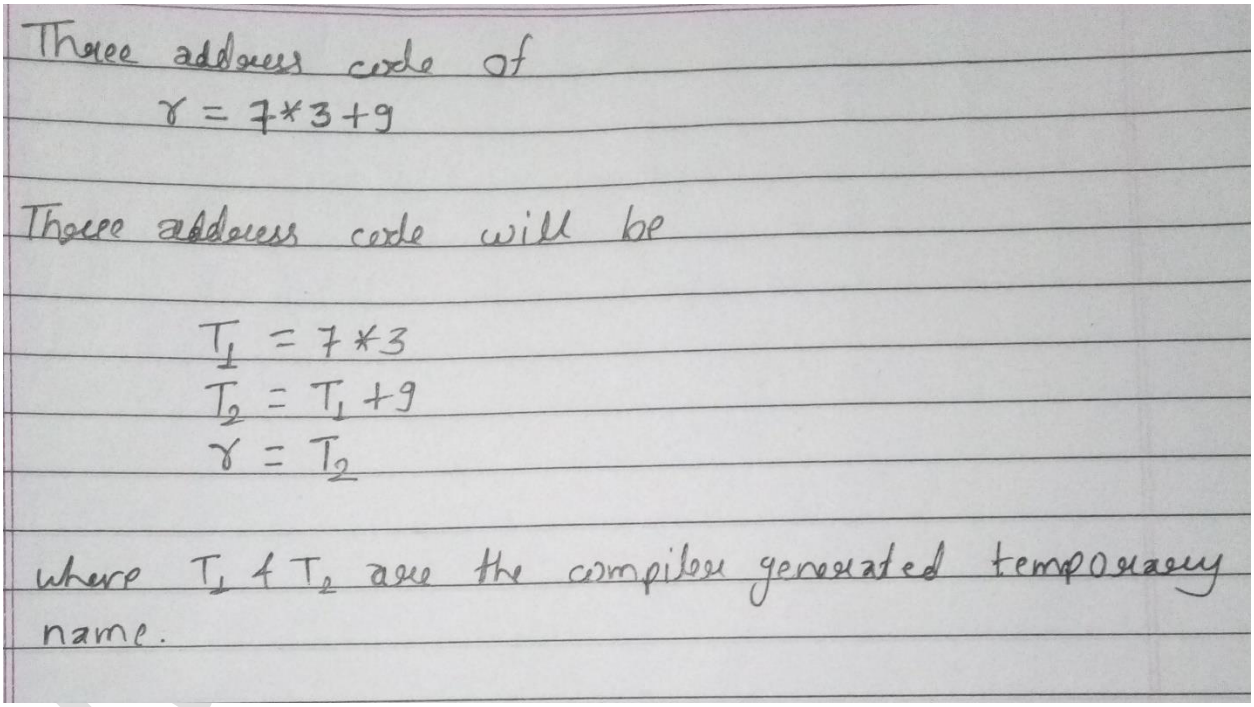
The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment

and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 → i++
i:=i-1 → i- -

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1.



10.) Differentiate between Pascal compiler and C++ compiler.

Ans: Pascal is a language indeed. At least it is easier to read than C++.

Besides readability, two main differences: C++ has been more portable than Pascal, because implementations differed less. C++ has become an ANSI standard well before Pascal was standardized, which also helped portability a lot.

Secondly, there is nothing you can do in assembly but not in C because the language syntax precludes this. C++ allows much more low-level programming than Pascal, as the Pascal

syntax simply doesn't allow some constructs.

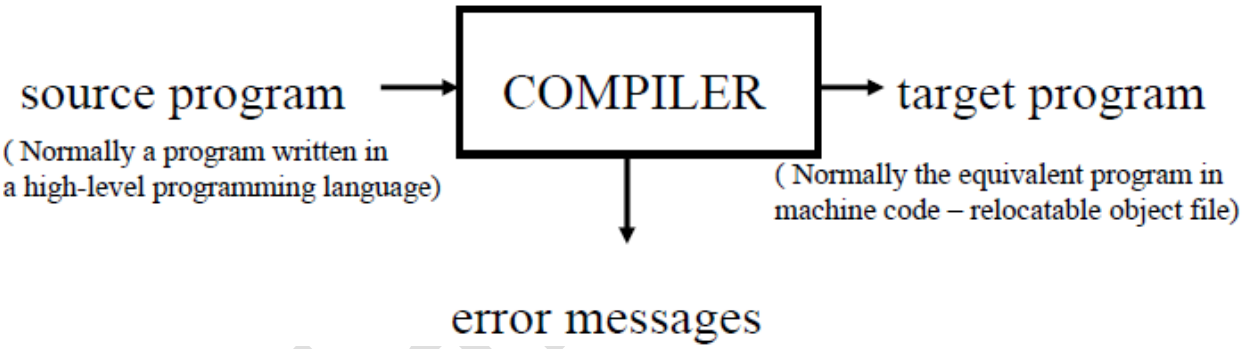
C++ is also well standardized, while I am not even sure if an Object Pascal standardization exists.

If you want to write portable programs, or programs a large community is able to maintain, use C. If you just program for fun, no system programming and educational purposes, Pascal has not many drawbacks.

Tribhuvan University
Institute of Science and Technology
Bachelor of Computer Science and Information Technology
Course Title: Compiler Design and Construction
2069

1.) What do you mean by compiler? Explain the semantic analysis phase of compiler construction.

Ans: : A compiler is a program that takes a program written in a *source language* and translates it into an equivalent low level program in a *target language*.



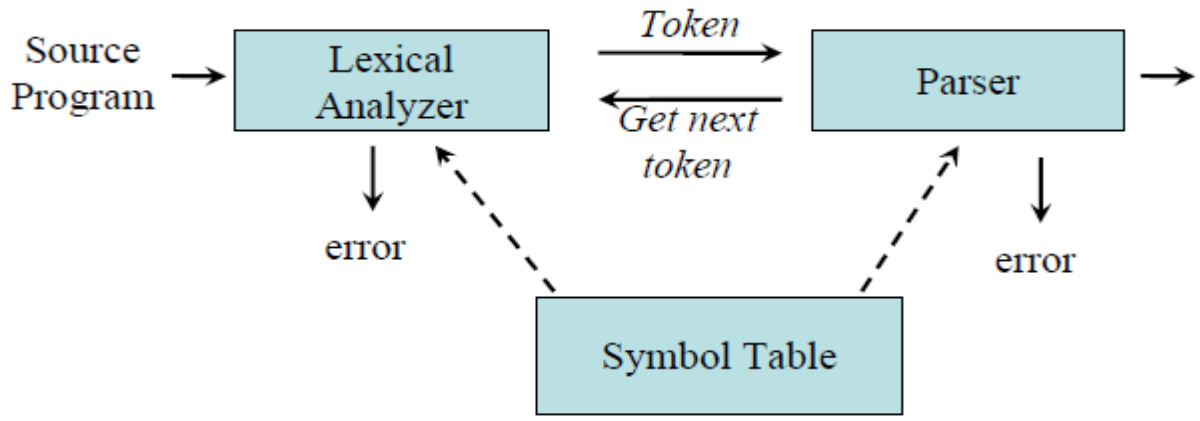
Semantic Analysis Phae:

Lexical Analyzer

Lexical Analyzer reads the source program in character by character ways and returns the *tokens* of the source program.

Normally a lexical analyzer doesn't return a list of tokens, it returns a token only when the parser asks a token from it.

Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc.



Example:

newval := oldval + 12

tokens :	newval	identifier
	:=	assignment operator
	Oldval	identifier
	+	add operator
	12	a number

Put information about identifiers into the symbol table.

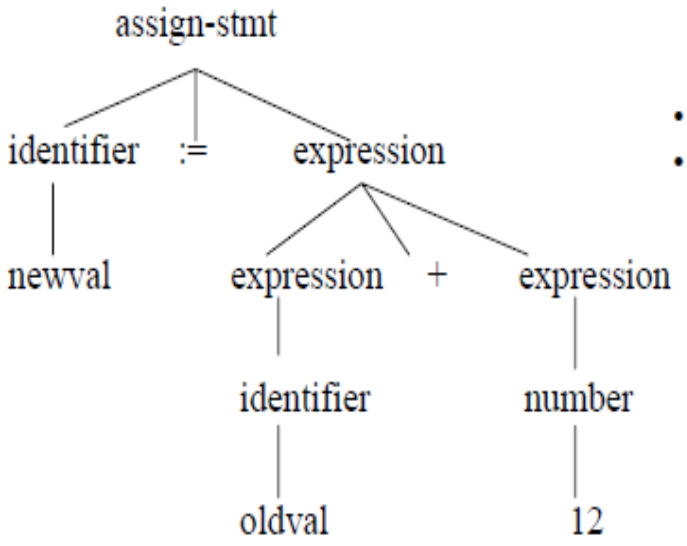
Regular expressions are used to describe tokens (lexical constructs).

A (Deterministic) Finite State Automaton (DFA) can be used in the implementation of a lexical analyzer.

Syntax Analyzer

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program. Syntax analyzer is also called the **parser.** Its job is to analyze the source program based on the definition of its syntax. It works in lock-step with the lexical analyzer and is responsible for creating a parse-tree of the source code.

Ex: newval := oldval + 12



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

The syntax of a language is specified by a context free grammar (CFG). The rules in a CFG are mostly recursive. A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.

– If it satisfies, the syntax analyzer creates a parse tree for the given program.

Semantic Analyzer

A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.

– Type-checking is an important part of semantic analyzer.

Ex: newval := oldval + 12

• The type of the identifier *newval* must match with type of the expression *(oldval+12)*

2.) Why are regular expressions used in token specification? Write the regular expression to specify the identifier like in C.

Ans:

Regular expressions **are notation for specifying patterns.**

- **Each *pattern* matches a set of strings.**
- ***Regular expressions*** **will serve as names for sets of strings.**
- ***Strings and Languages*:**
- **The term *alphabet or character class* denotes any finite set of symbols.**
- **e.g., set {0,1} is the binary alphabet.**
- **The term *sentence and word* are often used as synonyms for the term string.**
- **The *length of a string s* is written as |s| - is the number of occurrences of symbols in *s*.**
- **e.g., string "banana" is of length six.**
- **The *empty string* denoted by ε – length of empty string is zero.**

- **The term *language* denotes any set of strings over some fixed alphabet.**
- **e.g., {ε} – set containing only empty string is language under φ.**
- **If *x* and *y* are strings, then the *concatenation* of *x* and *y* (written as *xy*) is the string formed by appending *y* to *x*. *x* = *dog* and *y* = *house*; then *xy* is *doghouse*.**
- ***sε* = *εs* = *s*.**
- ***s*^{0} = ε, *s*^{1} = *s*, *s*^{2} = *ss*, *s*^{3} = *sss*, … so on.**

4.) Consider the grammar :

S ⟶ aSbS | bSaS | ε

a.)Show that this grammar is ambiguous by constructing two different leftmost derivations for sentence abab.

b.) Construct the corresponding rightmost derivations for abab.

c.)Construct the corresponding parse trees for abab.

Ans:

$S \rightarrow aSbS \mid bSaS \mid \epsilon$

a) Showing ambigious sentence abab

$S \rightarrow aSbS$
$\rightarrow abS \rightarrow abaSbS \rightarrow ababS$
$\rightarrow abab$

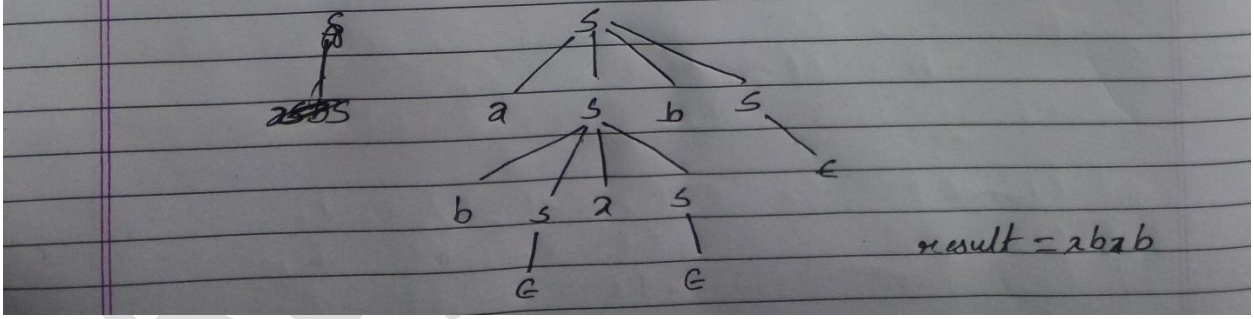
$S \rightarrow aSbS$
$\rightarrow abSaSbS$
$\rightarrow abaSbS$
$\rightarrow ababS$
$\rightarrow abab$

Since the sentence have same result from different way so the grammar is ambigious.

b) constructing the corresponding eighthmost derivation for abab.

$S \rightarrow aSbS \rightarrow aSbaSbS \rightarrow aSbaSb \rightarrow aSbab$
$\rightarrow abab$

c) Parse tree:



5.) Consider the grammar :

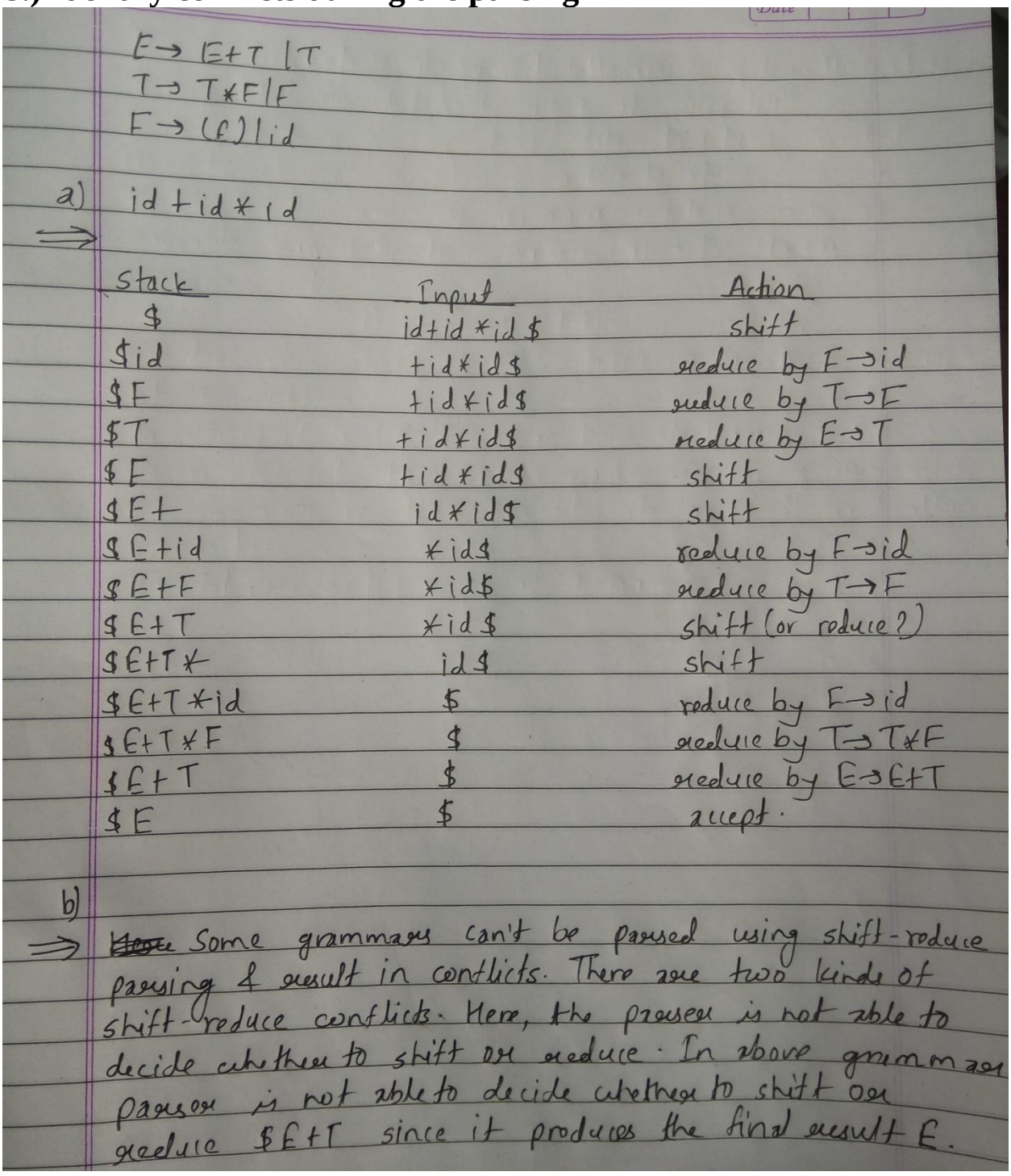
E→E+T | T

T→T*F|F

F→ (E) | id

a.) Show steps of shift-reduce parsing for the input string id+id*id.

b.) Identify conflicts during the parsing



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

a) id + id * id

⇒

Stack	Input	Action
$	id+id*id$	shift
$id	+id*id$	reduce by F→id
$F	+id*id$	reduce by T→F
$T	+id*id$	reduce by E→T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F→id
$E+F	*id$	reduce by T→F
$E+T	*id$	shift (or reduce?)
$E+T*	id$	shift
$E+T*id	$	reduce by F→id
$E+T*F	$	reduce by T→T*F
$E+T	$	reduce by E→E+T
$E	$	accept.

b)

⇒ Some grammars can't be parsed using shift-reduce parsing & result in conflicts. There are two kinds of shift-reduce conflicts. Here, the parser is not able to decide whether to shift or reduce. In above grammar parser is not able to decide whether to shift or reduce $E+T since it produces the final result E.

6.) Describe the L-attributed definitions. How L-attributed definitions are evaluated?

Ans:

L-Attributed Definitions

An L-attribute is an inherited attribute which can be evaluated in a left-to-right fashion. L-attributed definitions can be evaluated using a *depth-first* evaluation order. (L –> attributes flow from left to right)

More precisely, a syntax-directed definition is *L-attributed* if each inherited attribute of X_j on the right side of $A \rightarrow X_1 X_2 \ldots X_n$ depends only on

1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$ to the left of X_j in the production and
2. the inherited attributes of A

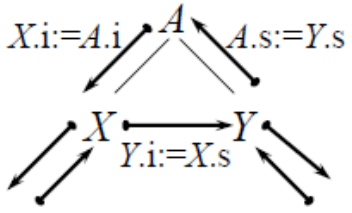
Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

Evaluation of L-Attributed Definitions

L-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

```
Procedure dfvisit(n: node);          //depth-first evaluation
        for each child m of n, from left to right do
                evaluate inherited attributes of m;
                dfvisit(m);
        evaluate synthesized attributes of n
```

$A \rightarrow X\,Y$



$X.\mathrm{i} := A.\mathrm{i}$
$Y.\mathrm{i} := X.\mathrm{s}$
$A.\mathrm{s} := Y.\mathrm{s}$

9.) Discuss the issues in design of simple code generator.

Ans: View in 2071(II) q no. 7

10.) Define the following optimization techniques:
a.) Unreachable code elimination
b.) Flow-of-control optimization

Ans:

a.) **Flows-Of-Control Optimizations:**

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

goto L1
….
L1: gotoL2 (d)

by the sequence

goto L2
….
L1: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1
….
L1: goto L2 (e)

can be replaced by

If a < b goto L2
….
L1: goto L2

Ø Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1
L1: if a < b goto L2 (f) L3:

may be replaced by

If a < b goto L2
goto L3
…….

L3:

While the number of instructions in(e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) in execution time

b.) **Unreachable Code:**

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1.

Tribhuvan University
Institute of Science and Technology
Bachelor of Computer Science and Information Technology
Course Title: Compiler Design and Construction
2068

1. Explain the phase of a compiler with block diagram. (6)

Ans: View in 2071(II) q no. 1

2. Define token, pattern and lexeme with suitable example. How input buffering can be implemented for scanner, explain.

Ans: -A *token* is a logical building block of language. They are the sequence of characters having a collective meaning.

Eg: identifier, keywords etc

-A sequence of input characters that make up a single token is called a lexeme.

A token can represent more than one lexeme.

Eg: abc, 12 etc

In the statement

Float pi=3.1415

The variable pi is called a lexeme for the token 'identifier'

-Patterns are the rules for describing whether a given lexeme belonging to a token or not.

Regular expressions are widely used to specify patterns.

Input Buffering:

Many times, a scanner has to look ahead several characters from the current character in order to recognize the token.

For example *int* is keyword in C, while the term *inp* may be a variable name. When the character *'i'* is encountered, the scanner cannot decide whether it is a keyword or a variable name until it reads two more characters.
In order to efficiently move back and forth in the input stream, input buffering is used.

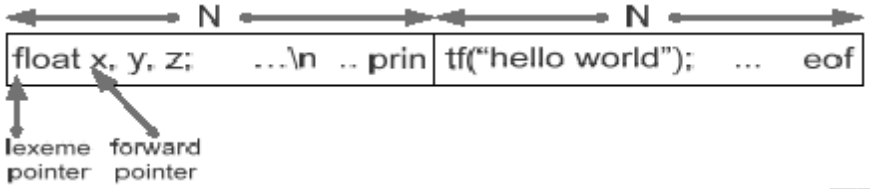


Fig: - An input buffer in two halves

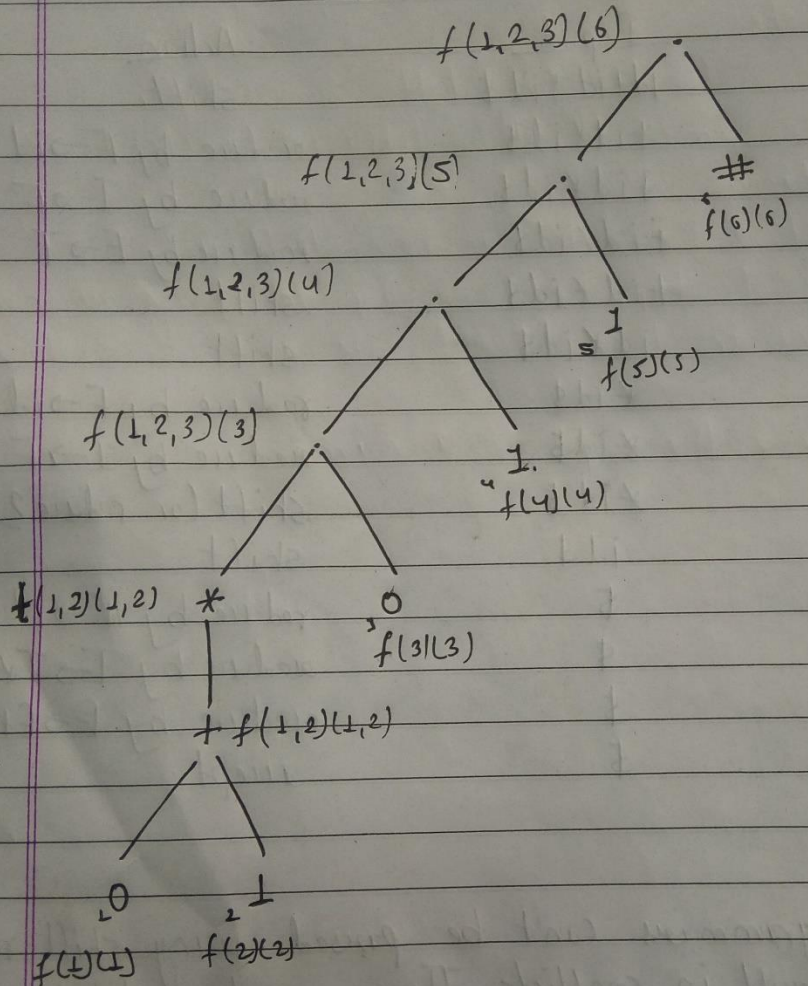
Here, we divide the buffer into two halves with N-characters each.
Rather than reading character by character from file we read N input character at once. If there are fewer than N characters in input eof marker is placed.
There are two pointers (see in above fig.) the portion between lexeme pointer and forward pointer is current lexeme. Once the match for pattern is found, both the pointers points at the same place and forward pointer is moved.

3. Give the regular expression (0+1)*011, construct a DFA equivalent to this regular expression computing follow pos ().
Ans:

$r = (0+1)^* 0 1 1 \#$

$r = (0+1)^* 0 1 1 \#$
$\quad\quad 1\ 2\ 3\ 4\ 5\ 6$

Constructing syntax tree as:



$S_0 = \text{firstpos(root)}$
$= (1, 2, 3)$

$\delta(S_0, 0) = \text{followpos}(1) \cup \text{followpos}(2) \cup \text{followpos}(3)$
$= \{1, 2, 3\} \cup \{1, 2, 3\} \cup \{4\}$
$= \{1, 2, 3, 4\} = S_1$

$\delta(S_0, 1) = \text{followpos}(2)$
$= \{1,2,3\} = S_0$

$\delta(S_1, 0) = \text{followpos}(1) \cup \text{followpos}(3)$
$= S_1$

$\delta(S_1, 1) = \text{followpos}(2) \cup \text{followpos}(4)$
$= \{1,2,3\} \cup \{5\}$
$= \{1,2,3,5\} = S_2$

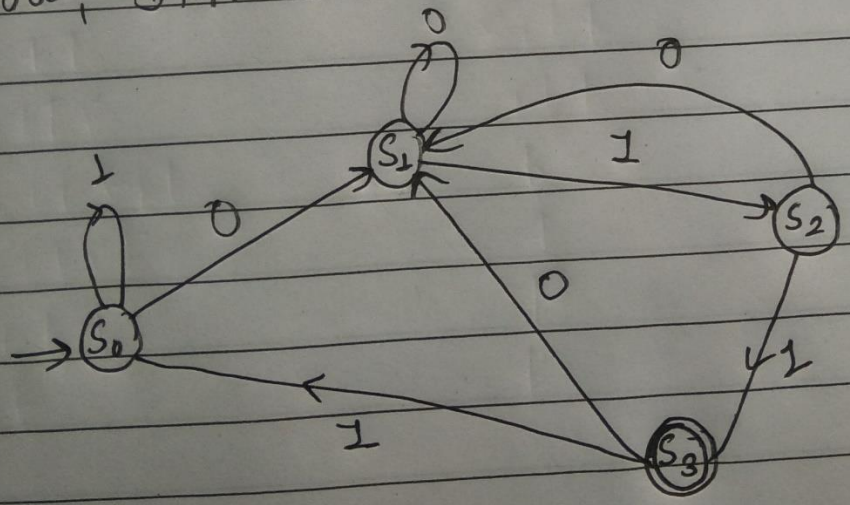
$\delta(S_2, 0) = \text{followpos}(1) \cup \text{followpos}(3) = S_1$

$\delta(S_2, 1) = \text{followpos}(2) \cup \text{follow}(5)$
$= \{1,2,3\} \cup \{6\} = \{1,2,3,6\} = S_3$

$\delta(S_3, 0) = S_1$

$\delta(S_3, 1) = \text{followpos}(2) = S_0$

Now, DFA will be as,



4. Explain the role of the parser. Write an algorithm for non-recursive predictive pursing. (6)

Ans: **The Role of a Parser:** The second phase of the compilation process is syntax analysis commonly known as parsing. A parser obtains the tokens from the lexical analyzer and analyzes syntactically according to the grammar of the source language whether the string can be generated or not from the grammar i.e. the parser works with the lexical analyzer as shown in figure below.

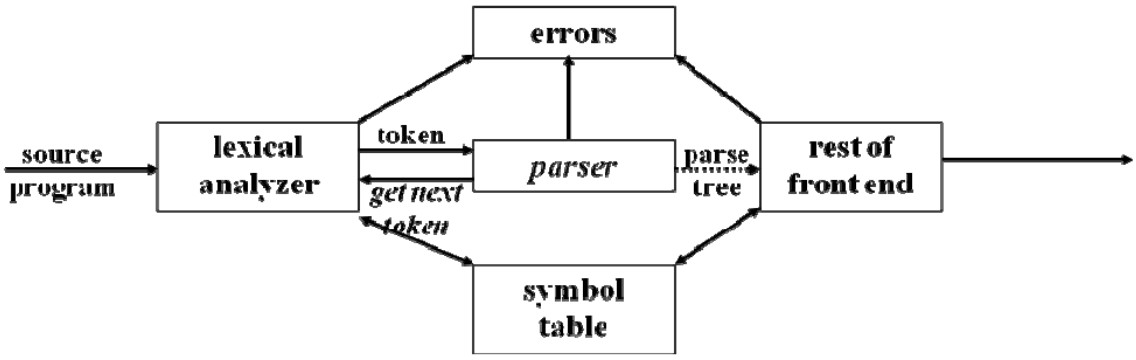


Figure: Position of parser in compiler model

A syntax analyzer (parser) is to analyze the source program based on the definition of its syntax. It works in lock-up step with the lexical analyzer (scanner) and responsible for creating a parse tree out of the source code.

- A parser implements a Context Free Grammar.
- Besides the checking of syntax the parser is responsible to report the syntax errors.
- A parser is also responsible to invoke semantic actions
  - for static semantics checking e.g. type checking of expressions, functions etc
  - for syntax directed translation of the source code to an intermediate representation
  - The possible intermediate representations outputs are

♉ Abstract syntax tree

♉ control-flow graphs (CFGs) with triples, three address code or register transfer list notations

Algorithm

Input : a string w.
Output: if w is in L(G), a leftmost derivation of w; otherwise error

1. Set *ip* to the first symbol of input stream
2. Set the stack to *$S* where *S* is the start symbol of the grammar
3. repeat
   Let *X* be the top stack symbol and *a* be the symbol pointed by *ip*
   If *X* is a terminal or *$* then
      if *X* = *a* then pop *X* from the stack and advance *ip*
      else error()
   else /* *X* is a non-terminal */
      if *M[X, a]* = *X* → *Y1, Y2, ..., ..., Yk* then
         pop *X* from stack
         push *Yk, Yk-1, ..., ..., Y1* onto stack (with *Y1* on top)
         output the production *X* → *Y1, Y2, ..., Yk*
      else error()
4. until *X* = *$* /* stack is empty */

5. Construct the grammar

E → E+T|T

T → T*F|F

F→ (E) |id

Compute the complete LR(0) collection of item set from above grammar.

Ans:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Augumented grammar,

0. $E' \rightarrow E$
1. $E \rightarrow E+T$
2. $E \rightarrow T$
3. $T \rightarrow T*F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

$I_0 = closure(E' \rightarrow .E)$

$= \{$

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id \}$

$goto(I_0, E) = closure(E' \rightarrow E.)$

$= \{$

$E' \rightarrow E.$

$\} = I_1$

$goto(I_0, T) = closure(E \rightarrow T.) \cup closure(T \rightarrow T.*F)$

$\{$

$E \rightarrow T. \quad \cup \quad T \rightarrow T.*F$

$\} = I_2$

goto(I_0, F) = closure($T \to F.$)

$\{$

$T \to F.$

$\}$ = I_3

goto(I_0, () = closure($F \to (.E)$)

$\{$

$F \to (.E)$

$E \to .E+T$, $E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$ $\}$ = I_4

goto(I_0, id) = closure($F \to id.$)

$\{$

$F \to id.$

$\}$ = I_5

goto(I_2, *) = closure($T \to T*.F$)

$\{$

$T \to T*.F$

$F \to .(E)$

$F \to .id$

$\}$ = I_6

goto(I_4, E) = closure($F \to (E.)$) $\cup$ closure($E \to E.+T$)

$\{$

$F \to (E.)$

$E \to E.+T$

$\}$ = I_7

goto(I_4, T) = closure($E \to T.$) ∪ closure($T \to T.*F$)

{

$E \to T.$

$T \to T*.F$

$F \to .(E)$

$F \to .id$

} = I_8

goto(I_4, F) = closure($T \to F.$) = I_3

goto(I_4, () = closure($F \to (.E)$) = I_4

goto(I_4, id) = closure($F \to id.$) = I_5

goto(I_6, F) = closure($T \to T*F.$)

{

$T \to T*F.$

} = I_9

goto(I_6, () = closure($F \to (.E)$) = I_4

goto(I_6, id) = closure($F \to id.$) = I_5

goto(I_7,)) = closure($F \to (E).$)

{

$F \to (E).$

} = I_{10}

goto (I_7, +) = closure (E → E+.T)

{

E → E+.T

T → .T*F

T → .F

F → .(E)

F → .id

} = I_{11}

goto(I_8, F) = closure (T → T*F.)

{

T → T*F.

} = I_{12}

goto(I_8, ( ) = I_4

goto(I_8, id) = I_5

goto(I_{11}, T) = closure (E → E+T.) ∪ closure (T → T.*F)

{

E → E+T.

T → T.*F

} = I_{13}

goto(I_{11}, F) = closure (T → F.) = I_3

goto(I_{11}, ( ) = I_4

goto(I_{11}, id) = I_5

goto(I_{13}, *) = closure (T → T*.F)

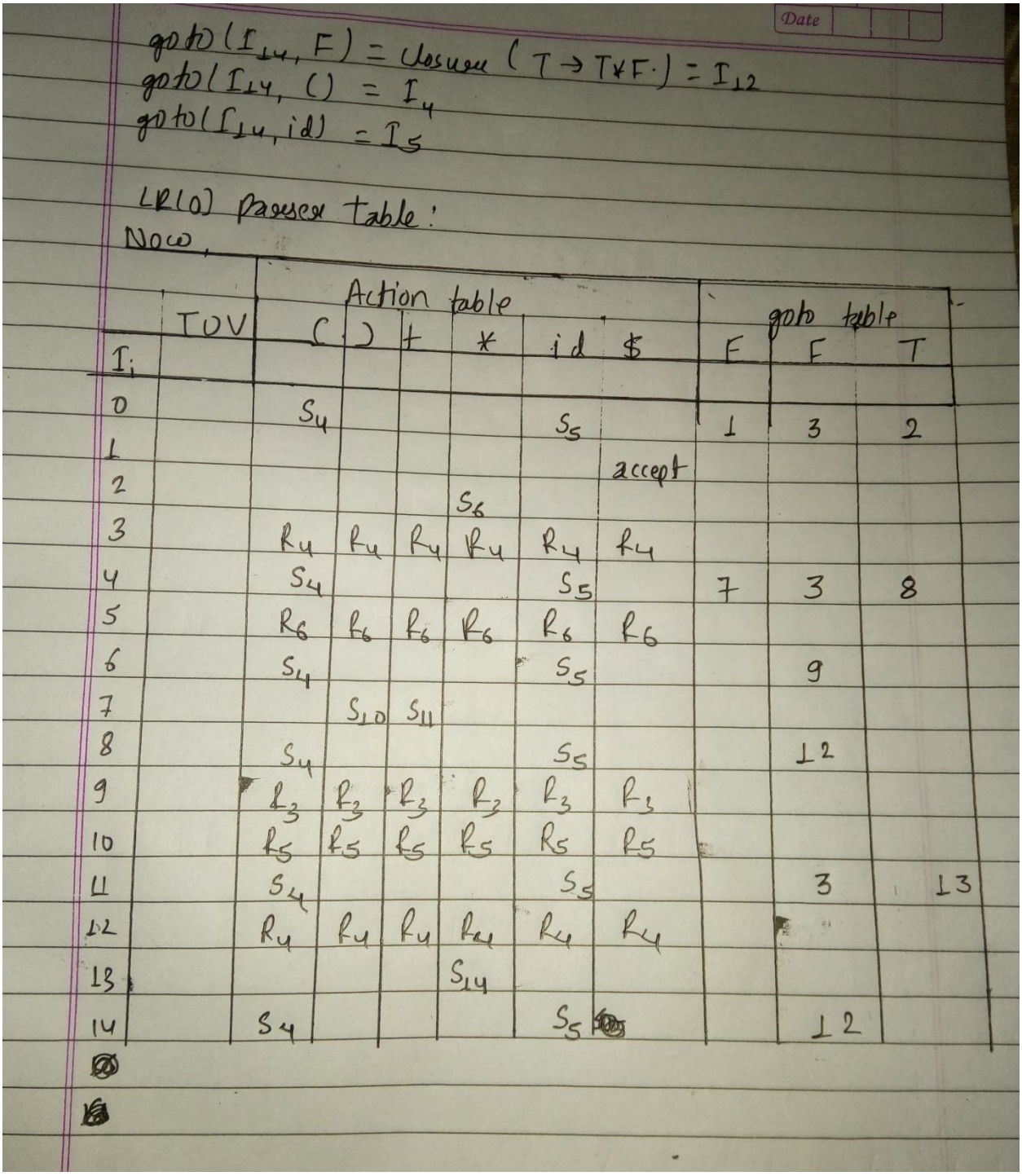
{

T → T*.F

F → .(E)

F → .id

} = I_{14}



goto(I_{14}, F) = closure($T \rightarrow T * F.$) = I_{12}
goto(I_{14}, () = I_4
goto(I_{14}, id) = I_5

LR(0) Parser table:
Now,

	Action table						goto table		
	TOV								
I_i	(	)	+	*	id	$	E	F	T
0	S_4				S_5		1	3	2
1						accept			
2				S_6					
3	R_4	R_4	R_4	R_4	R_4	R_4			
4	S_4				S_5		7	3	8
5	R_6	R_6	R_6	R_6	R_6	R_6			
6	S_4				S_5			9	
7		S_{10}	S_{11}						
8	S_4				S_5			12	
9	R_3	R_3	R_3	R_3	R_3	R_3			
10	R_5	R_5	R_5	R_5	R_5	R_5			
11	S_4				S_5			3	13
12	R_4	R_4	R_4	R_4	R_4	R_4			
13				S_{14}					
14	S_4				S_5			12	

8. What do you mean by S-attributed definition and how they are evaluated? Explain with example.

Ans:

S-Attributed Definitions

A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition* (or *S-attributed grammar*)

A parse tree of an S-attributed definition is annotated by evaluating the semantic rules for the attribute at each node bottom-up

Yacc/Bison only support S-attributed definitions

Bison Example

```
%token DIGIT
%%
L : E '\n'          { printf("%d\n", $1); }
  ;
E : E '+' T         { $$ = $1 + $3; }
  | T               { $$ = $1; }
  ;
T : T '*' F         { $$ = $1 * $3; }
  | F               { $$ = $1; }
  ;
F : '(' E ')'       { $$ = $2; }
  | DIGIT           { $$ = $1; }
  ;
%%
```

Synthesized attribute of parent node **F**

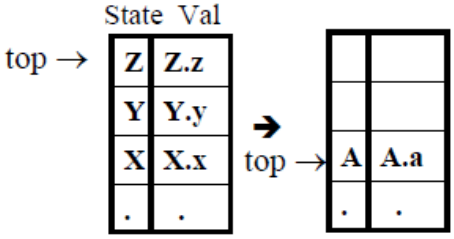
Bottom-up Evaluation of S-Attributed Definitions

An S-attributed grammar can be translated bottom-up as and when the grammar is being parsed using an LR parser.

The LR parser's stack can be extended to hold not only a grammar symbol, but also its semantic attribute as well.

Hence *action[s,a] = shift s'* becomes *action[s,a] = shift a.*$$, s'*

$A \rightarrow XYZ$ $A.a=f(X.x,Y.y,Z.z)$ where all attributes are synthesized.



When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry will hold the synthesized attribute(s) of the symbol X.

the values of the attributes are evaluated during reductions.

Bottom-up Evaluation of S-Attributed Definitions in Yacc

Input: **3*5+4n**

Grammar in Page 14

Stack	val	Input	Action	Semantic Rule
$	_	**3*5+4n$**	shift	
$ 3	*3*	***5+4n$**	reduce *F* → **digit**	$$ = $1
$ *F*	*3*	***5+4n$**	reduce *T* → *F*	$$ = $1
$ *T*	*3*	***5+4n$**	shift	
$ *T* *****	*3* _	**5+4n$**	shift	
$ *T* *** 5**	*3* _ *5*	**+4n$**	reduce *F* → **digit**	$$ = $1
$ *T* ***** *F*	*3* _ *5*	**+4n$**	reduce *T* → *T* * *F*	$$ = $1 * $3
$ *T*	*15*	**+4n$**	reduce *E* → *T*	$$ = $1
$ *E*	*15*	**+4n$**	shift	
$ *E* **+**	*15* _	**4n$**	shift	
$ *E* **+ 4**	*15* _ *4*	**n$**	reduce *F* → **digit**	$$ = $1
$ *E* **+** *F*	*15* _ *4*	**n$**	reduce *T* → *F*	$$ = $1
$ *E* **+** *T*	*15* _ *4*	**n$**	reduce *E* → *E* + *T*	$$ = $1 + $3
$ *E*	*19*	**n$**	shift	
$ *E* **n**	*19* _	**$**	reduce *L* → *E* **n**	*print* $1
$ *L*	*19*	**$**	accept	

9. What do you mean by three-code representation? Explain with example.

Ans:

Three-Address Code (Quadruples)

A quadruple is: $x = y$ ***op*** z

where x, y and z are names, constants or compiler-generated temporaries and ***op*** is any operator. (only one operator on the right side of the statement)

Postfix notation (much better notation because it looks like a machine code instruction)

op y,z,x apply operator *op* to y and z, and store the result in x.

We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

Thus the source language like $x + y * z$ might be translated into a sequence

$t1 = y * z$

$t2 = x + t1$

where *t1* and *t2* are the compiler generated temporary name.

Three-Address Statements

- Assignment statements: $x = y$ *op* z, *op is binary*
- Assignment statements: $x =$ *op* y, *op is unary*
- Indexed assignments: $x = y[i]$, $x[i] = y$
- Pointer assignments: $x =$ **&**y, $x =$ *y, *$x = y$
- Copy statements: $x = y$
- Unconditional jumps: **goto** *label*
- Conditional jumps: **if** x *relop* y **goto** *label*
- Function calls: **param** x... **call** p, n **return** y

10. How next-use information is useful in code-generation? Explain the steps involved on computing next-use information.
Ans: Next-use information:

- The next-use information is a collection of all the names that are useful for a next-use subsequent statement in a block. Th75e use of a name is defined as follows,
- Consider a statement,

  x := i

  j := x op y
- That means the statement j uses a value of x.
- So, The next-use information can be collected by making the backward scan of the use programming code in that specific block.

Next-use information is needed for dead-code elimination and register assignment (if the name in a register is no longer needed, then the register can be assigned to some other name)

If *i*: *x* = ... and *j*: *y* = *x* + *z* are two statements *i* & *j*, then *next-use* of *x* at *i* is *j*.

Next-use is computed by a backward scan of a basic block and performing the following actions on statement

```
i: x := y op z
```

- Add liveness/next-use info on *x*, *y*, and *z* to statement *i* (whatever in the symbol table)
- Before going up to the previous statement (scan up):
  - Set *x* info to "not live" and "no next use"
  - Set *y* and *z* info to "live" and the next uses of *y* and *z* to *i*

All nontemporary variables and temporary that is used across the block are considered live.

Computing Next-Use

Example

Step 1

i: **a := b + c**

j: **t := a + b** [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Attach current live/next-use information
Because info is empty, assume variables are live
(Data flow analysis Ch.10 can provide accurate information)

Step 2

i: **a := b + c**

live(**a**) = true	*nextuse*(**a**) = *j*
live(**b**) = true	*nextuse*(**b**) = *j*
live(**t**) = false	*nextuse*(**t**) = none

j: **t := a + b** [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true, *nextuse*(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information at *j*

Computing Next-Use

Step 3 *i*: **a := b + c** [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: **t := a + b** [*live*(**a**) = true, *live*(**b**) = true, *live*(**t**) = true,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Attach current live/next-use information to *i*

live(**a**) = false	*nextuse*(**a**) = none
live(**b**) = true	*nextuse*(**b**) = *i*
live(**c**) = true	*nextuse*(**c**) = *i*
live(**t**) = false	*nextuse*(**t**) = none

Step 4

i: **a := b + c** [*live*(**a**) = true, *live*(**b**) = true, *live*(**c**) = false,
nextuse(**a**) = *j*, *nextuse*(**b**) = *j*, *nextuse*(**c**) = none]

j: **t := a + b** [*live*(**a**) = false, *live*(**b**) = false, *live*(**t**) = false,
nextuse(**a**) = none, *nextuse*(**b**) = none, *nextuse*(**t**) = none]

Compute live/next-use information *i*