Chapter 1

Lexical Analysis and Parsing

LEARNING OBJECTIVES

- *Language processing system*
- *Lexical analysis*
- *Syntax analysis*
- *Context free grammars and ambiguity*
- *Types of parsing*
- *Top down parsing*
- *Bottom up parsing*
- *Conflicts*
- *Operator precedence grammar*
- *LR parser*
- *Canonical LR parser(CLR)*

Language Processing System

Language Processors

Interpreter

It is a computer program that executes instructions written in a programming language. It either executes the source code directly or translates source code into some efficient intermediate representation and immediately executes this.

Example: Early versions of Lisp programming language, BASIC.

Translator

A software system that converts the source code from one form of the language to another form of language is called as translator. There are 2 types of translators namely (1) Compiler (2) Assembler.

Compiler converts source code of high level language into low level language.

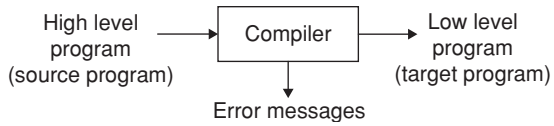
Assembler converts assembly language code into binary code.

Compilers

A compiler is a software that translates code written in high-level language (i.e., source language) into target language.

Example: source languages like C, Java,… etc. Compilers are user friendly.

The target language is like machine language, which is efficient for hardware.



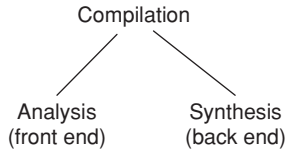
Passes

The number of iterations to scan the source code, till to get the executable code is called as a pass.

Compiler is two pass. Single pass requires more memory and multipass require less memory.

Analysis–synthesis model of compilation

There are two parts of compilation:

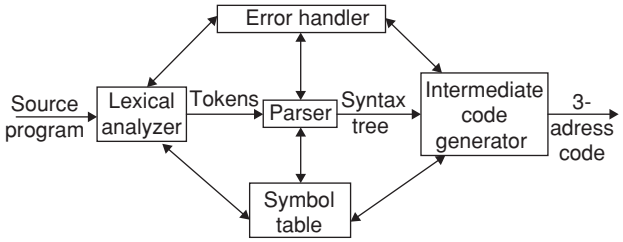


Analysis It breaks up the source program into pieces and creates an intermediate representation of the source program. This is more language specific.

Synthesis It constructs the desired target program from the intermediate representation. The target program will be more machine specific, dealing with registers and memory locations.

Front end vs back end of a compiler

The front end includes all analysis phases and intermediate code generator with part of code optimization.



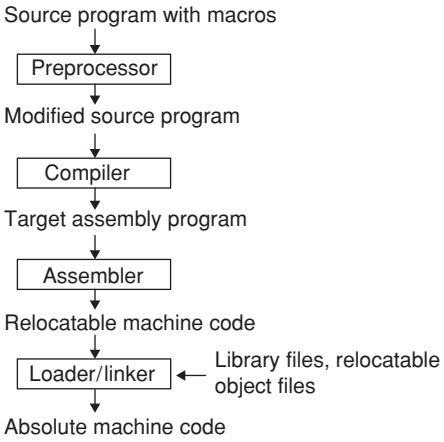
The back end includes code optimization and code generation phases. The back end synthesizes the target program from intermediate code.

Context of a compiler

In addition to a compiler, several other programs may be required to create an executable target program, like preprocessor to expand macros.

The target program created by a compiler may require further processing before it can be run.

The language processing system will be like this:



Phases

Compilation process is partitioned into some subproceses called phases.

In order to translate a high level code to a machine code, we need to go phase by phase, with each phase doing a particular task and parsing out its output for the next phase.

Lexical analysis or scanning

It is the first phase of a compiler. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

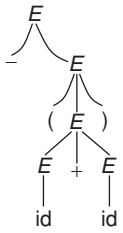
Example: Consider the statement: if ($a < b$)
In this sentence the tokens are if, (a, <, b,).
Number of tokens = 6
Identifiers: a, b
Keywords: if
Operators: <, (,)

Syntax analyzer or Parser

- Tokens are grouped hierarchically into nested collections with collective meaning.
- A context free grammar (CFG) specifies the rules or productions for identifying constructs that are valid in a programming language. The output is a parse/syntax/derivation tree.

Example: Parse tree for –(id + id) using the following grammar:

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow -E$ (G_1)
$E \rightarrow (E)$
$E \rightarrow id$



Semantic analysis

- It checks the source program for semantic errors.
- Type checking is done in this phase, where the compiler checks that each operator has matching operands for semantic consistency with the language definition.
- Gathers the type information for the next phases.

Example 1: The bicycle rides the boy.
This statement has no meaning, but it is syntactically correct.

Example 2:

```
int a;
bool b;
char c;
c = a + b;
```

We cannot add integer with a Boolean variable and assign it to a character variable.

Intermediate code generation

The intermediate representation should have two important properties:

(i) It should be easy to produce.
(ii) Easy to translate into the target program

'Three address code' is one of the common forms of Intermediate code.

Three address code consists of a sequence of instructions, each of which has at most three operands.

Example:

```
id1 = id2 + id3 × 10;
t1: = inttoreal(10)
```

t_2:= id_3 × t_1

t_3:= id_2 + t_2

id_1 = t_3

Code optimization

The output of this phase will result in faster running machine code.

Example: For the above intermediate code the optimized code will be

t_1:= id_3 × 10.0

id_1: = id_2 + t1

In this we eliminated t_2 and t_3 registers.

Code generation

- In this phase, the target code is generated.
- Generally the target code can be either a relocatable machine code or an assembly code.
- Intermediate instructions are each translated into a sequence of machine instructions.
- Assignment of registers will also be done.

Example:

```
MOVF    id3, R2
MULF    # 60.0, R2
MOVF    id2, R1
ADDF    R2, R1
MOVF    R1, id1
```

Symbol table management

A symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

What is the use of a symbol table?

1. To record the identifiers used in the source program.
2. Its type and scope
3. If it is a procedure name then the number of arguments, types of arguments, the method of parsing (by reference) and the type returned.

Error detection and reporting

(i) Lexical phase can detect errors where the characters remaining in the input 'do not form any token'.

(ii) Errors of the type, 'violation of syntax' of the language are detected by syntax analysis.

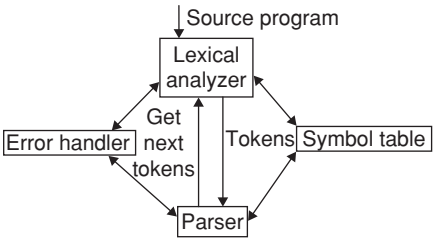
(iii) Semantic phase tries to detect constructs that have the right syntactic structure but no meaning.

Example: adding two array names etc.

Lexical Analysis

Lexical Analysis is the first phase in compiler design. The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis.

There will be interaction with the symbol table as well.



Lexeme: Sequence of characters in the source program that matches the pattern for a token. It is the smallest logical unit of a program.

Example: 10, *x*, *y*, <, >, =

Tokens: These are the classes of similar lexemes.

Example: Operators: <, >, =
Identifiers: *x*, *y*
Constants: 10
Keywords: if, else, int

Operations performed by lexical analyzer

1. Identification of lexemes and spelling check
2. Stripping out comments and white space (blank, new line, tab etc).
3. Correlating error messages generated by the compiler with the source program.
4. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by lexical analyzer.

Example 1: Take the following example from Fortran
DO 5 I = 1.25
Number of tokens = 5
The 1st lexeme is the keyword DO
Tokens are DO, 5, I, =, 1.25.

Example 2: An example from C program
for (int *i* = 1; *i* < = 10; i + +)
Here tokens are for, (, int, *i*, =, 1,;, *i*, < =, 10,;, i, ++,)
Number of tokens = 13

LEX compiler

Lexical analyzer divides the source code into tokens. To implement lexical analyzer we have two techniques namely hand code and the other one is LEX tool.

LEX is an automated tool which specifies lexical analyzer, from the rules given by the regular expression.

These rules are also called as pattern recognizing rules.

Syntax Analysis

This is the 2nd phase of the compiler, checks the syntax and constructs the syntax/parse tree.

Input of parser is token and output is a parse/ syntax tree.

Constructing parse tree

Construction of derivation tree for a given input string by using the production of grammar is called parse tree.

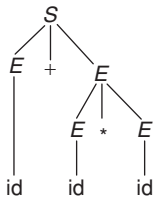
Consider the grammar

$S \rightarrow E + E/E * E$

$E \rightarrow id$

The parse tree for the string

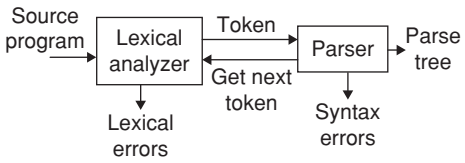
$\omega = id + id * id$ is



$\omega = id + id * id$

Role of the parser

1. Construct a parse tree.
2. Error reporting and correcting (or) recovery. A parser can be modeled by using CFG (Context Free Grammar) recognized by using pushdown automata/table driven parser.
3. CFG will only check the correctness of sentence with respect to syntax not the meaning.



How to construct a parse tree?

Parse tree's can be constructed in two ways.

(i) Top-down parser: It builds parse trees from the top (root) to the bottom (leaves).

(ii) Bottom-up parser: It starts from the leaves and works up to the root.

In both cases, the input to the parser is scanned from left to right, one symbol at a time.

Parser generator

Parser generator is a tool which creates a parser.

Example: compiler – compiler, YACC

The input of these parser generator is grammar we use and the output will be the parser code.

The parser generator is used for construction of the compilers front end.

Scope of declarations

Declaration scope refers to the certain program text portion, in which rules are defined by the language.

Within the defined scope, entity can access legally to declared entities.

The scope of declaration contains immediate scope always. Immediate scope is a region of declarative portion with enclosure of declaration immediately.

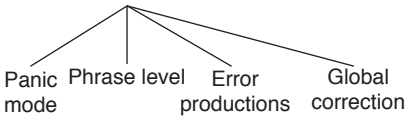
Scope starts at the beginning of declaration and scope continues till the end of declaration. Whereas in the over loadable declaration, the immediate scope will begin, when the callable entity profile was determined.

The visible part refers text portion of declaration, which is visible from outside.

Syntax Error Handling

1. Reports the presence of errors clearly and accurately.
2. Recovers from each error quickly.
3. It should not slow down the processing of correct programs.

Error Recovery Strategies



Panic mode On discovering an error, the parser discards input symbols one at a time until one of the synchronizing tokens is found.

Phrase level A parser may perform local correction on the remaining input. It may replace the prefix of the remaining input.

Error productions Parser can generate appropriate error messages to indicate the erroneous construct that has been recognized in the input.

Global corrections There are algorithms for choosing a minimal sequence of changes to obtain a globally least cost correction.

Context Free Grammars and Ambiguity

A grammar is a set of rules or productions which generates a collection of finite/infinite strings.

It is a 4-tuple defined as $G = (V, T, P, S)$

Where

V = set of variables

T = set of terminals

P = set of production rules

S = start symbol

Example: $S \rightarrow (S)/e$
$S \rightarrow (S)$ (1)
$S \rightarrow e$ (2)

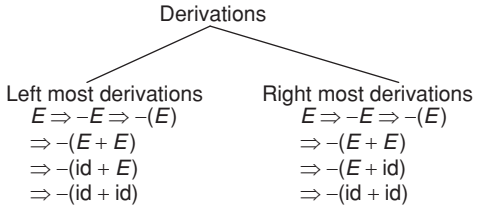
Here S is start symbol and the only variable.
(,), e is terminals.
(1) and (2) are production rules.

Sentential forms

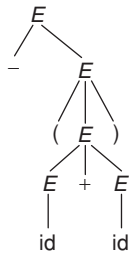
$s \overset{*}{\Rightarrow} \alpha$, Where α may contain non-terminals, then we say that α is a sentential form of G.

Sentence: A sentence is a sentential form with no non-terminals.

Example: $-(id + id)$ is a sentence of the grammar (G_1).



Right most derivations are also known as canonical derivations.



Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

Or

A grammar that produces more than one left most or more than one right most derivations is ambiguous.

For example consider the following grammar:

String → String + String/String – String /0/1/2/…/9

9 – 5 + 2 has two parse trees as shown below

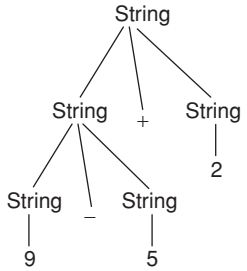


Figure 1 Leftmost derivation

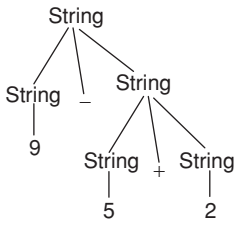


Figure 2 Rightmost derivation

- Ambiguity is problematic because the meaning of the program can be incorrect.
- Ambiguity can be handled in several ways
  1. Enforce associativity and precedence
  2. Rewrite the grammar by eliminating left recursion and left factoring.

Removal of ambiguity

The grammar is said to be ambiguous if there exists more than one derivation tree for the given input string.

The ambiguity of grammar is undecidable; ambiguity of a grammar can be eliminated by rewriting the grammar.

Example:
$E \rightarrow E + E/id\}$ → ambiguous grammar
$E \rightarrow E + T/T$ } rewritten grammar
$T \rightarrow id$ } (unambiguous grammar)

Left recursion

Left recursion can take the parser into infinite loop so we need to remove left recursion.

Elimination of left recursion
$A \rightarrow A\alpha/\beta$ is a left recursive.
It can be replaced by a non-recursive grammar:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A'/\varepsilon$$

In general

$$A \rightarrow A\alpha_1/A\alpha_2/\ldots/A\alpha_m/\beta_1/\beta_2/\ldots/\beta_n$$

We can replace A productions by

$$A \rightarrow \beta_1 A'/\beta_2 A'/-\beta_n A'$$
$$A' \rightarrow \alpha_1 A'/\alpha_2 A'/-\alpha_m A'$$

Example 3: Eliminate left recursion from
$E \rightarrow E + T/T$
$T \rightarrow T * F/F$
$F \rightarrow (E)/\text{id}$

Solution $E \rightarrow E + T/T$ it is in the form
$A \rightarrow A\alpha/\beta$
So, we can write it as $E \rightarrow TE'$
$E' \rightarrow +TE'/\varepsilon$
Similarly other productions are written as

$$T \rightarrow FT'$$
$$T^1 \rightarrow \times FT'/\in$$
$$F \rightarrow (E)/\text{id}$$

Example 4 Eliminate left recursion from the grammar

$$S \rightarrow (L)/a$$
$$L \rightarrow L, S/b$$

Solution: $S \rightarrow (L)/a$
$L \rightarrow bL'$
$L' \rightarrow SL'/\in$

Left factoring

A grammar with common prefixes is called non-deterministic grammar. To make it deterministic we need to remove common prefixes. This process is called as Left Factoring.

The grammar: $A \rightarrow \alpha\beta_1/\alpha\beta_2$ can be transformed into

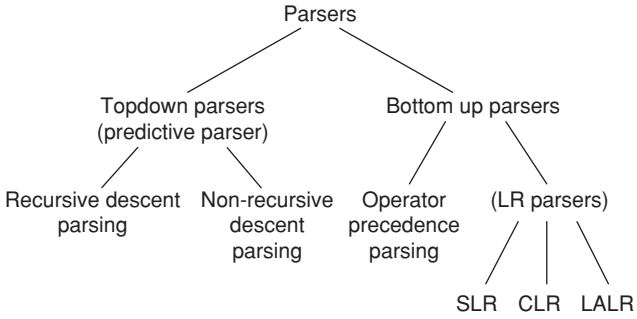
$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1/\beta_2$$

Example 5: What is the resultant grammar after left factoring the following grammar?

$S \rightarrow iEtS/iEtSeS/a$
$E \rightarrow b$

Solution: $S \rightarrow iEtSS'/a$
$S' \rightarrow eS/\in$
$E \rightarrow b$

Types of Parsing



Topdown Parsing

A parse tree is constructed for the input starting from the root and creating the nodes of the parse tree in preorder. It simulates the left most derivation.

Backtracking Parsing

If we make a sequence of erroneous expansions and subsequently discover a mismatch we undo the effects and roll back the input pointer.

This method is also known as brute force parsing.

Example: $S \rightarrow cAd$
$A \rightarrow ab/a$

Let the string w = cad is to generate:

The string generated from the above parse tree is cabd.
but, w = cad, the third symbol is not matched.
So, report error and go back to A.
Now consider the other alternative for production A.

String generated 'cad' and $w = cad$. Now, it is successful.

In this we have used back tracking. It is costly and time consuming approach. Thus an outdated one.

Predictive Parsers

By eliminating left recursion and by left factoring the grammar, we can have parse tree without backtracking. To construct a predictive parser, we must know,

1. Current input symbol
2. Non-terminal which is to be expanded

A procedure is associated with each non-terminal of the grammar.

Recursive descent parsing

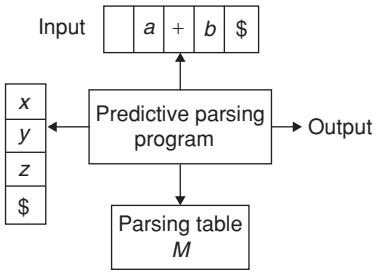
In recursive descent parsing, we execute a set of recursive procedures to process the input.

The sequence of procedures called implicitly, defines a parse tree for the input.

Non-recursive predictive parsing

(table driven parsing)

- It maintains a stack explicitly, rather than implicitly via recursive calls.
- A table driven predictive parser has
  - → An input buffer
  - → A stack
  - → A parsing table
  - → Output stream



Constructing a parsing table

To construct a parsing table, we have to learn about two functions:

1. FIRST ( )
2. FOLLOW ( )

FIRST(X) To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) is $\{X\}$.
2. If $X \to \varepsilon$ is a production, then add ε to FIRST(X).
3. If X is non-terminal and $X \to Y_1 Y_2 - Y_k$ is a production, then place 'a' in FIRST(X) if for some i, a is an FIRST (Y_i) and $\in$ is in all of FIRST(Y_1), …, FIRST(Y_{i-1}); that is, $Y_1, \ldots, Y_{i-1} \overset{*}{\Rightarrow} \in$. If $\in$ is in FIRST (Y_j) for all $j = 1, 2, \ldots, k$, then add $\in$ to FIRST(X). For example, everything in FIRST (Y_1) is surely in FIRST(X). If Y_1 does not derive $\in$, then add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \in$, then add FIRST ($Y_2$) and so on.

FOLLOW (A): To compute FOLLOW (A) for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is input right end marker.
2. If there is a production $A \to \alpha B\beta$, then everything in FIRST (β) except ε is placed in FOLLOW (B).
3. If there is a production $A \to \alpha B$ or a production $A \to \alpha B\beta$, where FIRST (β) contains ε, then everything in FOLLOW (A) is in FOLLOW (B).

Example: Consider the grammar

$E \to TE'$
$E' \to +TE'/\varepsilon$
$T \to FT'$
$T' \to *FT'/\varepsilon$
$F \to (E)/id$. Then
FIRST (E) = FIRST (T) = FIRST (F) = {(, id}
FIRST (E') = {$^+$, ε}
FIRST (T') = {*, ε}
FOLLOW (E) = FOLLOW (E') = {), \$}
FOLLOW (T) = FOLLOW (T') = {+,), \$}
FOLLOW (F) = {*, +,), \$}

Steps for the construction of predictive parsing table

1. For each production $A \to \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST (α), add $A \to \alpha$ to M [A, a]
3. If ε is in FIRST (α), add $A \to \alpha$ to M [A, b] for each terminal b in FOLLOW (A). If ε is in FIRST (α) and \$ is in FOLLOW (A), add $A \to \alpha$ to M [A, \$]
4. Make each undefined entry of M be error.

By applying these rules to the above grammar, we will get the following parsing table.

	Input Symbol					
Non-terminal	**id**	**+**	*****	**(**	**)**	**\$**
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ id			$F \to (E)$		

The parser is controlled by a program. The program consider x, the symbol on top of the stack and 'a' the current input symbol.

1. If $x = a =$ \$, the parser halts and announces successful completion of parsing.
2. If $x = a \neq$ \$, the parser pops x off the stack and advances the input pointer to the next input symbol.
3. If x is a non-terminal, the program consults entry $M[x, a]$ of the parsing table M. This entry will be either an x-production of the grammar or an error entry. If $M[x, a] = \{x \to UVW\}$, the parser replaces x on top of the stack by WVU with U on the top.

If $M[x, a]$ = error, the parser calls an error recovery routine.

For example, consider the moves made by predictive parser on input id + id ∗ id, which are shown below:

Matched	Stack	Input	Action
	E\$	id+id*id\$	
	TE'\$	id+id*id\$	Output E → TE'
	FT'E'\$	id+id*id\$	Output T → FT'
	idT'E'\$	id+id*id\$	Output F → id
id	T'E'\$	+id*id\$	Match id
id	E'\$	+id*id\$	Output T'→ ε
id	+TE'\$	+id*id\$	Output E'→ +TE'
id+	TE'\$	id*id\$	Match+
id+	FT'E'\$	id*id\$	Output T → FT'
id+	idT'E'\$	id*id\$	Output F → id
id+id	T'E'\$	*id\$	Match id
id+id	*FT'E'\$	*id\$	Output T' → *FT'
id+id*	FT'E'\$	id\$	Match*
id+id*	idT'E'\$	id\$	OutputF → id
id+id*id	T'E'\$	\$	Match id
id+id*id	E'\$	\$	Output T'→ ε
id+id*id	\$	\$	Output E'→ ε

Bottom up Parsing

- This parsing constructs the parse tree for an input string beginning at the leaves and working up towards the root.
- General style of bottom-up parsing is shift-reduce parsing.

Shift–Reduce Parsing

Reduce a string to the start symbol of the grammar. It simulates the reverse of right most derivation.

In every step a particular substring is matched (in left right fashion) to the right side of some production and then it is substituted by the non-terminal in the left hand side of the production.

For example consider the grammar

$S \rightarrow aABe$
$A \rightarrow Abc/b$
$B \rightarrow d$

In bottomup parsing the string 'abbcde' is verified as

$abbcde$
$aAbcde$
$aAde$ } → reverse order
$aABe$
S

Stack implementation of shift–reduce parser

The shift reduce parser consists of input buffer, Stack and parse table.

Input buffer consists of strings, with each cell containing only one input symbol.

Stack contains the grammar symbols, the grammar symbols are inserted using shift operation and they are reduced using reduce operation after obtaining handle from the collection of buffer symbols.

Parse table consists of 2 parts goto and action, which are constructed using terminal, non-terminals and compiler items.

Let us illustrate the above stack implementation.

→ Let the grammar be

$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

Let the input string 'ω' be abab$
ω = abab$

Stack	Input String	Action
$	*abab*$	Shift
$*a*	*bab*$	Shift
$*ab*	*ab*$	Reduce ($A \rightarrow b$)
$*aA*	*ab*$	Reduce ($A \rightarrow aA$)
$*A*	*ab*$	Shift
$*Aa*	*b*$	Shift
$*Aab*	$	Reduce ($A \rightarrow b$)
$*AaA*	$	Reduce ($A \rightarrow aA$)
$*AA*	$	Reduce ($S \rightarrow AA$)
$*S*	$	Accept

Rightmost derivation

$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$
For bottom up parsing, we are using right most derivation in reverse.

Handle of a string Substring that matches the RHS of some production and whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.

$$S \underset{*}{\overset{rm}{\Rightarrow}} \alpha Ar \Rightarrow \alpha\beta r$$

Right sentential forms of a unambiguous grammar have one unique handle.

Example: For grammar, $S \rightarrow aABe$
$A \rightarrow Abc/b$
$B \rightarrow d$
$S \Rightarrow \underline{aABe} \Rightarrow aA\underline{d}e \Rightarrow a\underline{Abc}de \Rightarrow a\underline{b}bcde$

Note: Handles are underlined.

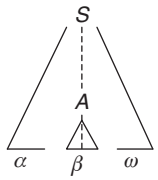
Handle pruning The process of discovering a handle and reducing it to the appropriate left hand side is called handle pruning. Handle pruning forms the basis for a bottomup parsing.

To construct the rightmost derivation:

$$S = r_0 \Rightarrow r_1 \Rightarrow r_2 \_\_\_\_ \Rightarrow r_n = w$$

Apply the following simple algorithm:
For $i \leftarrow n$ to 1
Find the handle $A_i \rightarrow B_i$ in r_i
Replace B_i with A_i to generate r_{i-1}

Consider the cut of a parse tree of a certain right sentential form:



Here $A \rightarrow \beta$ is a handle for $\alpha\beta\omega$.

Shift reduce parsing with a stack There are 2 problems with this technique:

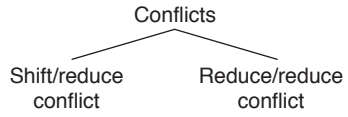
(i) To locate the handle
(ii) Decide which production to use

General construction using a stack

1. 'Shift' input symbols onto the stack until a handle is found on top of it.
2. 'Reduce' the handle to the corresponding non-terminal.
3. 'Accept' when the input is consumed and only the start symbol is on the stack.
4. Errors – call an error reporting/recovery routine.

Viable prefixes The set of prefixes of a right sentential form that can appear on the stack of a shift reduce parser are called viable prefixes.

Conflicts



Shift/reduce conflict

Example: stmt → if expr then stmt | if expr then stmt else stmt | any other statement

If exp then stmt is on the stack, in this case we can't tell whether it is a handle. i.e., 'shift/reduce' conflict.

Reduce/reduce conflict

Example: $S \to aA/bB$
$A \to c$
$B \to c$
W = ac it gives reduce/reduce conflict.

Operator Precedence Grammar

In operator grammar, no production rule can have:
- ε at the right side.
- two adjacent non-terminals at the right side.

Example 1: $E \to E + E / E - E /$ id is operator grammar.

Example 2: $E \to AB$, $A \to a$, $B \to b$ } not operator grammar

Example 3: $\underbrace{E \to E0E/\text{id}}$
not operator grammar

Precedence relation If
$a < b$ then b has higher precedence than a
$a = b$ then b has same precedence as a
$a > b$ then b has lower precedence than a

Common ways for determining the precedence relation between pair of terminals:

1. Traditional notations of associativity and precedence.

Example: × has higher precedence than + × .> + (or) + <. ×

2. First construct an unambiguous grammar for the language which reflects correct associativity and precedence in its parse tree.

Operator precedence relations from associativity and precedence

Let us use \$ to mark end of each string. Define \$ <. b and b ⋗ \$ for all terminals b. Consider the grammar is:

$$E \to E + E/E \times E/\text{id}$$

Let the operator precedence table for this grammar is:

	id	+	×	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
×	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	accept

1. Scan the string from left until ⋗ is encountered
2. Then scan backwards (to left) over any = until ⋖ is encountered.
3. The handle contains everything to the left of the first ⋗ and to the right of the ⋖ is encountered.

After inserting precedence relation is

\$id + id * id \$ is
\$ ⋖ id ⋗ + ⋖ id ⋗ * ⋖ id ⋗ \$

Precedence functions Instead of storing the entire table of precedence relations table, we can encode it by precedence functions f and g, which map terminal symbols to integers:

1. $f(a) \lessdot f(b)$ whenever $a \lessdot b$
2. $f(a) \gtrdot f(b)$ whenever $a \doteq b$
3. $f(a) > f(b)$ whenever $a \gtrdot b$

Finding precedence functions for a table

1. Create symbols $f(a)$ and $g(a)$ for each 'a' that is a terminal or \$.
2. Partition the created symbols into as many groups as possible in such away that $a = b$ then f (a) and g (b) are in the same group
3. Create a directed graph
   If $a < b$ then place an edge from $g(b)$ to $f(a)$
   If $a > b$ then place an edge from $f(a)$ to $g(b)$
4. If the graph constructed has a cycle then no precedence function exists.
   If there are no cycles, let $f(a)$ be the length of the longest path being at the group of $f(a)$.
   Let $g(a)$ be the length of the longest path from the group of $g(a)$.

Disadvantages of operator precedence parsing

- It can not handle unary minus.
- Difficult to decide which language is recognized by grammar.

Advantages

1. Simple
2. Powerful enough for expressions in programming language.

Error cases

1. No relation holds between the terminal on the top of stack and the next input symbol.
2. A handle is found, but there is no production with this handle as the right side.

Error recovery

1. Each empty entry is filled with a pointer to an error routine.
2. Based on the handle tries to recover from the situation.

To recover, we must modify (insert/change)

1. Stack or
2. Input or
3. Both

We must be careful that we don't get into an infinite loop.

LR Parsers

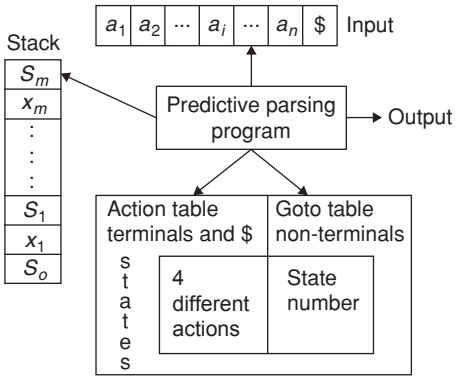
- In LR (K), L stands for Left to Right Scanning, R stands for Right most derivation, K stands for number of look ahead symbols.
- LR parsers are table-driven, much like the non-recursive LL parsers. A grammar which is used in construction of LR parser is LR grammar. For a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on the top of the stack.
- The Time complexity for such parsers is $O(n^3)$
- LR parsers are faster than LL (1) parser.
- LR parsing is attractive because
  - The most general non-backtracking shift reduce parser.
  - The class of grammars that can be passed using LR methods is a proper superset of predictive parsers. LL (1) grammars $\subset$ LR (1) grammars.
  - LR parser can detect a syntactic error in the left to right scan of the input.
- LR parsers can be implemented in 3 ways:
  1. Simple LR (SLR): The easiest to implement but the least powerful of the three.
  2. Canonical LR (CLR): most powerful and most expensive.
  3. Look ahead LR (LALR): Intermediate between the remaining two. It works on most programming language grammars.

Disadvantages of LR parser

1. Detecting a handle is an overhead, parse generator is used.
2. The main problem is finding the handle on the stack and it was replaced with the non-terminal with the left hand side of the production.

The LR parsing algorithm

- It consists of an input, an output, a stack, a driver program and a parsing table that has two parts (action and goto).
- The driver/parser program is same for all these LR parsers, only the parsing table changes from parser to another.



Stack: To store the string of the form,

$S_o x_1 S_1 \ldots x_m S_m$ where

S_m: state

x_m: grammar symbol

Each state symbol summarizes the information contained in the stack below it.

Parsing table: Parsing table consists of two parts:

1. Action part
2. Goto part

ACTION Part:

Let, $S_m \rightarrow$ top of the stack

$a_i \rightarrow$ current symbol

Then action $[S_m, a_i]$ which can have one of four values:

1. Shift S, where S is a state
2. Reduce by a grammar production $A \rightarrow \beta$
3. Accept
4. Error

GOTO Part:

If goto $(S, A) = X$ where $S \rightarrow$ state, $A \rightarrow$ non-terminal, then GOTO maps state S and non-terminal A to state X.

Configuration

$$(S_o x_1 S_1 x_2 S_2 - x_m S_m, a_i a_{i+1} - a_n \$)$$

The next move of the parser is based on action $[S_m, a_i]$

The configurations are as follows.

1. If action $[S_m, a_i]$ = shift S

$$(S_o x_1 S_1 x_2 S_2 \text{---} x_m S_m, a_i a_{i+1} \text{ --- } a_n \$)$$

2. If action $[S_m, a_i]$ = reduce $A \rightarrow \beta$ then

$$(S_o x_1 S_1 x_2 S_2 \text{---} x_{m-r} S_{m-r}, AS, a_i a_{i+1} \text{ --- } a_n \$)$$

Where S = goto $[S_{m-r}, A]$

3. If action $[S_m, a_i]$ = accept, parsing is complete.
4. If action $[S_m, a_i]$ = error, it calls an error recovery routine.

Example: Parsing table for the following grammar is shown below:

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

	Action						Goto		
State	id	+	×	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_1				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

Moves of LR parser on input string id*id+id is shown below:

Stack	Input	Action
0	id * id + id$	Shift 5
0id 5	* id + id$	reduce 6 means reduce with 6th production $F \rightarrow$ id and goto [0, F] = 3
0F 3	* id + id$	reduce 4 i.e $T \rightarrow F$ goto [0, T] = 2
0T 2	* id + id$	Shift 7
0T2 * 7	id + id$	Shift 5
0T2 * 7 id 5	+ id$	reduce 6 i.e $F \rightarrow$ id goto [7, F] = 10
0T2 * 7 F 10	+ id$	reduce 3 i.e $T \rightarrow T*F$
0T 2	+ id$	goto [0, T] = 2
0E 1	+ id$	reduce 2 i.e $E \rightarrow T$ & goto [0, E] = 1
0E1 + 6	id$	Shift 6
0E1 + 6 id 5	$	Shift 5
0E1 + 6F 3	$	reduce 6 & goto [6, F] = 3
0E1 + 6T 9	$	reduce 4 & goto [6, T] = 9
0E1	$	reduce 1 & goto [0, E] = 1
0E1	$	accept

Constructing SLR parsing table

LR (0) item: LR (0) item of a grammar G is a production of G with a dot at some position of the right side of production.

Example: $A \rightarrow BCD$
Possible LR (0) items are

$A \rightarrow .BCD$
$A \rightarrow B.CD$
$A \rightarrow BC.D$
$A \rightarrow BCD.$

$A \rightarrow B.CD$ means we have seen an input string derivable from B and hope to see a string derivable from CD.

The LR (0) items are constructed as a DFA from grammar to recognize viable prefixes.

The items can be viewed as the states of NFA.

The LR (0) item (or) canonical LR (0) collection, provides the basis for constructing SLR parser.

To construct LR (0) items, define
(a) An augmented grammar
(b) closure and goto

Augmented grammar (G′) If G is a grammar with start symbol S, G' the augmented grammar for G, with new start symbol S' and production $S' \rightarrow S$.

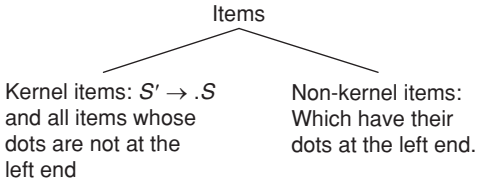
Purpose of G' is to indicate when to stop parsing and announce acceptance of the input.

Closure operation Closure (I) includes

1. Intially, every item in I is added to closure (I)
2. If A $\rightarrow \alpha.B\beta$ is in closure (I) and $\beta \rightarrow \gamma$ is a production then add $B \rightarrow .\gamma$ to I.

Goto operation

Goto (I, x) is defined to be the closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$ is in I.



Construction of sets of Items

Procedure items (G')
Begin
C: = closure ({[$S' \rightarrow .S$]});
repeat
For each set of items I in C and each grammar symbol x
Such that goto (I, x) is not empty and not in C do add goto (I, x) to C;
Until no more sets of items can be added to C, end;

Example: LR (0) items for the grammar

$E' \rightarrow E$
$E \rightarrow E + T/T$
$T \rightarrow T * F/F$
$F \rightarrow (E)/$id

is given below:
I_0: $E' \rightarrow .E$
$E \rightarrow .E + T$
$E \rightarrow .T$
$T \rightarrow .T * F$

$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .\text{id}$

I_1: got (I_0, E)
$E' \rightarrow E.$
$E \rightarrow E. + T$

I_2: goto (I_0, T)
$E \rightarrow T.$
$T \rightarrow T. * F$

I_3: goto (I_0, F)
$T \rightarrow F.$

I_4: goto $(I_0, ( )$
$F \rightarrow (.E)$
$E \rightarrow .E + T$
$E \rightarrow .T$
$E \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .\text{id}$

I_5: goto (I_0, id)
$F \rightarrow \text{id}.$

I_6: got $(I_1, +)$
$E \rightarrow E+ .T$
$T \rightarrow .T * F$
$T \rightarrow .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

I_7: goto $(I_2, *)$
$T \rightarrow T* .F$
$F \rightarrow .(E)$
$F \rightarrow .id$

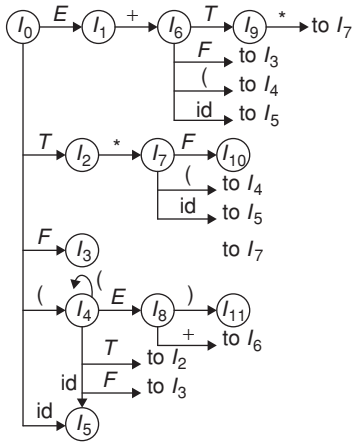
I_8: goto (I_4, E)
$F \rightarrow (E.)$

I_9: goto (I_6, T)
$E \rightarrow E+ T.$
$T \rightarrow T.* F$

I_{10}: goto (I_7, F)
$T \rightarrow T* F.$

I_{11}: goto $(I_8,))$
$F \rightarrow (E).$

For viable prefixes construct the DFA as follows:



SLR parsing table construction

1. Construct the canonical collection of sets of LR (0) items for G'.
2. Create the parsing action table as follows:
   (a) If a is a terminal and $[A \rightarrow \alpha.a\beta]$ is in I_i, goto $(I_i, a) = I_j$ then action (i, a) to shift j. Here 'a' must be a terminal.
   (b) If $[A \rightarrow \alpha.]$ is in I_i, then set action $[i, a]$ to 'reduce $A \rightarrow \alpha$' for all a in FOLLOW (A);
   (c) If $[S' \rightarrow S.]$ is in I_i then set action $[i, \$]$ to 'accept'.
3. Create the parsing goto table for all non-terminals A, if goto $(I_i, A) = I_j$ then goto $[i, A] = j$.
4. All entries not defined by steps 2 and 3 are made errors.
5. Initial state of the parser contains $S' \rightarrow S$.
   The parsing table constructed using the above algorithm is known as SLR (1) table for G.

Note: Every SLR (1) grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

Example 6: Construct SLR parsing table for the following grammar:

1. $S \rightarrow L = R$
2. $S \rightarrow R$
3. $L \rightarrow * R$
4. $L \rightarrow \text{id}$
5. $R \rightarrow L$

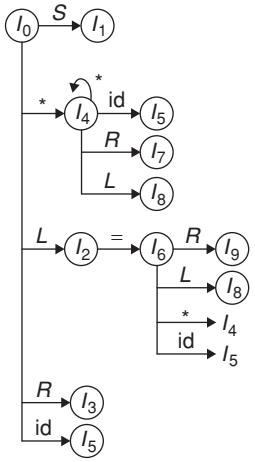
Solution: For the construction of SLR parsing table, add $S' \rightarrow S$ production.

$S' \rightarrow S$
$S \rightarrow L = R$
$S \rightarrow R$
$L \rightarrow *R$
$L \rightarrow \text{id}$
$R \rightarrow L$

LR (0) items will be

I_0: $S' \rightarrow .S$
$S \rightarrow .L = R$
$S \rightarrow .R$
$L \rightarrow .*R$
$L \rightarrow .\text{id}$
$R \rightarrow .L$
I_1: goto (I_0, S)
$S' \rightarrow S.$
I_2: goto (I_0, L)
$S \rightarrow L. = R$
$R \rightarrow L.$
I_3: got (I_0, R)
$S \rightarrow R.$
I_4: goto (I_0, *)
$L \rightarrow *.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .\text{id}$
I_5: goto(I_0, id)
$L \rightarrow \text{id}.$
I_6: goto(I_2, =)
$S \rightarrow L = .R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .\text{id}$
I_7: goto(I_4, R)
$L \rightarrow *R.$
I_8: goto(I_4, L)
$R \rightarrow L.$
I_9: goto(I_6, R)
$S \rightarrow L = R.$

The DFA of LR(0) items will be



States	Action =	Action *	Action id	Action $	Goto S	Goto L	Goto R
0		S_4	S_5		1	2	3
1				acc			
2	S_6, r_5			r_5			
3							
4		S_4	S_5			8	7
5							
6		S_4	S_5			8	9
7							
8							
9							

FOLLOW (S) = {\$}
FOLLOW (L) = {=}
FOLLOW (R) = {\$, =}
For action [2, =] = S_6 and r_5
∴ Here we are getting shift – reduce conflict, so it is not SLR (1).

Canonical LR Parsing (CLR)

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information input into a state by including a terminal symbol as a second component of an item.
- The general form of an item

  $[A \rightarrow \alpha.\beta, a]$
  Where $A \rightarrow \alpha\beta$ is a production.
  a is terminal/right end marker (\$). We will call it as LR (1) item.

LR (1) item

It is a combination of LR (0) items along with look ahead of the item. Here 1 refers to look ahead of the item.

Construction of the sets of LR (1) items Function closure (I):
Begin
Repeat
For each item $[A \rightarrow \alpha.B\beta, a]$ in I,
Each production $B \rightarrow .\gamma$ in G',
And each terminal b in FIRST (βa)
Such that $[B \rightarrow .\gamma, b]$ is not in I do
Add $[B \rightarrow .\gamma, b]$ to I;
End;
Until no more items can be added to I;

Example 7: Construct CLR parsing table for the following grammar:

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC/d$

Solution: The initial set of items is

I_0: $S' \to .S, \$$
$S \to .CC, \$$
$A \to \alpha.B\beta, a$

Here $A = S$, $\alpha = \in$, $B = C$, $\beta = C$ and $a = \$$
First (βa) is first ($C\$$) = first (C) = {c, d}
So, add items [$C \to .cC, c$]
[$C \to .cC, d$]

$\therefore$ Our first set I_0: $S' \to .S, \$$
$S \to .CC, \$$
$C \to .$coca, c/d
$C \to .d, c/d.$

I_1: goto (I_0, X) if $X = S$
$S' \to S., \$$

I_2 : goto (I_0, C)
$S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

I_3: goto (I_0, c)
$C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d\ c/d$

I_4: goto (I_0, d)
$C \to d., c/d$

I_5: goto (I_2, C)
$S \to CC., \$$

I_6: goto (I_2, c)
$C \to c.C; \$$
$C \to ..cC, \$$
$C \to ..d, \$$

I_7: goto (I_2, d)
$C \to d. \$$

I_8: goto (I_3, C)
$C \to cC., c/d$

I_9: goto (I_6, C)
$C \to cC., \$$

CLR table is:

States	Action			Goto	
	c	*1*	$	*S*	*C*
I_0	S_3	S_4		1	2
I_1			acc		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	R_2	r_2			
I_9			r_2		

Consider the string derivation '*dcd*':

$$S \Rightarrow CC \Rightarrow CcC \Rightarrow Ccd \Rightarrow dcd$$

Stack	Input	Action
0	*dcd*$	shift 4
0*d*4	*Cd*$	reduce 3 i.e. $C \to d$
0*C*2	*Cd*$	shift 6
0*C*2*C*6	*D*$	shift 7
0*C*2*C*6*d*7	$	reduce $C \to d$
0*C*2*C*6*C*9	$	reduce $C \to cC$
0*C*2*C*5	$	reduce $S \to CC$
0*S*1	$	

Example 8: Construct CLR parsing table for the grammar:
$S \to L = R$
$S \to R$
$L \to *R$
$L \to id$
$R \to L$

Solution: The canonical set of items is

I_0: $S' \to .S, \$$
$S \to .L = R, \$$
$S \to .R, \$$
$L \to .* R, =$ [first (= R$) = {=}]
$L \to .$id, =
$R \to .L, \$$

I_1: got (I_0, S)
$S' \to S., \$$

I_2: goto (I_0, L)
$S \to L. = R, \$$
$R \to L., \$$

I_3: goto (I_0, R)
$S \to R., \$$

I_4: got (I_0, *)
$L \to *. R, =$
$R \to .L, =$
$L \to .* R, =$
$L \to .$id, =
I_5: goto (I_0, id)
$L \to$ id.,=
I_6: goto (I_7, L)
$R \to L., \$$

I_7: goto (I_2, =)
$S \to L = .R,$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .$id, $

I_8: goto (I_4, R)
$L \to *R., =$

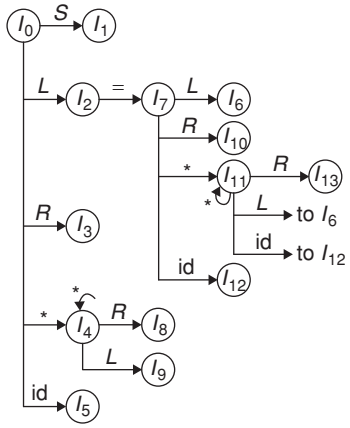
I_9: goto (I_4, L)
$R \rightarrow L.,=$

I_{10}: got (I_7, R)
$S \rightarrow L = R., \$$

I_{11}: goto (I_7, *)
$L \rightarrow *.R, \$$
$R \rightarrow .L, \$$
$L \rightarrow .*R, \$$
$L \rightarrow .\text{id}, \$$

I_{12}: goto (I_7, id)
$L \rightarrow \text{id}. , \$$

I_{13}: goto (I_{11}, R)
$L \rightarrow *R., \$$



We have to construct CLR parsing table based on the above diagram.

In this, we are going to have 13 states

The shift –reduce conflict in the SLR parser is reduced here.

States	id	*	=	$	S	L	R
0	S_5	S_4			1	2	3
1				acc			
2			S_7	r_5			
3				r_2			
4	S_5	S_4				9	8
5			r_4				
6				r_5			
7	s_{12}	s_{11}				6	10
8			r_3				
9			r_5				
10				r_1			
11	S_{12}	S_{11}					13
12				r_4			
13				r_3			

Stack	Input
0	*Id* = *id*$
0*id*5	= *id*$
0*L*2	= *id*$
0*L*2 = 7	*id*$
0*L*2 = 7! *d*12	$
0*L*2 = 7*L*6	$
0*L*2= 7*R*10	$
0*S*1 (accept)	$

Every SLR (1) grammar is LR (1) grammar.
CLR (1) will have 'more number of states' than SLR Parser.

LALR Parsing Table

- The tables obtained by it are considerably smaller than the canonical LR table.
- LALR stands for Lookahead LR.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR.
- YACC creates a LALR parser for the given grammar.
- YACC stands for 'Yet another Compiler'.
- An easy, but space-consuming LALR table construction is explained below:
  1. Construct $C = \{I_0, I_1, -I_n\}$, the collection of sets of LR (1) items.
  2. Find all sets having the common core; replace these sets by their union
  3. Let $C' = \{J_O, J_1 \text{ --- } J_m\}$ be the resulting sets of LR (1) items. If there is a parsing action conflict then the grammar is not a LALR (1).
  4. Let k be the union of all sets of items having the same core. Then goto $(J, X) = k$
- If there are no parsing action conflicts then the grammar is said to LALR (1) grammar.
- The collection of items constructed is called LALR (1) collection.

Example 9: Construct LALR parsing table for the following grammar:

$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC/d$

Solution: We already got LR (1) items and CLR parsing table for this grammar.
After merging I3 and I6 are replaced by I36.

I_{36}: $C \rightarrow c.C, c/d/\$$
$C \rightarrow .cC, c/d/\$$
$C \rightarrow .d, c/d/\$$

I_{47}: By merging I_4 and I_7
$C \rightarrow d.\ c/d/\$$

I_{89}: I_8 and I_9 are replaced by I_{89}
$C \rightarrow cC.,\ c/d/\$$

The LALR parsing table for this grammar is given below:

	Action			goto	
State	***c***	***d***	**$**	***S***	***C***
0	S_{36}	S_{47}		1	2
1			*acc*		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	R_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Example: Consider the grammar:

$S' \rightarrow S$
$S \rightarrow aAd$
$S \rightarrow bBd$
$S \rightarrow aBe$
$S \rightarrow bAe$
$A \rightarrow c$
$B \rightarrow c$

Which generates strings acd, bcd, ace and bce

LR (1) items are

I_0: $S' \rightarrow .S,\ \$$
$S \rightarrow .aAd,\ \$$
$S \rightarrow .bBd,\ \$$
$S \rightarrow .aBe,\ \$$
$S \rightarrow .bAe,\ \$$

I_1: goto (I_0, S)
$S' \rightarrow S.,\ \$$

I_2: goto (I_0, a)
$S \rightarrow a.Ad,\ c$
$S \rightarrow a.Be,\ c$
$A \rightarrow .c,d$
$B \rightarrow .c,e$

I_3: goto (I_0, b)
$S \rightarrow b.Bd,\ c$
$S \rightarrow b.Ae,\ c$
$A \rightarrow .c,\ e$
$B \rightarrow .c,\ e$

I_4: goto (I_2, A)
$S \rightarrow aA.d,\ c$

I_5: goto (I_2, B)
$S \rightarrow aB.e,\ c$

I_6: goto (I_2, c)
$A \rightarrow c.,\ d$
$B \rightarrow c.,\ e$

I_7: goto (I_3, c)
$A \rightarrow c.,\ e$
$B \rightarrow c.,\ d$

I_8: goto (I_4, d)
$S \rightarrow aAd.,\ c$

I_9: goto (I_5, e)
$S \rightarrow aBe.,\ c$

If we union I_6 and I_7
$A \rightarrow c.,\ d/e$
$B \rightarrow c.,\ d/e$

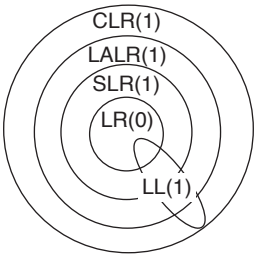
It generates reduce/reduce conflict.

Notes:

1. The merging of states with common cores can never produce a shift/reduce conflict, because shift action depends only on the core, not on the lookahead.
2. SLR and LALR tables for a grammar always have the same number of states (several hundreds) whereas CLR have thousands of states for the same grammar.

Comparison of parsing methods

Method	Item	Goto and Closures	Grammar it Applies to
SLR (1)	LR(0) item	Different from LR(1)	SLR (1) ⊂ LR(1)
LR (1)	LR(1) item		LR(1) – Largest class of LR grammars
LALR(1)	LR(1) item	Same as LR(1)	LALR(1) ⊂ LR(1)



Every LR (0) is SLR (1) but vice versa is not true.

Difference between SLR, LALR and CLR parsers

Differences among SLR, LALR and CLR are discussed below in terms of size, efficiency, time and space.

Table 1 *Comparison of parsing methods*

Sl. No.	Factors	SLR Parser	LALR Parser	CLR Parser
1	Size	Smaller	Smaller	Larger
2.	Method	It is based on FOLLOW function	This method is applicable to wider class than SLR	This is most powerful than SLR and LALR.
3.	Syntactic features	Less exposure compared to other LR parsers	Most of them are expressed	Less
4.	Error detection	Not immediate	Not immediate	Immediate
5.	Time and space complexity	Less time and space	More time and space complexity	More time and space complexity

EXERCISES

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Consider the grammar
$S \rightarrow a$
$S \rightarrow ab$
The given grammar is:
(A) LR (1) only
(B) LL (1) only
(C) Both LR (1) and LL (1)
(D) LR (1) but not LL (1)

2. Which of the following is an unambiguous grammar, that is not LR (1)?
(A) $S \rightarrow$ Uab | Vac
$U \rightarrow d$
$V \rightarrow d$
(B) $S \rightarrow$ Uab/Vab/Vac
$U \rightarrow d$
$V \rightarrow d$
(C) $S \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$
(D) $S \rightarrow Ab$
$A \rightarrow a/c$

Common data for questions 3 and 4: Consider the grammar:

$S \rightarrow T; S/\in$
$T \rightarrow UR$
$U \rightarrow x/y/[S]$
$R \rightarrow .T/\in$

3. Which of the following are correct FIRST and FOLLOW sets for the above grammar?
(i) FIRST(S) = FIRST (T) = FIRST (U) = $\{x, y, [, \varepsilon\}$
(ii) FIRST (R) = $\{,\varepsilon\}$
(iii) FOLLOW (S) = {], $}
(iv) FOLLOW (T) = Follow (R) = {;}
(v) FOLLOW (U) = {. ;}
(A) (i) and (ii) only
(B) (ii), (iii), (iv) and (v) only
(C) (ii), (iii) and (iv) only
(D) All the five

4. If an LL (1) parsing table is constructed for the above grammar, the parsing table entry for [$S \rightarrow$ [ ] is
(A) $S \rightarrow T; S$ (B) $S \rightarrow \in$
(C) $T \rightarrow UR$ (D) $U \rightarrow [S]$

Common data for questions 5 to 7: Consider the augmented grammar

$S \rightarrow X$
$X \rightarrow (X)/a$

5. If a DFA is constructed for the LR (1) items of the above grammar, then the number states present in it are:
(A) 8 (B) 9
(C) 7 (D) 10

6. Given grammar is
(A) Only LR (1)
(B) Only LL (1)
(C) Both LR (1) and LL (1)
(D) Neither LR (1) nor LL (1)

7. What is the number of shift-reduce steps for input (a)?
(A) 15 (B) 14
(C) 13 (D) 16

8. Consider the following two sets of LR (1) items of a grammar:

$X \rightarrow c.X, c/d$	$X \rightarrow c.X, \$$
$X \rightarrow .cX, c/d$	$X \rightarrow .cX, \$$
$X \rightarrow d, c/d$	$X \rightarrow .d, \$$

Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?
1. Cannot be merged since look ahead are different.
2. Can be merged but will result in $S - R$ conflict.
3. Can be merged but will result in $R - R$ conflict.
4. Cannot be merged since goto on c will lead to two different sets.

(A) 1 only (B) 2 only
(C) 1 and 4 only (D) 1, 2, 3 and 4

9. Which of the following grammar rules violate the requirements of an operator grammar?
(i) $A \rightarrow BcC$ (ii) $A \rightarrow dBC$
(iii) $A \rightarrow C/\in$ (iv) $A \rightarrow cBdC$

(A) (i) only (B) (i) and
(C) (ii) and (iii) only (D) (i) and (iv) only

10. The FIRST and FOLLOW sets for the grammar:
$S \rightarrow SS + /SS^*/a$
(A) First (S) = {a}
Follow (S) = {+, *, $}
(B) First (S) = {+}
Follow (S) = {+, *, $}
(C) First (S) = {a}
Follow (S) = {+, *}
(D) First (S) = {+, *}
Follow (S) = {+, *, $}

11. A shift reduces parser carries out the actions specified within braces immediately after reducing with the corresponding rule of the grammar:
$S \rightarrow xxW$ [print '1']
$S \rightarrow y$ [print '2']
$W \rightarrow Sz$ [print '3']
What is the translation of '$x\ x\ x\ y\ z\ z$'?
(A) 1231 (B) 1233
(C) 2131 (D) 2321

12. After constructing the predictive parsing table for the following grammar:
$Z \rightarrow d$
$Z \rightarrow XYZ$
$Y \rightarrow c/\in$
$X \rightarrow Y$
$X \rightarrow a$
The entry/entries for [Z, d] is/are
(A) $Z \rightarrow d$
(B) $Z \rightarrow XYZ$
(C) Both (A) and (B)
(D) $X \rightarrow Y$

13. The following grammar is
$S \rightarrow AaAb/BbBa$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$
(A) LL (1) (B) Not LL (1)
(C) Recursive (D) Ambiguous

14. Compute the FIRST (P) for the below grammar:
$P \rightarrow AQR$be/mn/DE
$A \rightarrow ab/\varepsilon$
$Q \rightarrow q_1q_2/\varepsilon$
$R \rightarrow r_1r_2/\varepsilon$
$D \rightarrow d$
$E \rightarrow e$
(A) {m, a} (B) {m, a, q_1, r_1, b, d}
(C) {d, e} (D) {m, n, a, b, d, e, q_1, r_1}

15. After constructing the LR(1) parsing table for the augmented grammar
$S' \rightarrow S$
$S \rightarrow BB$
$B \rightarrow aB/c$
What will be the action [I_3, a]?
(A) Accept (B) S_7
(C) r_2 (D) S_5

Practice Problems 2

Directions for questions 1 to 19: Select the correct alternative from the given choices.

1. Consider the grammar
$S \rightarrow aSb$
$S \rightarrow aS$
$S \rightarrow \varepsilon$
This grammar is ambiguous by generating which of the following string.
(A) aa (B) $\in$
(C) aaa (D) aab

2. To convert the grammar $E \rightarrow E + T$ into LL grammar
(A) use left factor
(B) CNF form
(C) eliminate left recursion
(D) Both (B) and (C)

3. Given the following expressions of a grammar
$E \rightarrow E \times F/F + E/F$
$F \rightarrow F$? F/id
Which of the following is true?
(A) × has higher precedence than +
(B) ? has higher precedence than ×
(C) + and? have same precedence
(D) + has higher precedence than *

4. The action of parsing the source program into the proper syntactic classes is known as
(A) Lexical analysis
(B) Syntax analysis
(C) Interpretation analysis
(D) Parsing

5. Which of the following is not a bottom up parser?
(A) LALR (B) Predictive parser
(C) CLR (D) SLR

6. A system program that combines separately compiled modules of a program into a form suitable for execution is
(A) Assembler.
(B) Linking loader.
(C) Cross compiler.
(D) None of these.

7. Resolution of externally defined symbols is performed by a
(A) Linker (B) Loader.
(C) Compiler. (D) Interpreter.

8. LR parsers are attractive because
(A) They can be constructed to recognize CFG corresponding to almost all programming constructs.
(B) There is no need of backtracking.
(C) Both (A) and (B).
(D) None of these

9. YACC builds up
(A) SLR parsing table
(B) Canonical LR parsing table
(C) LALR parsing table
(D) None of these

10. Language which have many types, but the type of every name and expression must be calculated at compile time are
(A) Strongly typed languages
(B) Weakly typed languages
(C) Loosely typed languages
(D) None of these

11. Consider the grammar shown below:
$S \rightarrow iEtSS'/a/b$
$S' \rightarrow eS/\varepsilon$

In the predictive parse table M, of this grammar, the entries $M[S', e]$ and $M[S', \$]$ respectively are
(A) $\{S' \rightarrow eS\}$ and $\{S' \rightarrow \in\}$
(B) $\{S' \rightarrow eS\}$ and $\{\ \}$
(C) $\{S' \rightarrow \in\}$ and $\{S' \rightarrow \in\}$
(D) $\{S' \rightarrow eS, S' \rightarrow \varepsilon\}\}$ and $\{S' \rightarrow \in\}$

12. Consider the grammar $S \rightarrow CC$, $C \rightarrow cC/d$.
The grammar is
(A) LL (1)
(B) SLR (1) but not LL (1)
(C) LALR (1) but not SLR (1)
(D) LR (1) but not LALR (1)

13. Consider the grammar
$E \rightarrow E + n/E - n/n$
For a sentence $n + n - n$, the handles in the right sentential form of the reduction are
(A) n, $E + n$ and $E + n - n$
(B) n, $E + n$ and $E + E - n$
(C) n, $n + n$ and $n + n - n$
(D) n, $E + n$ and $E - n$

14. A top down parser uses ___ derivation.
(A) Left most derivation
(B) Left most derivation in reverse
(C) Right most derivation
(D) Right most derivation in reverse

15. Which of the following statement is false?
(A) An unambiguous grammar has single leftmost derivation.
(B) An LL (1) parser is topdown.
(C) LALR is more powerful than SLR.
(D) An ambiguous grammar can never be LR (K) for any k.

16. Merging states with a common core may produce ___ conflicts in an LALR parser.
(A) Reduce – reduce
(B) Shift – reduce
(C) Both (A) and (B)
(D) None of these

17. LL (K) grammar
(A) Has to be CFG
(B) Has to be unambiguous
(C) Cannot have left recursion
(D) All of these

18. The I_0 state of the LR (0) items for the grammar
$S \rightarrow AS/b$
$A \rightarrow SA/a$.
(A) $S' \rightarrow .S$
$S \rightarrow .As$
$S \rightarrow .b$
$A \rightarrow .SA$
$A \rightarrow .a$
(B) $S \rightarrow .AS$
$S \rightarrow .b$
$A \rightarrow .SA$
$A \rightarrow .a$
(C) $S \rightarrow .AS$
$S \rightarrow .b$
(D) $S \rightarrow A$
$A \rightarrow .SA$
$A \rightarrow .a$

19. In the predictive parsing table for the grammar:
$S \rightarrow FR$
$R \rightarrow \times S/e$
$F \rightarrow$ id
What will be the entry for [S, id]?
(A) $S \rightarrow FR$
(B) $F \rightarrow$ id
(C) Both (A) and (B)
(D) None of these

Previous Years' Questions

1. Consider the grammar:
$S \rightarrow (S) \mid a$
Let the number of states in SLR (1), LR (1) and LALR (1) parsers for the grammar be n_1, n_2 and n_3 respectively. The following relationship holds good: **[2005]**
(A) $n_1 < n_2 < n_3$ (B) $n_1 = n_3 < n_2$
(C) $n_1 = n_2 = n_3$ (D) $n_1 \geq n_3 \geq n_2$

2. Consider the following grammar:
$S \rightarrow S * E$
$S \rightarrow E$
$E \rightarrow F + E$
$E \rightarrow F$
$F \rightarrow$ id
Consider the following LR (0) items corresponding to the grammar above.
(i) $S \rightarrow S * .E$
(ii) $E \rightarrow F. + E$
(iii) $E \rightarrow F + .E$
Given the items above, which two of them will appear in the same set in the canonical sets-of items for the grammar? **[2006]**
(A) (i) and (ii) (B) (ii) and (iii)
(C) (i) and (iii) (D) None of the above

3. Consider the following statements about the context-free grammar
$G = \{S \rightarrow SS, S \rightarrow ab, S \rightarrow ba, S \rightarrow \in\}$
(i) G is ambiguous
(ii) G produces all strings with equal number of a's and b's
(iii) G can be accepted by a deterministic PDA.
Which combination below expresses all the true statements about G? **[2006]**
(A) (i) only (B) (i) and (iii) only
(C) (ii) and (iii) only (D) (i), (ii) and (iii)

4. Consider the following grammar:
$S \rightarrow FR$
$R \rightarrow *S \mid \varepsilon$
$F \rightarrow$ id
In the predictive parser table, M, of the grammar the entries M[S, id] and M[R, $] respectively. **[2006]**
(A) $\{S \rightarrow FR\}$ and $\{R \rightarrow \varepsilon\}$
(B) $\{S \rightarrow FR\}$ and $\{\ \}$
(C) $\{S \rightarrow FR\}$ and $\{R \rightarrow *S\}$
(D) $\{F \rightarrow \text{id}\}$ and $\{R \rightarrow \varepsilon\}$

5. Which one of the following grammars generates the language $L = \{a^i b^j \mid i \neq j\}$? **[2006]**
(A) $S \rightarrow AC \mid CB$
$C \rightarrow aCb \mid a \mid b$
$A \rightarrow aA \mid \in$
$B \rightarrow Bb \mid \in$
(B) $S \rightarrow aS \mid Sb \mid a \mid b$
(C) $S \rightarrow AC \mid CB$
$C \rightarrow aC \mid b \mid \in$
$A \rightarrow aA \mid \in$
$B \rightarrow Bb \mid \in$
(D) $S \rightarrow AC \mid CB$
$C \rightarrow aCb \mid \in$
$A \rightarrow aA \mid a$
$B \rightarrow Bb \mid b$

6. In the correct grammar above, what is the length of the derivation (number of steps starting from S) to generate the string $a^\ell b^m$ with $\ell \neq m$? **[2006]**
(A) max (l, m) + 2
(B) l + m + 2
(C) l + m + 3
(D) max (l, m) + 3

7. Which of the following problems is undecidable? **[2007]**
(A) Membership problem for CFGs.
(B) Ambiguity problem for CFGs.
(C) Finiteness problem for FSAs.
(D) Equivalence problem for FSAs.

8. Which one of the following is a top-down parser? **[2007]**
(A) Recursive descent parser.
(B) Operator precedence parser.
(C) An LR (k) parser.
(D) An LALR (k) parser.

9. Consider the grammar with non-terminals $N = \{S, C, \text{and } S_1\}$, terminals $T = \{a, b, i, t, e\}$ with S as the start symbol, and the following set of rules: **[2007]**
$S \rightarrow iCtSS_1/a$
$S_1 \rightarrow eS/\varepsilon$
$C \rightarrow b$
The grammar is NOT LL (1) because:
(A) It is left recursive
(B) It is right recursive
(C) It is ambiguous
(D) It is not context-free.

10. Consider the following two statements:
P: Every regular grammar is LL (1)
Q: Every regular set has a LR (1) grammar
Which of the following is TRUE? **[2007]**
(A) Both P and Q are true
(B) P is true and Q is false
(C) P is false and Q is true
(D) Both P and Q are false

Common data for questions 11 and 12: Consider the CFG with $\{S, A, B\}$ as the non-terminal alphabet, $\{a, b\}$ as the terminal alphabet, S as the start symbol and the following set of production rules:

$S \rightarrow aB$	$S \rightarrow bA$
$B \rightarrow b$	$A \rightarrow a$
$B \rightarrow bS$	$A \rightarrow aS$
$B \rightarrow aBB$	$S \rightarrow bAA$

11. Which of the following strings is generated by the grammar? **[2007]**
(A) *aaaabb* (B) *aabbbb*
(C) *aabbab* (D) *abbbba*

12. For the correct answer strings to Q.78, how many derivation trees are there? **[2007]**
(A) 1 (B) 2
(C) 3 (D) 4

13. Which of the following describes a handle (as applicable to LR-parsing) appropriately? **[2008]**
(A) It is the position in a sentential form where the next shift or reduce operation will occur.
(B) It is non-terminal whose production will be used for reduction in the next step.
(C) It is a production that may be used for reduction in a future step along with a position in the sentential form where the next shift or reduce operation will occur.
(D) It is the production p that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found.

14. Which of the following statements are true?
(i) Every left-recursive grammar can be converted to a right-recursive grammar and vice-versa
(ii) All ε-productions can be removed from any context-free grammar by suitable transformations
(iii) The language generated by a context-free grammar all of whose productions are of the form $X \rightarrow w$ or $X \rightarrow wY$ (where, w is a string of terminals and Y is a non-terminal), is always regular
(iv) The derivation trees of strings generated by a context-free grammar in Chomsky Normal Form are always binary trees **[2008]**
(A) (i), (ii), (iii) and (iv) (B) (ii), (iii) and (iv) only
(C) (i), (iii) and (iv) only (D) (i), (ii) and (iv) only

15. An LALR (1) parser for a grammar G can have shift-reduce (S–R) conflicts if and only if **[2008]**
(A) The SLR (1) parser for G has S–R conflicts
(B) The LR (1) parser for G has S–R conflicts
(C) The LR (0) parser for G has S–R conflicts
(D) The LALR (1) parser for G has reduce-reduce conflicts

16. $S \rightarrow aSa|bSb|a|b$;
The language generated by the above grammar over the alphabet $\{a, b\}$ is the set of **[2009]**
(A) All palindromes.
(B) All odd length palindromes.
(C) Strings that begin and end with the same symbol.
(D) All even length palindromes.

17. Which data structure in a compiler is used for managing information about variables and their attributes? **[2010]**
(A) Abstract syntax tree
(B) Symbol table
(C) Semantic stack
(D) Parse table

18. The grammar $S \rightarrow aSa|bS|c$ is **[2010]**
(A) LL (1) but not LR (1)
(B) LR (1) but not LR (1)
(C) Both LL (1) and LR (1)
(D) Neither LL (1) nor LR (1)

19. The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense? **[2011]**
(A) Finite state automata
(B) Deterministic pushdown automata
(C) Non-deterministic pushdown automata
(D) Turing machine

Common data for questions 20 and 21: For the grammar below, a partial LL (1) parsing table is also presented along with the grammar. Entries that need to be filled are indicated as E_1, E_2, and E_3. Is the empty string, \$ indicates end of input, and, I separates alternate right hand side of productions

$S \rightarrow a\ A\ b\ B|b\ A\ a\ B|\ e$

$A \rightarrow S$

$B \rightarrow S$

	a	**b**	**$**
S	E_1	E_2	$S \rightarrow e$
A	$A \rightarrow S$	$A \rightarrow S$	Error
B	$B \rightarrow S$	$B \rightarrow S$	E_3

20. The FIRST and FOLLOW sets for the non-terminals A and B are **[2012]**
(A) FIRST $(A) = \{a, b, \varepsilon\}$ = FIRST (B)
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \{a, b, \$\}$
(B) FIRST $(A) = \{a, b, \$\}$
FIRST $(B) = \{a, b, \varepsilon\}$
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \{\$\}$
(C) FIRST $(A) = \{a, b, \varepsilon\}$ = FIRST (B)
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \varnothing$
(D) FIRST $(A) = \{a, b\}$ = FIRST (B)
FOLLOW $(A) = \{a, b\}$
FOLLOW $(B) = \{a, b\}$

21. The appropriate entries for E_1, E_2, and E_3 are **[2012]**
(A) E_1: $S \rightarrow a\ A\ b\ B$, $A \rightarrow S$
E_2: $S \rightarrow b\ A\ a\ B$, $B \rightarrow S$
E_3: $B \rightarrow S$

(B) $E_1: S \to a\ A\ b\ B, S \to \varepsilon$
$E_2: S \to b\ A\ a\ B, S \to \varepsilon$
$E_3: S \to \in$

(C) $E_1: S \to a\ A\ b\ B, S \to \varepsilon$
$E_2: S \to b\ A\ a\ B, S \to \varepsilon$
$E_3: B \to S$

(D) $E_1: A \to S, S \to \varepsilon$
$E_2: B \to S, S \to \varepsilon$
$E_3: B \to S$

22. What is the maximum number of reduce moves that can be taken by a bottom-up parser for a grammar with no epsilon-and unit-production (i.e., of type $A \to \in$ and $A \to a$) to parse a string with n tokens? **[2013]**

(A) $n/2$ (B) $n - 1$
(C) $2n - 1$ (D) 2^n

23. Which of the following is/are undecidable?

(i) G is a CFG. Is $L(G) = \phi$?
(ii) G is a CFG, Is $L(G) = \Sigma^*$?
(iii) M is a Turing machine. Is $L(M)$ regular?
(iv) A is a DFA and N is an NFA. Is $L(A) = L(N)$? **[2013]**

(A) (iii) only
(B) (iii) and (iv) only
(C) (i), (ii) and (iii) only
(D) (ii) and (iii) only

24. Consider the following two sets of LR (1) items of an LR (1) grammar. **[2013]**

$X \to c.X, c/d$ $X \to c.X, \$$
$X \to .cX, c/d$ $X \to .cX, \$$
$X \to .d, c/d$ $X \to .d, \$$

Which of the following statements related to merging of the two sets in the corresponding LALR parser is/are FALSE?

(i) Cannot be merged since look - ahead are different.
(ii) Can be merged but will result in S-R conflict.
(iii) Can be merged but will result in R-R conflict.
(iv) Cannot be merged since goto on c will lead to two different sets.

(A) (i) only (B) (ii) only
(C) (i) and (iv) only (D) (i), (ii), (iii) and (iv)

25. A canonical set of items is given below

$S \to L. > R$
$Q \to R.$

On input symbol < the sset has **[2014]**

(A) A shift–reduce conflict and a reduce–reduce conflict.
(B) A shift–reduce conflict but not a reduce–reduce conflict.
(C) A reduce–reduce conflict but not a shift reduce conflict.
(D) Neither a shift–reduce nor a reduce–reduce conflict.

26. Consider the grammar defined by the following production rules, with two operators * and +

$S \to T * P$
$T \to U | T * U$
$P \to Q + P | Q$
$Q \to Id$
$U \to Id$

Which one of the following is TRUE? **[2014]**

(A) + is left associative, while * is right associative
(B) + is right associative, while * is left associative
(C) Both + and * are right associative.
(D) Both + and * are left associative

27. Which one of the following problems is undecidable? **[2014]**

(A) Deciding if a given context -free grammar is ambiguous.
(B) Deciding if a given string is generated by a given context-free grammar.
(C) Deciding if the language generated by a given context-free grammar is empty.
(D) Deciding if the language generated by a given context free grammar is finite.

28. Which one of the following is TRUE at any valid state in shift-reduce parsing? **[2015]**

(A) Viable prefixes appear only at the bottom of the stack and not inside.
(B) Viable prefixes appear only at the top of the stack and not inside.
(C) The stack contains only a set of viable prefixes.
(D) The stack never contains viable prefixes.

29. Among simple LR (SLR), canonical LR, and look-ahead LR (LALR), which of the following pairs identify the method that is very easy to implement and the method that is the most powerful, in that order? **[2015]**

(A) SLR, LALR
(B) Canonical LR, LALR
(C) SLR, canonical LR
(D) LALR, canonical LR

30. Consider the following grammar G

$S \to F \mid H$
$F \to p \mid c$
$H \to d \mid c$

Where S, F and H are non-terminal symbols, p, d and c are terminal symbols. Which of the following statement(s) is/are correct? **[2015]**

S_1. LL(1) can parse all strings that are generated using grammar G

S_2. LR(1) can parse all strings that are generated using grammar G

(A) Only S_1 (B) Only S_2
(C) Both S_1 and S_2 (D) Neither S_1 nor S_2

31. Match the following: **[2016]**

(P) Lexical analysis	(i) Leftmost derivation
(Q) Top down parsing	(ii) Type checking
(R) Semantic analysis	(iii) Regular expressions
(S) Runtime environments	(iv) Activation records

(A) $P \leftrightarrow$ i, $Q \leftrightarrow$ ii, $R \leftrightarrow$ iv, $S \leftrightarrow$ iii
(B) $P \leftrightarrow$ iii, $Q \leftrightarrow$ i, $R \leftrightarrow$ ii, $S \leftrightarrow$ iv
(C) $P \leftrightarrow$ ii, $Q \leftrightarrow$ iii, $R \leftrightarrow$ i, $S \leftrightarrow$ iv
(D) $P \leftrightarrow$ iv, $Q \leftrightarrow$ i, $R \leftrightarrow$ ii, $S \leftrightarrow$ iii

32. A student wrote two context - free grammars **G1** and **G2** for generating a single C-like array declaration. The dimension of the array is at least one.

For example, int a [10] [3];

The grammars use D as the start symbol, and use six terminal symbols int; id [ ] num. **[2016]**

Grammar **G1**	Grammar **G2**
D → **int** L;	D → **int**L;
L → **id** [E	L → **id** E
E → **num**]	E → E [**num**]
E → **num**] [E	E → [**num**]

Which of the grammars correctly generate the declaration mentioned above?

(A) Both **G1** and **G2**
(B) Only **G1**
(C) Only **G2**
(D) Neither **G1** nor **G2**

33. Consider the following grammar:

$$P \rightarrow xQRS$$
$$Q \rightarrow yz \mid z$$
$$R \rightarrow w \mid \varepsilon$$
$$S \rightarrow y$$

What is FOLLOW (Q)? **[2017]**

(A) $\{R\}$ (B) $\{w\}$
(C) $\{w, y\}$ (D) $\{w, \$\}$

34. Which of the following statements about parser is/are CORRECT? **[2017]**

I. Canonical LR is more powerful than SLR.
II. SLR is more powerful than LALR.
III. SLR is more powerful than Canonical LR.

(A) I only (B) II only
(C) III only (D) I and III only

35. Consider the following expression grammar G :

E – > E – T | T
T – > T + F | F
F – > (E) | id

Which of the following grammars is not left recursive, but is equivalent to G? **[2017]**

(A) E – >E – T | T
T – > T + F | F
F – > (E) | id

(B) E – > TE
E′ – > –TE | ∈
T – > T + F | F
F – > (E) | id

(C) E – > TX
X – > –TX | ∈
T – > FY
Y – > + FY | ∈
F – > (E) | id

(D) E – > TX | (TX)
X – > –TX | +TX | ∈
T – > id

36. Which one of the following statements is FALSE? **[2018]**

(A) Context-free grammar can be used to specify both lexical and syntax rules.
(B) Type checking is done before parsing.
(C) High-level language programs can be translated to different Intermediate Representations.
(D) Arguments to a function can be passed using the program stack.

37. A lexical analyzer uses the following patterns to recognize three tokens T_1, T_2, and T_3 over the alphabet $\{a, b, c\}$.

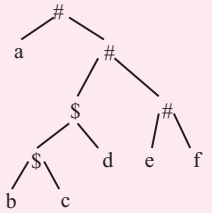
T_1: $a?(b|c)^*a$
T_2: $b?(a|c)^*b$
T_3: $c?(b|a)^*c$

Note that 'x?' means 0 or 1 occurrence of the symbol x. Note also that the analyzer outputs the token that matches the longest possible prefix.

If the string bbaacabc is processed by the analyzer, which one of the following is the sequence of tokens it outputs? **[2018]**

(A) $T_1T_2T_3$ (B) $T_1T_1T_3$
(C) $T_2T_1T_3$ (D) T_3T_3

38. Consider the following parse tree for the expression a#bcd#e#f, involving two binary operators $ and #.



Which one of the following is correct for the given parse tree? **[2018]**

(A) $ has higher precedence and is left associative; # is right associative
(B) # has higher precedence and is left associative; $ is right associative
(C) $ has higher precedence and is left associative; # is left associative
(D) # has higher precedence and is right associative; $ is left associative

Answer Keys

Exercises

Practice Problems 1

1. D	**2.** A	**3.** B	**4.** A	**5.** D	**6.** C	**7.** C	**8.** D	**9.** C	**10.** A
11. C	**12.** C	**13.** A	**14.** B	**15.** D					

Practice Problems 2

1. D	**2.** C	**3.** B	**4.** A	**5.** B	**6.** B	**7.** A	**8.** C	**9.** C	**10.** A
11. D	**12.** A	**13.** D	**14.** A	**15.** D	**16.** A	**17.** C	**18.** A	**19.** A	

Previous Years' Questions

1. B	**2.** D	**3.** B	**4.** A	**5.** D	**6.** A	**7.** B	**8.** A	**9.** C	**10.** A
11. C	**12.** B	**13.** D	**14.** C	**15.** B	**16.** B	**17.** B	**18.** C	**19.** A	**20.** A
21. C	**22.** B	**23.** D	**24.** D	**25.** D	**26.** B	**27.** A	**28.** C	**29.** C	**30.** D
31. B	**32.** A	**33.** C	**34.** A	**35.** C	**36.** B	**37.** D	**38.** A		

Chapter 2

Syntax Directed Translation

LEARNING OBJECTIVES

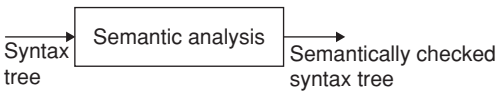
- *Syntax directed translation*
- *Syntax directed definition*
- *Dependency graph*
- *Constructing syntax trees for expressions*
- *Types of SDD's*
- *S-attributed definition*
- *L-attributed definitions*
- *Synthesized attributes on the parser*
- *Syntax directed translation schemes*
- *Bottom up evaluation of inherited attributes*

Syntax Directed Translation

To translate a programming language construct, a compiler may need to know the type of construct, the location of the first instruction, and the number of instructions generated…etc. So, we have to use the term 'attributes' associated with constructs.

An attribute may represent type, number of arguments, memory location, compatibility of variables used in a statement which cannot be represented by CFG alone.

So, we need to have one more phase to do this, i.e., 'semantic analysis' phase.



In this phase, for each production CFG, we will give some semantic rule.

Syntax directed translation scheme

A CFG in which a program fragment called output action (semantic action or semantic rule) is associated with each production is known as Syntax Directed Translation Scheme.

These semantic rules are used to

1. Generate intermediate code.
2. Put information into symbol table.
3. Perform type checking.
4. Issues error messages.

Notes:

1. Grammar symbols are associated with attributes.
2. Values of the attributes are evaluated by the semantic rules associated with production rules.

Notations for Associating Semantic Rules

There are two techniques to associate semantic rules:

Syntax directed definition (SDD) It is high level specification for translation. They hide the implementation details, i.e., the order in which translation takes place.

Attributes + CFG + Semantic rules = Syntax directed definition (SDD).

Translation schemes These schemes indicate the order in which semantic rules are to be evaluated. This is an input and output mapping.

Syntax Directed Definitions

A SDD is a generalization of a CFG in which each grammar symbol is associated with a set of attributes.

There are two types of set of attributes for a grammar symbol.

1. Synthesized attributes
2. Inherited attributes

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

Semantic rules setup dependencies between attributes which can be represented by a dependency graph.

The 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 a value.

Attribute grammar: An attribute grammar is a syntax directed definition in which the functions in semantic rules 'cannot have side effects'.

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 attribute values at the nodes is called annotating (or decorating) of the parse tree.

In a SDD, each production $A \rightarrow \propto$ is associated with a set of semantic rules of the form:

$b = f(c_1, c_2, \ldots c_n)$ where

$f : A$ function

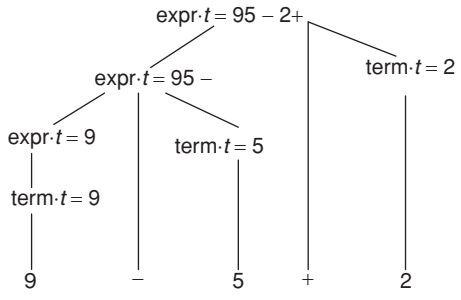
b can be one of the following:

b is a 'synthesized attribute' of A and $c_1, c_2, \ldots c_n$ are attributes of the grammar symbols in $A \rightarrow \propto$.

The value of a 'synthesized attribute' at a node is computed from the value of attributes at the children of that node in the parse tree.

Example:

Production	Semantic Rule
expr → expr1 + term	expr.t: = expr1.t||term.t||'+'
expr → expr1 – term	expr.t: = expr1.t||term.t||'–'
expr → term	expr.t: = term.t
term → 0	term.t: = '0'
term → 1	term.t: = '1'
⋮	⋮
term → 9	term.t: = '9'



b is an 'inherited attribute' of one of the grammar symbols on the right side of the production.

An 'inherited attribute' is one whose value at a node is defined in terms of attributes at the parent and/or siblings of that node. It is used for finding the context in which it appears.

Example: An inherited attribute distributes type information to the various identifiers in a declaration.

For the grammar

$D \rightarrow$ TL

$T \rightarrow$ int

$T \rightarrow$ real

$L \rightarrow L_1$, id

$L \rightarrow$ id

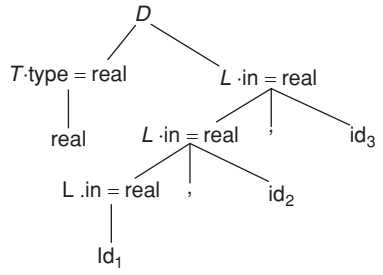
That is, The keyword int or real followed by a list of identifiers.

In this T has synthesized attribute type: T.type. L has an inherited attribute in L.in

Rules associated with L call for procedure add type to the type of each identifier to its entry in the symbol table.

Production	Semantic Rule
$D \rightarrow$ TL	L.in = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ real	T.type = real
$L \rightarrow L_1$, id	addtype L_1.in = L.in(id.entry, L.in)
$L \rightarrow$ id	addtype (id.entry, L.in)

The annotated parse tree for the sentence real id_1, id_2, id_3 is shown below:



Synthesized Attribute

The value of a synthesized attribute at a node is computed from the value of attributes at the children of that node in a parse tree. Consider the following grammar:

$L \rightarrow E_n$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

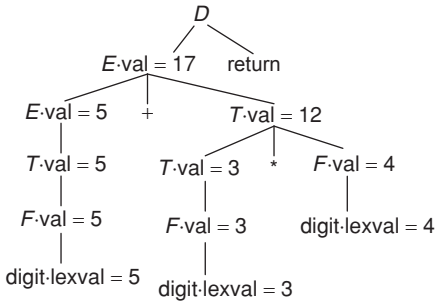
$F \rightarrow$ digit.

Let us consider synthesized attribute value with each of the non-terminals E, T and F.

Token digit has a synthesized attribute lexical supplied by lexical analyzer.

Production	Semantic Rule
$L \rightarrow E_n$	print (E.val)
$E \rightarrow E_1 + T$	E.val: = E_1.val + T.val
$E \rightarrow T$	E.val: = T_1.val
$T \rightarrow T_1 * F$	T.val: = T_1.val*F.val
$T \rightarrow F$	T.val: = F.val
$F \rightarrow (E)$	F.val: = E.val
$F \rightarrow$ digit	F.val: = digit.lexval

The Annotated parse tree for the expression 5 + 3 * 4 is shown below:



Example 1: Consider an example, which shows semantic rules for Infix to posfix translation:

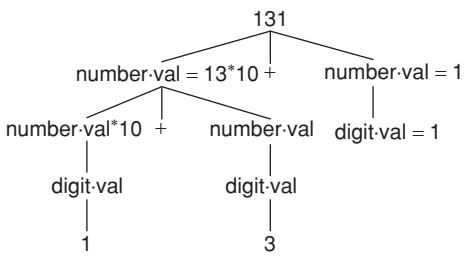
Production	Semantic Rules
expr → expr1 + term	expr.t: = expr1.t||term.t||'+'
expr → expr1 – term	expr.t: = expr1.t||term.t ||'–'
expr → term	expr.t: = term.t
term → 0	term.t: = '0'
⋮	⋮
term→ 9	term.t := '9'

Example 2: Write a SDD for the following grammar to determine number.val.

number → number digit
digit→ 0|1|…9

$$\begin{cases} \text{digit.val} := \text{'0'} \\ \text{digit.val} := \text{'1'} \\ \vdots \\ \text{digit.val} = \text{'9'} \end{cases}$$

number.val:=number.val * 10 + digit.val
Annotated tree for 131 is

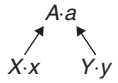


Dependency Graph

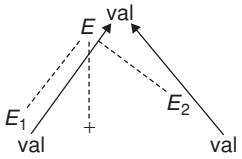
The interdependencies among the attributes at the nodes in a parse tree can be depicted by a directed graph called dependency graph.

- Synthesized attributes have edges pointing upwards.
- Inherited attributes have edges pointing downwards and/or sidewise.

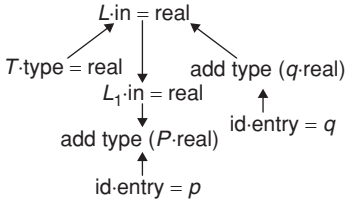
Example 1: $A.a := f(X.x, Y.y)$ is a semantic rule for $A \rightarrow XY$. For each semantic rule that consists of a procedure call:



Example 2:



Example 3: real p, q;



Evaluation order

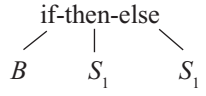
A topological sort of directed acyclic graph is an ordering $m_1, m_2, \ldots m_k$ of nodes of the graph S. t edges go from nodes earlier in the ordering to later nodes.

$m_i \rightarrow m_j$ means m_i appears before m_j in the ordering.

If $b := f(c_1, c_2, \ldots, c_k)$, the dependent attributes $c_1, c_2, \ldots c_k$ are available at node before f is evaluated.

Abstract syntax tree

It is a condensed form of parse tree useful for representing language constructs.
Example



Constructing Syntax Trees for Expressions

Each node in a syntax tree can be implemented as a record with several fields.

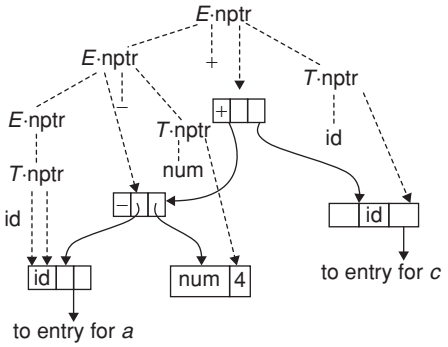
In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes for the operands.

1. mknode (op, left, right)
2. mkleaf (id, entry). Entry is a pointer to symbol table.
3. mkleaf (num, val)

Example:

Production	Semantic Rules
$E \to E_1 + T$	E.nptr := mknode ('+', E_1.nptr, T.nptr)
$E \to E_1 - T$	E.nptr := mknode ('–', E_1.nptr, T.nptr)
$E \to T$	E.nptr := T.nptr
$T \to (E)$	T.nptr := E.nptr
$T \to$ id	T.nptr := mkleaf(id, id.entry)
$T \to$ num	T.nptr := mkleaf(num, num.val)

Construction of a syntax tree for $a - 4 + c$



Types of SDD's

Syntax Directed definitions (SDD) are used to specify syntax directed translations. There are two types of SDD.

1. S-Attributed Definitions
2. L-Attributed Definitions.

S-attributed definitions

- Only synthesized attributes used in syntax direct definition.
- S-attributed grammars interact well with *LR* (*K*) parsers since the evaluation of attributes is bottom-up. They do not permit dependency graphs with cycles.

L-attributed definitions

- Both inherited and synthesized attribute are used.
- L-attributed grammar support the evaluation of attributes associated with a production body, dependency–graph edges can go from left to right only.
- Each S-attributed grammar is also a L-attributed grammar.
- L-attributed grammars can be incorporated conveniently in top down parsing.
- These grammars interact well with *LL* (*K*) parsers (both table driven and recursive descent).

Synthesized Attributes on the Parser Stack

A translator for an S-attributed definition often be implemented with *LR* parser generator. Here the stack is implemented by a pair of array state and val.

- Each state entry is pointed to a *LR* (1) parsing table.
- Each val[i] holds the value of the attributes associated with the node. For $A \to xyz$, the stack will be:

	State	Val
Top →	Z	Z.z
	Y	Y.y
	X	X.x

Example: Consider the following grammar:

$S \to E\$$	{print(E.val)}
$E \to E + E$	{E.val := E.val + E.val}
$E \to E^*E$	{E.val := E.val * E.val}
$E \to (E)$	{E.val := E.val}
$E \to I$	{I.val := I.val * 10 + digit}
$I \to I$ digit	
$I \to$ digit	{I.val := digit}

Implementation

$S \to E\$$	print (val [top])
$E \to E + E$	val[ntop] := val[top] + val[top-2]
$E \to E^*E$	val[ntop] := val[top] * val[top-2]
$E \to (E)$	val[ntop] := val[top-1]
$E \to I$	val[ntop] := val[top]
$I \to I$ digit	val[ntop] := 10*val[top] + digit
$I \to$ digit	val[ntop] := digit

L-attributed Definitions

A syntax directed definition is *L*-attributed if each inherited attribute of X_j, $1 \le j \le n$, on the right side of $A \to X_1 X_2 \ldots X_n$, depends only on

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

Every S-attributed definition is L-attributed, because the above two rules apply only to the inherited attributes.

Syntax Directed Translation Schemes

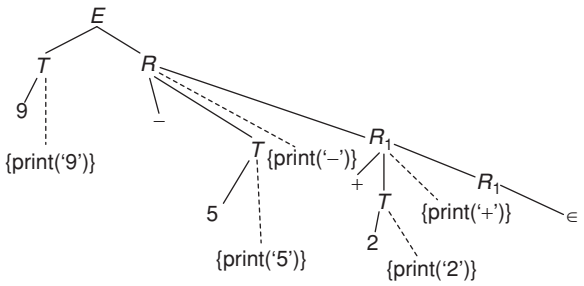
A translation scheme is a CFG in which attributes are associated with grammar symbols and semantic actions are enclosed between braces { } are inserted within the right sides of productions.

Example: $E \to TR$

$R \to$ op T {print (op.lexeme)} $R_1 | \in$

$T \to$ num {print (num.val)}

Using this, the parse tree for 9 – 5 + 2 is



If we have both inherited and synthesized attributes then we have to follow the following rules:

1. An inherited attribute for a symbol on the right side of a production must be computed in an action before that symbol.
2. An action must not refer to a synthesized attribute of a symbol on the right side of the action.
3. A synthesized attribute for the non–terminal on the left can only be computed after all attributes it references, have been computed.

Note: In the implementation of L-attributed definitions during predictive parsing, instead of syntax directed translations, we will work with translation schemes.

Eliminating left recursion from translation scheme

Consider following grammar, which has left recursion

$E \rightarrow E + T$ {print ('+') ;}

When transforming the grammar, treat the actions as if they were terminal symbols. After eliminating recursion from the above grammar.

$E \rightarrow TR$
$R \rightarrow +T$ {print ('+');} R
$R \rightarrow \in$

Bottom-up Evaluation of Inherited Attributes

- Using a bottom up translation scheme, we can implement any L-attributed definition based on LL (1) grammar.
- We can also implement some of L-attributed definitions based on LR (1) using bottom up translations scheme.
  - The semantic actions are evaluated during the reductions.
  - During the bottom up evaluation of S-attributed definitions, we have a parallel stack to hold synthesized attributes.

Where are we going to hold inherited attributes?

We will convert our grammar to an equivalent grammar to guarantee the following:

- All embedding semantic actions in our translation scheme will be moved to the end of the production rules.
- All inherited attributes will be copied into the synthesized attributes (may be new non-terminals).

Thus we will evaluate all semantic actions during reductions, and we find a place to store an inherited attribute. The steps are

1. Remove an embedding semantic action S_i, put new non-terminal M_i instead of that semantic action.
2. Put S_i into the end of a new production rule $M_i \rightarrow \in$.
3. Semantic action S_i will be evaluated when this new production rule is reduced.
4. Evaluation order of semantic rules is not changed. i.e., if

$$A \rightarrow \{S_1\}\ X_1\{S_2\}X_2 \ldots \{S_n\}X_n$$

After removing embedding semantic actions:

$A \rightarrow M_1X_1M_2X_2 \ldots M_nX_n$
$M_1 \rightarrow \in \{S_1\}$
$M_2 \rightarrow \in \{S_2\}$
$\vdots$
$M_n \rightarrow \in \{S_n\}$

For example,

$E \rightarrow TR$
$R \rightarrow +T$ {print ('+')} R_1
$R \rightarrow \in$
$T \rightarrow$ id {print (id.name)}
$\Downarrow$ remove embedding semantic actions
$E \rightarrow TR$
$R \rightarrow +TMR_1$
$R \rightarrow \in$
$T \rightarrow$ id {print (id.name)}
$M \rightarrow \in$ {print ('+')}

Translation with inherited attributes

Let us assume that every non-terminal A has an inherited attribute $A.i$ and every symbol X has a synthesized attribute $X.s$ in our grammar.

For every production rule $A \rightarrow X_1, X_2 \ldots X_n$, introduce new marker non-terminals

$M_1, M_2, \ldots M_n$ and replace this production rule with $A \rightarrow M_1X_1M_2X_2 \ldots M_nX_n$

The synthesized attribute of X_i will not be changed.

The inherited attribute of X_i will be copied into the synthesized attribute of M_i by the new semantic action added at the end of the new production rule

$$M_i \rightarrow \in$$

Now, the inherited attribute of X_i can be found in the synthesized attribute of M_i.

$A \rightarrow \{B.i = f_1(\ldots)\ B\ \{\ c.i = f_2(\ldots)\}\ c\ \{A.s = f_3(\ldots)\}$

$\Downarrow$

$A \rightarrow \{M_1.i = f_1(\ldots)\}\ M_1\ \{B.i = M_1.s\}\ B\ \{M_2.i = f_2(\ldots)\}M_2$
$\{c.i = M_2.S\}\ c\ \{A.s = f_3(\ldots)\}$
$M_1 \rightarrow \in\ \{M_1.s = M_1.i\}$
$M_2 \rightarrow \in\ \{M_2.s = M_2.i\}$

EXERCISES

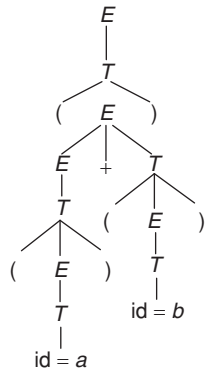
Practice Problems I

Directions for questions 1 to 13: Select the correct alternative from the given choices.

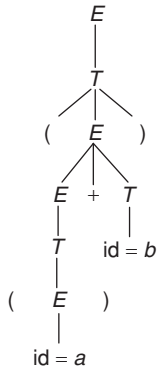
1. The annotated tree for input ((*a*) + (*b*)), for the rules given below is

Production	Semantic Rule
$E \to E + T$	$ $ = mknode ('+', $1, $3)
$E \to E - T$	$ $ = mknode ('–', $1, $3)
$E \to T$	$ $ = $1;
$T \to (E)$	$ $ = $2;
$T \to$ id	$ $ = mkleaf (id, $1)
$T \to$ num	$ $ = mkleaf (num, $1)

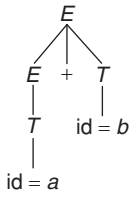
(A)



(B)



(C)



(D) None of these

2. Let synthesized attribute val give the value of the binary number generated by S in the following grammar.

$S \to L\ L$
$S \to L$
$L \to LB$
$L \to B$
$B \to 0$
$B \to 1$

Input 101.101, S.val = 5.625
use synthesized attributes to determine S.val
Which of the following are true?

(A) $S \to L_1.L_2$ {S.val = L_1.val + L_2.val/ (2**L_2.bits)
$|L$ {S.val = L.val; S.bits = L.bits}

(B) $L \to L_1\ B$ {L.val = L_1.val*2 + B.val;
L.bits = L_1.bits + 1}
$|B$ {L.val = B.val; L.bits = 1}

(C) $B \to 0$ {B.val = 0}
$|1$ {B.val = 1}

(D) All of these

3. Which of the following productions with translation rules converts binary number representation into decimal.

(A)

Production	Semantic Rule
$B \to 0$	B.trans = 0
$B \to 1$	B.trans = 1
$B \to B_0$	B_1.trans = B_2.trans*2
$B \to B_1$	B_1.trans = B_2.trans * 2 + 1

(B)

Production	Semantic Rule
$B \to 0$	B.trans = 0
$B \to B_0$	B_1.trans = B_2.trans*4

(C)

Production	Semantic Rule
$B \to 1$	B.trans = 1
$B \to B_1$	B_1.trans = B_2.trans*2

(D) None of these

4. The grammar given below is

Production	Semantic Rule
$A \to$ LM	L.i := l(A. i)
	M.i := m(L.s)
	A.s := f(M.s)
A → QR	R.i := r(A.i)
	Q.i := q(R.s)
	A.s := f(Q.s)

(A) A L-attributed grammar
(B) Non-L-attributed grammar
(C) Data insufficient
(D) None of these

5. Consider the following syntax directed translation:

$S \to aS$ {$m := m + 3$; print (m);}
$|bS$ {m: = m*2; print (m) ;}
$|\in$ {m: = 0 ;}

A shift reduce parser evaluate semantic action of a production whenever the production is reduced.

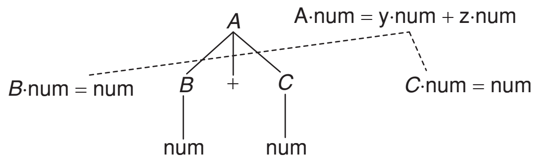
If the string is = *a a b a b b* then which of the following is printed?

(A) 0 0 3 6 9 12 (B) 0 0 0 3 6 9 12
(C) 0 0 0 3 6 9 12 15 (D) 0 0 3 9 6 12

6. Which attribute can be evaluated by shift reduce parser that execute semantic actions only at reduce moves but never at shift moves?

(A) Synthesized attribute (B) Inherited attribute
(C) Both (a) and (b) (D) None of these

7. Consider the following annotated parse tree:



Which of the following is true for the given annotated tree?

(A) There is a specific order for evaluation of attribute on the parse tree.
(B) Any evaluation order that computes an attribute 'A' after all other attributes which 'A' depends on, is acceptable.
(C) Both (A) and (B)
(D) None of these.

Common data for questions 8 and 9: Consider the following grammar and syntax directed translation.

$E \to E + T$	E_1.val = E_2.val + T.val
$E \to T$	E.val = T.val
$T \to T*P$	T_1.val = T_2.val * P.val * P.num
$T \to P$	T.val = P.val * P.num
$P \to (E)$	P.val = E.val
$P \to 0$	P.num = 1
	P.val = 2
$P \to 1$	P.num = 2
	P.val = 1

8. What is E.val for string 1*0?
(A) 8 (B) 6
(C) 4 (D) 12

9. What is the E.val for string 0 * 0 + 1?
(A) 8 (B) 6
(C) 4 (D) 12

10. Consider the following syntax directed definition:

Production	Semantic Rule
$S \to b$	$S.x = 0$ $S.y = 0$
$S \to S_1\ I$	$S.x = S_1.x + I.dx$ $S.y = S_1.y + I.dy$
$I \to$ east	$I.dx = 1$ $I.dy = 0$
$I \to$ north	$I.dx = 0$ $I.dy = 1$
$I \to$ west	$I.dx = -1$ $I.dy = 0$
$I \to$ south	$I.dx = 0$ $I.dy = -1$

If Input = begin east south west north, after evaluating this sequence what will be the value of $S.x$ and $S.y$?
(A) (1, 0) (B) (2, 0)
(C) (−1, −1) (D) (0, 0)

11. What will be the values $s.x$, $s.y$ if input is 'begin west south west'?
(A) (–2, –1)
(B) (2, 1)
(C) (2, 2)
(D) (3, 1)

12. Consider the following grammar:

$S \to E$	S.val = E.val
	E.num = 1
$E \to E*T$	E_1.val = 2 * E_2.val + 2 * T.val
	E_2.num = E_1.num + 1
	T.num = E_1.num + 1
$E \to T$	E.val = T.val
	T.num = E.num + 1
$T \to T + P$	T_1.val = T_2.val + P.val
	T_2.num = T_1.num + 1
	P.num = T_1.num + 1
$T \to P$	T.val = P.val
	P.num = T.num + 1
$P \to (E)$	P.val = E.val
$P \to i$	$\left\{\begin{array}{l} E\text{.num} = P\text{.num} \\ P\text{.val} = I\ \lvert P\text{.num} \end{array}\right\}$

Which attributes are inherited and which are synthesized in the above grammar?

(A) Num attribute is inherited attribute. Val attribute is synthesized attribute.
(B) Num is synthesized attribute. Val is inherited attribute.
(C) Num and val are inherited attributes.
(D) Num and value are synthesized attributes.

13. Consider the grammar with the following translation rules and E as the start symbol.

$E \to E_1 @ T$ {E.value = E_1.value*T.value}

|T {E.value = T.value}

$T \to T_1$ and F {T.value = T_1.value + F.value}

|F {T.value = F.value}

$F \to$ num {F.value = num.value}

Compute E.value for the root of the parse tree for the expression: 2 @ 3 and 5 @ 6 and 4

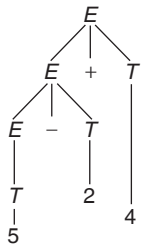
(A) 200 (B) 180
(C) 160 (D) 40

Practice Problems 2

Directions for questions 1 to 10: select the correct alternative from the given choices.

1. Consider the following Tree:

Production	Meaning
$E \rightarrow E_1 + T$	$E.t = E_1.t * T.t$
$E \rightarrow E_1 - T$	$E.t = E_1.t + T.t$
$E \rightarrow T$	$E.t = T.t$
$t \rightarrow 0$	$T.t =$ '0'
$t \rightarrow 5$	$T.t =$ '5'
$t \rightarrow 2$	$T.t =$ '2'
$t \rightarrow 4$	$T.t =$ '4'



After evaluation of the tree the value at the root will be:
(A) 28 (B) 32
(C) 14 (D) 7

2. The value of an inherited attribute is computed from the values of attributes at the _____
(A) Sibling nodes (B) Parent of the node
(C) Children node (D) Both (A) and (B)

3. Consider an action translating expression:

expr → expr + term {print ('+')}
expr → expr – term {print ('–')}
expr → → term
term → 1 {print ('1')}
term → 2 {print ('2')}
term → 3 {print ('3')}

Which of the following is true regarding the above translation expression?
(A) Action translating expression represents infix notation.
(B) Action translating expression represents prefix notation.
(C) Action translating expression represents postfix notation.
(D) None of these

4. In the given problem, what will be the result after evaluating 9 – 5 + 2?
(A) + – 9 5 2 (B) 9 – 5 + 2
(C) 9 5 – 2+ (D) None of these

5. In a syntax directed translation, if the value of an attribute node is a function of the values of attributes of children, then it is called:
(A) Synthesized attribute (B) Inherited attribute
(C) Canonical attributes (D) None of these

6. Inherited attribute is a natural choice in:
(A) Keeping track of variable declaration
(B) Checking for the correct use of L-values and R-values.
(C) Both (A) and (B)
(D) None of these

7. Syntax directed translation scheme is desirable because
(A) It is based on the syntax
(B) Its description is independent of any implementation.
(C) It is easy to modify
(D) All of these

8. A context free grammar in which program fragments, called semantic actions are embedded within right side of the production is called,
(A) Syntax directed translation
(B) Translation schema
(C) Annotated parse tree
(D) None of these

9. A syntax directed definition specifies translation of construct in terms of:
(A) Memory associated with its syntactic component
(B) Execution time associated with its syntactic component
(C) Attributes associated with its syntactic component
(D) None of these

10. If an error is detected within a statement, the type assigned to the Statement is:
(A) Error type (B) Type expression
(C) Type error (D) Type constructor

Previous Years' Questions

Common data for questions 1 (A) and 1 (B): Consider the following expression grammar. The semantic rules for expression evaluation are stated next to each grammar production: **[2005]**

$E \rightarrow$ number	E.val = number.val
$\mid E$ '+' E	$E^{(1)}$.val = $E^{(2)}$.val + $E^{(3)}$.val
$\mid E \rightarrow E$	$E^{(1)}$.val = $E^{(2)}$.val × $E^{(3)}$.val

1. (A) The above grammar and the semantic rules are fed to a yacc tool (which is an LALR (1) parser generator) for parsing and evaluating arithmetic expressions. Which one of the following is true about the action of yacc for the given grammar?
(A) It detects recursion and eliminates recursion
(B) It detects reduce-reduce conflict, and resolves

(C) It detects shift-reduce conflict, and resolves the conflict in favor of a shift over a reduce action.
(D) It detects shift-reduce conflict, and resolves the conflict in favor of a reduce over a shift action.

(B) Assume the conflicts in Part (A) of this question are resolved and an LALR (1) parser is generated for parsing arithmetic expressions as per the given grammar. Consider an expression 3 × 2 + 1. What precedence and associativity properties does the generated parser realize?
(A) Equal precedence and left associativity; expression is evaluated to 7
(B) Equal precedence and right associativity; expression is evaluated to 9
(C) Precedence of '×' is higher than that of '+', and both operators are left associative; expression is evaluated to 7
(D) Precedence of '+' is higher than that of '×', and both operators are left associative; expression is evaluated to 9

2. In the context of abstract-syntax-tree (AST) and control-flow-graph (CFG), which one of the following is TRUE? **[2015]**
(A) In both AST and CFG, let node N_2 be the successor of node N_1. In the input program, the code corresponding to N_2 is present after the code corresponding to N_1.
(B) For any input program, neither AST nor CFG will contain a cycle.
(C) The maximum number of successors of a node in an AST and a CFG depends on the input program.
(D) Each node in AST and CFG corresponds to at most one statement in the input program.

3. Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {S, A} and terminals {**a, b**}. **[2016]**

$S \rightarrow aA$ { print 1 }
$S \rightarrow a$ { print 2 }
$A \rightarrow Sb$ { print 3 }

Using the above SDTS, the output printed by a bottom-up parser, for the input **aab** is:
(A) 1 3 2 (B) 2 2 3
(C) 2 3 1 (D) syntax error

4. Which one of the following grammars is free from *left recursion*? **[2016]**
(A) $S \rightarrow AB$
$A \rightarrow Aa/b$
$B \rightarrow c$
(B) $S \rightarrow Ab/Bb/c$
$A \rightarrow Bd/\varepsilon$
$B \rightarrow e$
(C) $S \rightarrow Aa/B$
$A \rightarrow Bb/Sc/\varepsilon$
$B \rightarrow d$
(D) $S \rightarrow Aa/Bb/c$
$A \rightarrow Bd/\varepsilon$
$B \rightarrow Ae/\varepsilon$

Answer Keys

Exercises

Practice Problems 1

1. A **2.** D **3.** A **4.** B **5.** A **6.** A **7.** B **8.** C **9.** B **10.** D
11. A **12.** A **13.** C

Practice Problems 2

1. A **2.** D **3.** C **4.** C **5.** A **6.** C **7.** D **8.** B **9.** C **10.** C

Previous Years' Questions

1. (a) C (b) B **2.** C **3.** C **4.** A

Chapter 3

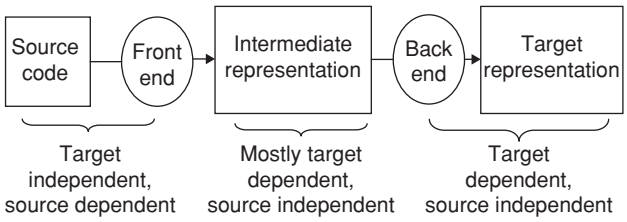
Intermediate Code Generation

LEARNING OBJECTIVES

- *Introduction*
- *Directed Acyclic Graphs (DAG)*
- *Three address code*
- *Symbol table operations*
- *Assignment statements*
- *Boolean expression*
- *Flow control of statements*
- *Procedure calls*
- *Code generation*
- *Next use information*
- *Run-time storage management*
- *DAG representations of basic blocks*
- *Peephole optimization*

Introduction

In the analysis–synthesis model, the front end translates a source program into an intermediate representation (IR). From IR the back end generates target code.



There are different types of intermediate representations:

- High level IR, i.e., AST (Abstract Syntax Tree)
- Medium level IR, i.e., Three address code
- Low level IR, i.e., DAG (Directed Acyclic Graph)
- Postfix Notation (Reverse Polish Notation, RPN).

In the previous sections already we have discussed about AST and RPN.

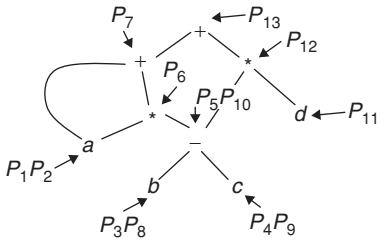
Benefits of Intermediate code generation: The benefits of ICG are

1. We can obtain an optimized code.
2. Compilers can be created for the different machines by attaching different backend to existing front end of each machine.
3. Compilers can be created for the different source languages.

Directed acyclic graphs for expression: (DAG)

- A DAG for an expression identifies the common sub expressions in the given expression.
- A node N in a DAG has more than one parent if N represents a common sub expression.
- DAG gives the compiler, important clues regarding the generation of efficient code to evaluate the expressions.

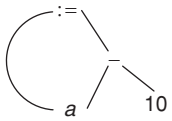
Example 1: DAG for $a + a*(b - c) + (b - c)*d$



P_1 = makeleaf (id, a)
P_2 = makeleaf (id, a) = P_1
P_3 = makeleaf (id, b)
P_4 = makeleaf (id, c)
P_5 = makenode (–, P_3, P_4)
P_6 = makenode (*, P_1, P_5)
P_7 = makenode (+, P_1, P_6)
P_8 = makeleaf (id, b) = P_3
P_9 = makeleaf (id, c) = P_4
P_{10} = makenode (–, P_8, P_9) = P_5

P_{11} = makeleaf (id, d)
P_{12} = makenode (*, P_{10}, P_{11})
P_{13} = makenode (+, P_7, P_{12})

Example 2: $a := a - 10$



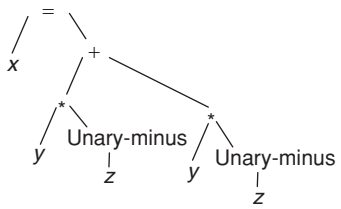
Three-Address Code

In three address codes, each statement usually contains 3 addresses, 2 for operands and 1 for the result.

Example: $-x = y$ OP z

- x, y, z are names, constants or complier generated temporaries,
- OP stands for any operator. Any arithmetic operator (or) Logical operator.

Example: Consider the statement $x = y * -z + y* -z$



The corresponding three address code will be like this:

Syntax Tree	DAG
$t_1 = -z$	$t_1 = -z$
$t_2 = y * t_1$	$t_2 = y * t_1$
$t_3 = -z$	$t_5 = t_2 + t_2$
$t_4 = y * t_3$	$X = t_5$
$t_5 = t_4 + t_2$	
$X = t_5$	

The postfix notation for syntax tree is: xyz unaryminus $*yz$ unaryminus $*+=$.

- Three address code is a 'Linearized representation' of syntax tree.
- Basic data of all variables can be formulated as syntax directed translation. Add attributes whenever necessary.

Example: Consider below *SDD* with following specifications:
E might have *E*. place and E.code
E.place: the name that holds the value of *E*.
E.code: the sequence of intermediate code starts evaluating *E*.
Let Newtemp: returns a new temporary variable each time it is called.
New label: returns a new label.
Then the SDD to produce three–address code for expressions is given below:

Production	Semantic Rules
$S \rightarrow$ id ASN E	S. code = E.code \\ gen (ASN, id.place, *E*.place) E. Place = newtemp ();
$E \rightarrow E_1$ PLUS E_2	E. code = E_1. code || E_2. code || gen (PLUS, *E*. place, E_1. place, E_2. place); E. place = newtemp();
$E \rightarrow E_1$MUL E_2	E. code = E_1. code || E_2.code || gen (MUL, E. place, E_1. place, E_2. place); E. Place = Newtemp();
$E \rightarrow$ UMINUS E_1	E. code = E_1 code || gen (NEG, *E*. Place, E_1. place); E. code = E_1.code
$E \rightarrow$ LP E_1 RP	E. Place = E_1. Place
$E \rightarrow$ IDENT	E.place = id. place E. code = empty.list ();

Types of Three Address Statement

Assignment

- Binary assignment: $x := y$ OP z Store the result of y OP z to x.
- Unary assignment: $x :=$ op y Store the result of unary operation on y to x.

Copy

- Simple Copy $x := y$ Store y to x
- Indexed Copy $x := y[i]$ Store the contents of $y[i]$ to x
- $x[i] := y$ Store y to $(x + i)$th address.

Address and pointer manipulation

$x := \&y$	Store address of y to x
$x := *y$	Store the contents of y to x
$*x := y$	Store y to location pointed by x .

Jump

- Unconditional jump:- goto L, jumps to L.
- Conditional:

```
if (x relop y)
goto L1;
else
```

```
{
goto L2;
}
```

Where relop is <, < =, >, > = , = or ≠.

Procedure call

Param x_1;

Param x_2;

.

.

.

Param x_n;

Call p, n, x;	Call procedure p with n parameters and store the result in x.
return x	Use x as result from procedure.

Declarations

- Global x, n_1, n_2: Declare a global variable named x at offset n_1 having n_2 bytes of space.
- Proc x, n_1, n_2: Declare a procedure x with n_1 bytes of parameter space and n_2 bytes of local variable space.
- Local x, m: Declare a local variable named x at offset m from the procedure frame.
- End: Declare the end of the current procedure.

Adaption for object oriented code

- $x = y$ field z: Lookup field named z within y, store address to x
- Class x, n_1, n_2: declare a class named x with n_1 bytes of class variables and n_2 bytes of class method pointers.
- Field x, n: Declare a field named x at offset n in the class frame.
- New x: Create a new instance of class name x.

Implementation of Three Address Statements

Three address statements can be implemented as records with fields for the operator and the operands. There are 3 types of representations:

1. Quadruples
2. Triples
3. Indirect triples

Quadruples

A quadruple has four fields: op, arg1, arg2 and result.

- Unary operators do not use arg2.
- Param use neither arg2 nor result.
- Jumps put the target label in result.
- The contents of the fields are pointers to the symbol table entries for the names represented by these fields.
- Easier to optimize and move code around.

Example 1: For the expression $x = y * - z + y * - z$, the quadruple representation is

	OP	Arg1	Arg2	Result
(0)	Uminus	z		t_1
(1)	*	y	t_1	t_2
(2)	Uminus	z		t_3
(3)	*	y	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Example 2: Read (x)

	Op	Arg1	Arg2	Result
(0)	Param	x		
(1)	Call	READ	(x)	

Example 3: WRITE ($A*B$, x +5)

	OP	Arg1	Arg2	Result
(0)	*	A	B	t_1
(1)	+	x	5	t_2
(2)	Param	t_1		
(3)	Param	t_2		
(4)	Call	Write	2	

Triples

Triples have three fields: OP, arg1, arg2.

- Temporaries are not used and instead references to instructions are made.
- Triples are also known as two address code.
- Triples takes less space when compared with Quadruples.
- Optimization by moving code around is difficult.
- The DAG and triple representations of expressions are equivalent.
- For the expression $a = y* - z + y*-z$ the Triple representation is

	Op	Arg1	Arg2
(0)	Uminus	z	
(1)	*	y	(0)
(2)	Uminus	z	
(3)	*	y	(2)
(4)	+	(1)	(3)
(5)	=	a	(4)

Array – references

Example: For A [I]: = B, the quadruple representation is

	Op	Arg1	Arg2	Result
(0)	[ ] =	A	I	T_1
(1)	=	B		T_2

The same can be represented by Triple representation also.

[] = is called L-value, specifies the address to an element.

	Op	Arg1	Arg2
(0)	[ ] =	A	I
(1)	=	(0)	B

Example 2: $A := B[I]$

	Op	Arg1	Arg2
(0)	= [ ]	B	I
(1)	=	A	(0)

= [ ] is called r-value, specifies the value of an element.

Indirect Triples

- In indirect triples, pointers to triples will be there instead of triples.
- Optimization by moving code around is easy.
- Indirect triples takes less space when compared with Quadruples.
- Both indirect triples and Quadruples are almost equally efficient.

Example: Indirect Triple representation of 3-address code

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

	Op	Arg1	Arg2
(14)	Uminus	z	
(15)	*	y	(14)
(16)	Uminus	z	
(17)	*	y	(16)
(18)	+	(15)	(17)
(19)	=	x	(18)

Symbol Table Operations

Treat symbol tables as objects.

- Mktable (previous);
  - create a new symbol table.
  - Link it to the symbol table previous.
- Enter (table, name, and type, offset)
  - insert a new identifier name with type and offset into table
  - Check for possible duplication.
- Add width (table, width);
  - increase the size of symbol table by width.
- Enterproc (table, name, new table)
  - Enter a procedure name into table.
  - The symbol table of name is new table.
- Lookup (name, table);
  - Check whether name is declared in the symbol table, if it is in the table then return the entry.

Example:

Declaration → $M_1 D$

$M_1 \rightarrow \in$ `{TOP (Offset): = 0 ;}`

$D \rightarrow D$ ID

$D \rightarrow$ id: T `{enter (top (tblptr), id.name, T.type top (offset)); top (offset): = top (offset) + T. width ;}`

$T \rightarrow$ integer `{T.type : = integer; T. width: = 4 :}`

$T \rightarrow$ double `{T.type: = double; T.width = 8 ;}`

$T \rightarrow * T_1$ `{T. type: = pointer (T. type); T.width = 4;}`

Need to remember the current offset before entering the block, and to restore it after the block is closed.

Example: Block → begin M4 Declarations statements end `{pop (tblptr); pop (offset) ;}`

$M_4 \rightarrow \in$ `{t: = mktable (top (tblptr); push (t, tblptr); push (top (offset), offset) ;`

Can also use the block number technique to avoid creating a new symbol table.

Field names in records

- A record declaration is treated as entering a block in terms of offset is concerned.
- Need to use a new symbol table.

Example: $T \rightarrow$ record M_5 D end

```
                    {T. type: = (top (tblptr));
                    T. width = top (offset);
                    pop (tblptr);
                    pop (offset) ;}
M5 → ∈  {t: = mktable (null);
        push (t, tblptr);
        push {(o, offset) ;}
```

Assignment Statements

Expressions can be of type integer, real, array and record. As part of translation of assignments into three address code, we show how names can be looked up in the symbol table and how elements of array can be accessed.

Code generation for assignment statements gen ([address # 1], [assignment], [address #2], operator, address # 3);

Variable accessing Depending on the type of [address # i], generate different codes.

Types of [address # i]:

- Local temp space
- Parameter
- Local variable
- Non-local variable
- Global variable
- Registers, constants,…

Error handling routine error – msg (error information);

The error messages can be written and stored in other file. Temp space management:

- This is used for generating code for expressions.
- newtemp (): allocates a temp space.
- freetemp (): free t if it is allocated in the temp space

Label management

- This is needed in generating branching statements.
- newlabel (): generate a label in the target code that has never been used.

Names in the symbol table

$S \rightarrow$ id: = E {p: = lookup (id-name, top (tblptr));
If p is not null then gen (p, ":=", E.place);
Else error ("var undefined", id. Name) ;}

$E \rightarrow E_1 + E_2$ {E. place = newtemp ();
gen (E.place, ": = ", E_1.place, "+", E_2.Place); free temp (E1.pace); freetemp
(E2. place) ;}

$E \rightarrow -E_1$ {E. place = newtemp ();
gen (E.place, ": =", "uminus", E_1.place);
Freetemp (E_1. place ;)}

$E \rightarrow (E_1)$ {E. place = E_1. place ;}

$E \rightarrow$ id {p: = lookup (id.name, top (tblptr);
If p ≠ null then E.place = p. place else error ("var undefined", id. name) ;}

Type conversions

Assume there are only two data types: integer, float.

For the expression,

$$E \rightarrow E_1 + E_2$$

If E_1. type = E_2. type then
generate no conversion code
E.type = E_1. type;

Else
E.type = float;
temp1 = newtemp ();
If E_1. type = integer then
gen (temp1,':=' int - to - float, E_1.place);
gen (E,':=' temp1, '+', E_2.place);

Else
gen (temp1,':=' int - to - float, E_2. place);
gen (E,':=' temp1, '+', E_1. place);
Free temp (temp1);

Addressing array elements

Let us assume
low: lower bound
w: element data width

Start_addr: starting address

1D Array: A[i]

- Start_addr + (i – low)* $w = i * w$ + (start_addr – low *w)
- The value called base, (start_addr – low * w) can be computed at compile time and then stored at the symbol table.

Example: array [−8 …100] of integer.
To declare [−8] [−7] … [100] integer array in Pascal.

2D Array A [i_1, i_2]

Row major order: row by row. A [i] means the ith row.

1st row A [1, 1]
A [1, 2]

2nd row A [2, 1]
A [2, 2]
A [i, j] = A [i] [j]

Column major: column by column.

A [1, 1] : A [1, 2]
A [2, 1] : A [2, 2]
1st Column 2nd column

Address for A [i_1, i_2]:
Start _ addr + ((i, – low$_1$) *n_2 + (i_2 – low$_2$))*w

Where low$_1$ and low$_2$ are the lower bounds of i_1 and i_2. n_2 is the number of values that i_2 can take. High$_2$ is the upper bound on the valve of i_2. n_2 = high$_2$ – low$_2$ + 1

We can rewrite address for A [i_1, i_2] as (($i_1 \times n_2$) + i_2) × w + (start _ addr - ((low$_1$ × n_2) + low$_2$) × w). The value (start _ addr – low$_1$ × n_2 × w – low$_2$ × w) can be computed at compile time and then stored in the symbol table.

Multi-Dimensional Array A [i_1, i_2,…i_k]

Address for A [i_1, i_2,…i_k]

$$= i_1 * \pi_{i=2}^{k} {}^{n_i} + i_2 * \pi_{i=3}^{k} {}^{n_i} + \cdots + i_k \Big) * w$$
$$+ \Big(start\_addr - low_1 * w * \pi_{i=2}^{k} ni$$
$$- low_2 * w * \pi_{i=3}^{k} {}^{n_i} - \cdots - low_k * w \Big)$$

It can be computed incrementally in grammar rules:
$f(1) = i_1$;
$f(j) = f(j-1) * n_j + i_j$;
$f(k)$ is the value we wanted to compute.
Attributes needed in the translation scheme for addressing array elements:

Elegize: size of each element in the array

Array: a pointer to the symbol table entry containing information about the array declaration.

Ndim: the current dimension index

Base: base address of this array

Place: where a variable is stored.

Limit (array, n) = n_m is the number of elements in the mth coordinate.

Translation scheme for array elements

Consider the grammar
$S \rightarrow L := E$
$E \rightarrow L$

$L \rightarrow$ id

$L \rightarrow$ [Elist]

Elist $\rightarrow$ Elist$_1$, E

Elist $\rightarrow$ id [E]

$E \rightarrow$ id

$E \rightarrow E + E$

$E \rightarrow (E)$

- $S \rightarrow L := E$ {if L. offset = null then /* L is a simple id */ gen (L. place, ":=", E.place);
  Else
  gen (L. place, "[", L. offset, "]",":=", E.place);
- $E \rightarrow E_1 + E_2$ {E.place = newtemp ();
  gen (E. place, ":=", E1.place, "+", E_2. place) ;}
- $E \rightarrow (E_1)$ {E.place= E_1.place}
- $E \rightarrow L$ {if L. offset = null then /* L is a simple id */ E.place:= L .place);
  Else begin
  E.place:=newtemp();
  gen (E.place, ":=",L.place, "[",L.offset, ']");
  end }
- $L \rightarrow$ id {P! = lookup (id.name, top (tblptr));
  If P ≠ null then
  Begin
  L.place: = P.place:
  L.offset:= null;
  End
  Else
  Error ("Var underfined", id. Name) ;}
- $L \rightarrow$ Elist {L. offset: = newtemp ();
  gen (L. offset, ":=", Elist.elesize, "*", Elist.place);
  freetemp (Elist.place);
  L.Place := Elist . base ;}
- Elist $\rightarrow$ Elist$_1$, E {t: =newtemp (); m: = Elist1. ndim+1;
  gen (t, ":=" Elist1.place, "*", limit (Elist1. array, m));
  Gen (t, ":=", t"+", E.place); freetemp (E.place);
  Elist.array: = Elist.array;
  Elist.place:= t; Elist.ndim:= m ;}
  Elist $\rightarrow$ id [E {Elist.Place:= E.place; Elist. ndim:=1;
  P! = lookup (id.name, top (tblptr)); check for id errors;
  Elist.elesize:= P.size; Elist.base: = p.base;
  Elist.array:= p.place ;}
- $E \rightarrow$ id {P:= lookup (id,name, top (tblptr);
  Check for id errors; E. Place: = Populace ;}

Boolean Expressions

There are two choices for implementation of Boolean expressions:

1. Numerical representation
2. Flow of control

Numerical representation

Encode true and false values.
Numerically, 1:true 0: false.
Flow of control: Representing the value of a Boolean expression by a position reached in a program.

Short circuit code: Generate the code to evaluate a Boolean expression in such a way that it is not necessary for the code to evaluate the entire expression.

- If a_1 or a_2
  a_1 is true then a_2 is not evaluated.
- If a_1 and a_2
  a_1 is false then a_2 is not evaluated.

Numerical representation

$E \rightarrow$ id$_1$ relop id2
{B.place:= newtemp ();
gen ("if", id1.place, relop.op, id2. place,"goto", next stat +3);
gen (B.place,":=", "0");
gen ("goto", nextstat+2);
gen (B.place,":=", "1")'}

Example 1: Translate the statement (if $a < b$ or $c < d$ and $e < f$) without short circuit evaluation.

```
100: if a < b goto 103
101: t1:= 0
102: goto 104
103: t1:= 1 /* true */
104: if c < d goto 107
105: t2:= 0 /* false */
106: goto 108
107: t2:= 1
108: if e < f goto 111
109: t3:= 0
110: goto 112
111: t3 := 1
112: t4 := t2 and t3
113: t3:= t1 or t4
```

Flow of Control Statements

$B \rightarrow$ id$_1$ relop id$_2$
{
B.true: = newlabel ();
B.false:= newlabel ();
B.code:= gen ("if", id$_1$. relop, id$_2$, "goto",

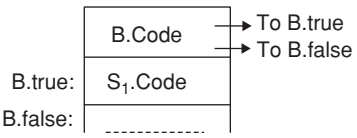
```
B.true, "else", "goto", B. false) ||
gen (B.true, ":")
}
```

$S \rightarrow$if B then S_1 S.code:= B.code || S_1.code ||gen (B.false, ':')

|| is the code concatenation operator.

1. **If – then implementation:**

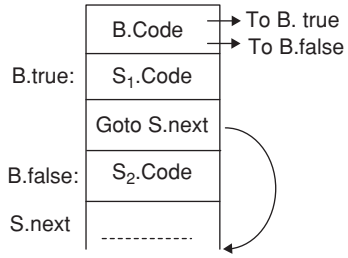
   $S \rightarrow$if B then $S1$ `{gen (Befalls," :");}`



2. **If – then – else**

   $P \rightarrow S$ `{S.next:= newlabel ();`
   `P.code:= S.code || gen (S.next," :")}`
   $S \rightarrow$ if B then S_1 else S_2 `{`S_1`.next:= S.next;`
   S_2`.next:= S.next;`
   `Secede: = B.code ||` S_1`.code ||.`
   `Gen ("goto" S.next) || B. false," :")`
   `||`S_2`.code}`

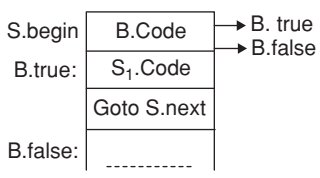
   Need to use inherited attributes of S to define the attributes of S_1 and S_2



3. **While loop:**

   $B \rightarrow$ id$_1$ relop id$_2$ B.true:= newlabel ();
   B.false:= newlabel ();
   B.code:=gen ('if', id.relop,
   id$_2$, 'goto', B.true 'else', 'goto', B. false) ||
   gen (B.true ':');

   $S \rightarrow$ while B do S_1 S.begin:= newlabel ();
   S.code:=gen (S.begin,':')||
   B.code||S1.code || gen
   ('goto', S.begin) || gen (B.false, ':');



4. **Switch/case statement:**

   The c - like syntax of switch case is
   switch epr {
   `case V [1]: S [1]`
   .
   .
   .
   `case V [k]: S[k]`
   `default: S[d]`
   }

Translation sequence

- Evaluate the expression.
- Find which value in the list matches the value of the expression, match default only if there is no match.
- Execute the statement associated with the matched value.

How to find the matched value? The matched value can be found in the following ways:

1. Sequential test
2. Lookup table
3. Hash table
4. Back patching

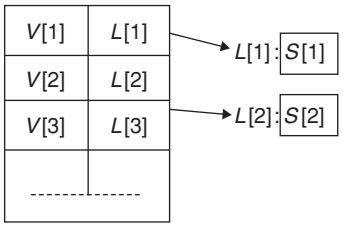
Two different translation schemes for sequential test are shown below:

1. Code to evaluate E into t
   Goto test
   $L[i]$: code for S [1]
   goto next

   $L[k]$: code for $S[k]$
   goto next
   $L[d]$: code for $S[d]$
   Go to next test:
   If $t = V$ [1]: goto L [1]
   .
   .
   .
   goto L[d]
   Next:
2. Can easily be converted into look up table
   If $t <> V[i]$ goto L [1]
   Code for S [1]
   goto next

   L [1]: if $t <> V$ [2] goto L [2]
   Code for S [2]
   Goto next
   L [k – 1]: if $t <> V$ [k] goto $L[k]$
   Code for $S[k]$
   Goto next
   .
   .
   .
   $L[k]$: code for $S[d]$
   Next:

Use a table and a loop to find the address to jump



3. **Hash table:** When there are more than two entries use a hash table to find the correct table entry.
4. **Back patching:**
   - Generate a series of branching statements with the targets of jumps temporarily left unspecified.
   - To determine label table: each entry contains a list of places that need to be back patched.
   - Can also be used to implement labels and gotos.

Procedure Calls

- Space must be allocated for the activation record of the called procedure.
- Arguments are evaluated and made available to the called procedure in a known place.
- Save current machine status.
- When a procedure returns:
  - Place returns value in a known place.
  - Restore activation record.

Example: $S \rightarrow$ call id (Elist)

```
{for each item P on the queue Elist.
Queue do gen ('PARAM', q);
gen ('call:', id.place) ;}
```

Elist $\rightarrow$ Elist, E {append E.place to the end of Elist.queue}

Elist $\rightarrow$ E {initialize Elist.queue to contain only E.place}

Use a queue to hold parameters, then generate codes for params.

Code for E_1, store in t_1

.
.
.

Code for E_k, store in t_k

PARAM $t1$

:
.
.
.

PARAM tk

Call P

Terminology:

Procedure declaration:

Parameters, formal parameters

Procedure call:

Arguments, actual parameters.

The values of a variable: $x = y$

r – value: value of the variable, i.e., on the right side of assignment. Ex: y, in above assignment.

l – value: The location/address of the variable, i.e., on the leftside of assignment. Ex: x, in above assignment.

There are different modes of parameter passing

1. call-by-value
2. call-by-reference
3. call-by-value-result (copy-restore)
4. call-by-name

Call by value

Calling procedure copies the r values of the arguments into the called proceduce's Activation Record.

Changing a formal parameter has no effect on the actual parameter.

Example: void add (int C)

```
{
C = C+ 10;
printf ('\nc = %d', &C);
}
main ()
{
int a = 5;
printf ('a=%d', &a);
add (a);
printf ('\na = %d', &a);
}
```

In main a will not be affected by calling add (a)

It prints $a = 5$

$a = 5$

Only the value of C in add ( ) will be changed to 15.

Usage:

1. Used by PASCAL and C++ if we use non-var parameters.
2. The only thing used in C.

Advantages:

1. No aliasing.
2. Easier for static optimization analysis.
3. Faster execution because of no need for redirecting.

Call by reference

Calling procedure copies the l-values of the arguments into the called procedure's activation record. i.e., address will be passed to the called procedure.

- Changing formal parameter affects the corresponding actual parameter.
- It will have some side effects.

Example:

```
void add (int *c)
{
*c = *c + 10;
printf('\nc=%d', *c);
```

```
}
void main()
{
int a = 5;
printf ('\na = %d', a);
add (&a);
printf ('\na = %d', a);
```

output: a = 5
c = 15
a = 15

That is, here the actual parameter is also modified.

Advantages

1. Efficiency in passing large objects.
2. Only need to copy addresses.

Call-by-value-result

Equivalent to call-by-reference except when there is aliasing. That is, the program produces the same result, but not the same code will be generated.

Aliasing: Two expressions that have the same l-values are called aliases. They access the same location from different places.

Aliasing happens through pointer manipulation.

1. Call by reference with global variable as an argument.
2. Call by reference with the same expression as argument twice.

Example: test (*x*,*y*,*x*)

Advantages:

1. If there is no aliasing, we can implement it by using call – by – reference for large objects.
2. No implicit side effect if pointers are not passed.

Call by-name

used in Algol.

- Procedure body is substituted for the call in calling procedure.
- Each occurrence of a parameter in the called procedure is replaced with the corresponding argument.
- Similar to macro expansion.
- A parameter is not evaluated unless its value is needed during computation.

Example:

```
void show (int x)
{
for (int y = 0; y < 10; y++)
x++;
}
main ()
{
int j;
j = -1;
show (j);
}
Actually it will be like this
main ()
{
int j;
j = - 1;
For (in y= 0; y < 10; y ++)
x ++;
}
```

- Instead of passing values or address as arguments, a function is passed for each argument.
- These functions are called **thunks.**
- Each time a parameter is used, the thunk is called, then the address returned by the thunk is used.

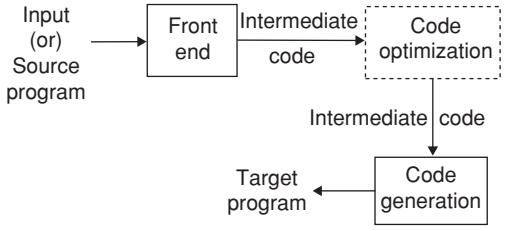
$y = 0$: use return value of thunk for y as the ℓ-value.

Advantages

- More efficient when passing parameters that are never used.
- This saves lot of time because evaluating unused parameter takes a longtime.

Code Generation

Code generation is the final phase of the compiler model.



The requirements imposed on a code generator are

1. Output code must be correct.
2. Output code must be of high quality.
3. Code generator should run efficiently.

Issues in the Design of a Code Generator

The generic issues in the design of code generators are

- Input to the code generator
- Target programs
- Memory Management
- Instruction selection
- Register Allocation
- Choice of Evaluation order

Input to the code generator

Intermediate representation with symbol table will be the input for the code generator.

- High Level Intermediate representation

Example: Abstract Syntax Tree (AST)

- Medium – level intermediate representation

Example: control flow graph of complex operations

- Low – Level Intermediate representation

Example: Quadruples, DAGS

- Code for abstract stack machine, i.e., postfix code.

Target programs

The output of the code generator is the target program. The output may take on a variety of forms:

1. Absolute machine language
2. Relocatable machine language
3. Assembly language

Absolute machine language

- Final memory area for a program is statically known.
- Hard coded addresses.
- Sufficient for very simple systems.

Advantages:

- Fast for small programs
- No separate compilation

Disadvantages: Can not call modules from other languages/ compliers.

Relocatable code It Needs

- Relocation table
- Relocating linker + loader (or) runtime relocation in Memory management Unit (MMU).

Advantage: More flexible.

Assembly language Generates assembly code and use an assembler tool to convert this to binary (object) code. It needs (i) assembler (ii) linker and loader.

Advantage: Easier to handle and closer to machine.

Memory management

Mapping names in the source program to addresses of data objects in runtime memory is done by the front end and the code generator.

- A name in a three address statement refers to a symbol entry for the name.
- Stack, heap, garbage collection is done here.

Instruction selection

Instruction selection depends on the factors like

- Uniformity
- Completeness of the instruction
- Instruction speed
- Machine idioms

- Choose set of instructions equivalent to intermediate representation code.
- Minimize execution time, used registers and code size.

Example: $x = y + z$ in three address statements:

MOV y, R_0 / * load y into R_0 * /
ADD z, R_0
MOV R_0, x /* store R_0 into x*/

Register allocation

- Instructions with register operands are faster. So, keep frequently used values in registers.
- Some registers are reserved.

Example: *SP*, *PC* … etc.
Minimize number of loads and stores.

Evaluation order

- The order of evaluation can affect the efficiency of the target code.
- Some orders require fewer registers to hold intermediate results.

Target Machine

Lets us assume, the target computer is

- Byte addressable with 4 bytes per word
- It has n general purpose registers
  $R_0, R_1, R_2, \ldots R_{n-1}$
- It has 2 address instructions of the form

  OP source, destination
  [cost: 1 + added]

Example: The op may be MOV, ADD, MUL.
Generally cost will be like this

Source	Destination	Cost
Register	Register	1
Register	Memory	2
Memory	Register	2
Memory	Memory	3

Addressing modes:

Mode	Form	Address	Cost
Absolute	M	M	2
Register	R	R	1
Indexed	C(R)	C+contents(R)	2
Indirect register	*R	Contents (R)	1
Indirect indexed	*C(R)	Contents (C+contents (R))	2

Example: $x := y - z$

MOV y, R0 → cost = 2
SUB z, R0 → cost = 2
MOV R_0, x → cost = 2

Runtime Storage Management

Storage Organization

To run a compiled program, compiler will demand the operating system for the block of memory. This block of memory is called runtime storage.

This run time storage is subdivided into the generated target code, Data objects and Information which keeps track of procedure activations.

The fixed data (generated code) is stored at the statically determined area of the memory. The Target code is placed at the lower end of the memory.

The data objects are stored at the statically determined area as its size is known at the compile time. Compiler stores these data objects at statically determined area because these are compiled into target code. This static data area is placed on the top of the code area.

The runtime storage contains stack and the heap. Stack contains activation records and program counter, data object within this activation record are also stored in this stack with relevant information.

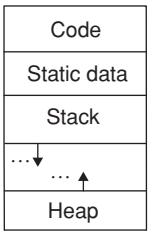
The heap area allocates the memory for the dynamic data (for example some data items are allocated under the program control)

The size of stack and heap will grow or shrink according to the program execution.

Activation Record

Information needed during an execution of a procedure is kept in a block of storage called an activation record.

- Storage for names local to the procedures appears in the activation record.
- Each execution of a procedure is referred as activation of the procedure.
- If the procedure is recursive, several of its activation might be alive at a given time.
- Runtime storage is subdivided into
  1. Generated target code area
  2. Data objects area
  3. Stack
  4. Heap



- Sizes of stack and heap can change during program execution.

For code generation there are two standard storage allocations:

1. **Static allocation:** The position of an activation record in memory is fixed at compile time.
2. **Stack allocation:** A new activation record is pushed on to the stack for each execution of the procedure.

The record is poped when the activation ends.

Control stack The control stack is used for managing active procedures, which means when a call occurs, the execution of activation is interrupted and status information of the stack is saved on the stack.

When control is returned from a call, the suspended activation is resumed after storing the values of relevant registers it also includes program counter which sets to point immediately after the call.

The size of stack is not fixed.

Scope of declarations Declaration scope refers to the certain program text portion, in which rules are defined by the language.

Within the defined scope, entity can access legally to declared entities.

The scope of declaration contains immediate scope always. Immediate scope is a region of declarative portion with enclosure of declaration immediately.

Scope starts at the beginning of declaration and scope continues till the end of declaration. Whereas in the over loadable declaration, the immediate scope will begin, when the callable entity profile was determined.

The visible part refers text portion of declaration, which is visible from outside.

Flow Graph

A flow graph is a graph representation of three address statement sequences.

- Useful for code generation algorithms.
- Nodes in the flow graph represents computations.
- Edges represent flow of control.

Basic Blocks

Basic blocks are sequences of consecutive statements in which flow of control enters at the beginning and leaves at the end without a halt or branching.

1. First determine the set of leaders
   - First statement is leader
   - Any target of goto is a leader
   - Any statement that follows a goto is a leader.
2. For each leader its basic block consists of the leader and all statements up to next leader.

Initial node: Block with first statement is leader.

Example: consider the following fragment of code that computes dot product of two vectors x and y of length 10.

```
begin
Prod: = 0;
```

```
i: = 1;
repeat
begin
Prod: = Prod + x [i] * y [i];
i: = i + 1;
end
until i < = 10;
end
```

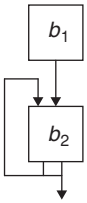
B_1

(1)	Prod : = 0
(2)	I: = 1

B_2

(3)	t_1:= 4*i
(4)	t_2: =x[t_1]
(5)	t_3: =4 * i
(6)	t_4: =y [t_3]
(7)	t_5: =t_2* t_4
(8)	t_6; =Prod + t_5
(9)	Prod := t_6
(10)	t_7: = i+1
(11)	i:= t_7
(12)	if i < = 10 goto (3)

∴The flow graph for this code will be



Here b_1 is the initial node/block.

- Once the basic blocks have been defined, a number of transformations can be applied to them to improve the quality of code.
  1. **Global:** Data flow analysis
  2. **Local:**
     - Structure preserving transformations
     - Algebraic transformations
- Basic blocks compute a set of expressions. These expressions are the values of the names live on exit from the block.
- Two basic blocks are equivalent if they compute the same set of expressions.

Structure preserving transformations:

1. *Common sub-expression elimination:*

$$\begin{array}{l} a := b + c \\ b := a - d \\ c := b + c \\ d := a - d \end{array} \Rightarrow \begin{array}{l} a := b + c \\ b := a - d \\ c := b + c \\ d := b \end{array}$$

2. *Dead code elimination:* Code that computes values for names that will be dead i.e., never subsequently used can be removed.
3. *Renaming of temporary variables*
4. *Interchange of two independent adjacent statements*

Algebraic Transformations

Algebraic identities represent other important class optimizations on basic blocks. For example, we may apply arithmetic identities, such as $x + 0 = 0 + x = x$,

$$x * 1 = 1 * x = x$$
$$x - 0 = x$$
$$x/1 = x$$

Next-Use Information

- Next-use info used in code generation and register allocation.
- Remove variables from registers if not used.
- Statement of the form $A = B$ or C defines A and uses B and C.
- Scan each basic block backwards.
- Assume all temporaries are dead or exit and all user variables are live or exit.

Algorithm to compute next use information

Suppose we are scanning

i: x: = y op z

in backward scan

- attach to i, information in symbol table about x, y, z.
- set x to not live and no next-use in symbol table
- set y and z to be live and next-use in symbol table.

Consider the following code:

1: $t_1 = a * a$

2: $t_2 = a * b$

3: $t_3 = 2 * t_2$

4: $t_4 = t_1 + t_2$

5: $t_5 = b * b$

6: $t_6 = t_4 + t_5$

7: $x = t_6$

Statements:

7: no temporary is live

6: t_6: use (7) t_4 t_5 not live

5: t_5: use (6)

4: t_4: use (6), t_1 t_3 not live

3: t_3: use (4) t_2 not live

2: t_2: use (3)

1: t_1: use (4)

Symbol Table:

t_1 dead use in 4

t_2 dead use in 3
t_3 dead use in 4
t_4 dead use in 6
t_5 dead use in 6
t_6 dead use in 7

The six temporaries in the basic block can be packed into two locations t_1 and t_2:

1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_2 = 2 * t_2$
4: $t_1 = t_1 + t_2$
5: $t_2 = b * b$
6: $t_1 = t_1 + t_2$
7: $x = t_1$

Code Generator

- Consider each statement
- Remember if operand is in a register
- Descriptors are used to keep track of register contents and address for names
- There are 2 types of descriptors
  1. Register Descriptor
  2. Address Descriptor

Register Descriptor

Keep track of what is currently in each register. Initially all registers are empty.

Address Descriptors

- Keep track of location where current value of the name can be found at runtime.
- The location might be a register, stack, memory address or a set of all these.

Issues in design of code generation The issues in the design of code generation are

1. Intermediate representation
2. Target code
3. Address mapping
4. Instruction set.

Intermediate Representation It is represented in post fix, 3-address code (or) quadruples and syntax tree (or) DAG.

Target Code The Target Code could be absolute code, relocatable machine code (or) assembly language code. Absolute code will execute immediately as it is having fixed address relocatable, requires linker and loader to get the code from appropriate location for the assembly code, assemblers are required to convert it into machine level code before execution.

Address mapping In this, mapping is defined between intermediate representations to target code address.

It is based on run time environment like static, stack or heap.

Instruction set It should provide a complete set in such a way that all its operations can be implemented.

Code Generation Algorithm

For each three address statement $x = y$ op z do

- Invoke a function getreg to determine location L where x must be stored. Usually L is a register.
- Consult address descriptor of y to determine y'. Prefer a register for y'. If value of y is not already in L generate MOV y', L.
- Generate
  OP z', L
  Again prefer a register for z. Update address descriptor of x to indicate x is in L. If L is a register update its descriptor to indicate that it contains x and remove x from all other register descriptors.
- If current value of y and/or z have no next use and are dead or exit from block and are in registers then change the register descriptor to indicate that it no longer contain y and /or z.

Function getreg

1. If y is in register and y is not live and has no next use after $x = y$ OP z then return register of y for L.
2. Failing (1) return an empty register.
3. Failing (2) if x has a next use in the block or OP requires register then get a register R, store its contents into M and use it.
4. Else select memory location x as L.

Example: $D: = (a - b) + (a - c) + (a - c)$

Stmt	Code Generated	reg desc	addr desc
$t = a - b$	MOV a, R_0 SUB b, R_0	R_0 contains t	t in R_0
$u = a - c$	MOV a, R_1 SUB c, R_1	R_0 contains t R_1 contains u	t in R_0 u in R_1
$v = t + u$	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_0 v in R_0
$d = v + u$	ADD R_1, R_0 MOV R_0,d	R_o contains d	d in R_0 d in R_0 and memory

Conditional Statements

Machines implement conditional jumps in 2 ways:

1. Based on the value of the designated register (R) Branch if values of R meets one of six conditions.
   (i) Negative (ii) Zero
   (iii) Positive (iv) Non-negative
   (v) Non-zero (vi) Non-positive

Example: Three address statement: if $x < y$ goto z
It can be implemented by subtracting y from x in R, then jump to z if value of R is negative.

2. Based on a set of **condition codes** to indicate whether last quantity computed or loaded into a location is negative (or) Zero (or) Positive.
   - compare instruction set codes without actually computing the value.

Example: CMP x, y
CJL Z.

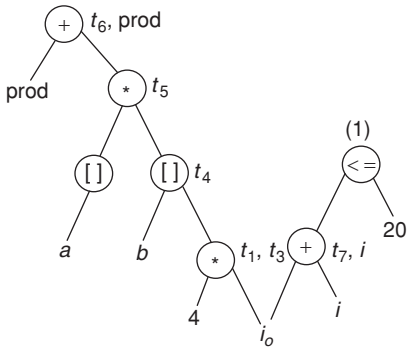
- Maintains a condition code descriptor, which tells the name that last sets the condition codes.

Example: $X := y + z$
If $x < 0$ goto z
By
MOV y, R_o
ADD z, R_o
MOV R_o, x
CJN z.

DAG Representation of Basic Blocks

- DAGS are useful data structures for implementing transformations on basic blocks.
- Tells, how value computed by a statement is used in subsequent statements.
- It is a good way of determining common sub expressions.
- A DAG for a basic block has following labels on the 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 as a sequence of identifiers for labels.

Example: 1: $t_1 := 4 * i$
2: $t_2 := a\ [t_1]$
3: $t_3 := 4 * i$
4: $t_4 := b\ [t_3]$
5: $t_5 := t_2 * t_4$
6: $t_6 := \text{prod} + t_5$
7: prod: $= t_6$
8: $t_7 := i + 1$
9: $i = t_7$
10: if $i <= 20$ got (1)

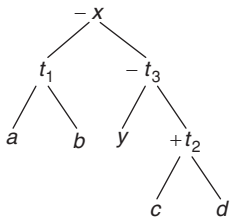


Code Generation from DAG:

$S_1 = 4 * i$	$S_1 = 4 * i$
$S_2 = \text{add(A)} - 4$	$S_2 = \text{add(A)} - 4$
$S_3 = S_2\ [S_1]$	$S_3 = S_2\ [S_1]$
$S_4 = 4 * i$	
$S_5 = \text{add(B)} - 4$	$S_5 = \text{add(B)} - 4$
$S_6 = S_5[S_4]$	$S_6 = S_5[S_4]$
$S_7 = S_3 * S_6$	$S_7 = S_3 * S_6$
$S_8 = \text{prod} + S_7$	$\text{prod} = \text{prod} + S_7$
$\text{prod} = S_8$	
$S_9 = I + 1$	
$I = S_9$	$I = I + 1$
if $I <= 20$ got (1)	if $I <= 20$ got (1)

Rearranging order of the code

Consider the following basic block
$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$x = t_1 - t_3$ and its DAG



Three address code for the DAG:
(Assuming only two registers are available)
MOV a, R_o
ADD b, R_o
MOV c, R_1
MOV R_o, t_1 Register Spilling
MOV e, R_o Register Reloading
SUB R_1, R_o
MOV t_1, R_1
SUB R_o, R_1
MOV R_1, x

Rearranging the code as
$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$x = t_1 - t_3$
The rearrangement gives the code:
MOV c, R_o
ADD d, R_o
MOV e, R_1
SUB R_o, R_1

MOV a, R_o
ADD b, R_o
SUB R_1, R_0
MOV R_1, x

Error detection and Recovery The errors that arise while compiling

1. Lexical errors
2. Syntactic errors
3. Semantic errors
4. Run-time errors

Lexical errors If the variable (or) constants are declared (or) defined, not according to the rules of language, special symbols are included which were not part of the language, etc is the lexical error.

Lexical analyzer is constructed based on pattern recognizing rules to form a token, when a source code is made into tokens and if these tokens are not according to rules then errors are generated.

Consider a c program statement

printf ('Hello World');

Main printf, (, ', Hello world,' ,),; are tokens.

Printf is not recognizable pattern, actually it should be printf. It generates an error.

Syntactic error These errors include semi colons, missing braces etc. which are according to language rules.

The parser reports the errors

Semantic errors This type of errors arises, when operation is performed over incompatible type of variables, double declaration, assigning values to undefined variables etc.

Runtime errors The Runtime errors are the one which are detected at runtime. These include pointers assigned with NULL values and accessing a variable which is out of its boundary, unlegible arithmetic operations etc.

After the detection of errors. The following recovery strategies should be implemented.

1. Panic mode recovery
2. Phrase level recovery
3. Error production
4. Global correction.

Peephole Optimization

- Target code often contains redundant instructions and suboptimal constructs.
- Improving the performance of the target program by examining a short sequence of target instructions (peephole) and replacing these instructions by a shorter or faster sequence is peephole optimization.
- The peephole is a small, moving window on the target program. Some well known peephole optimizations are

1. Eliminating redundant instructions
2. Eliminating unreachable code
3. Flow of control optimizations or Eliminating jumps over jumps
4. Algebraic simplifications
5. Strength reduction
6. Use of machine idioms

Elimination of Redundant Loads and stores

Example 1: (1) MOV R_o, a
(2) MOV a, R_o

We can delete instruction (2), because the value of a is already in R_0.

Example 2: Load x, R_0
Store R_0, x

If no modifications to R_0/x then store instruction can be deleted

Example 3: (1) Load x, R_0
(2) Store R_0, x

Example 4: (1) store R_0, x
(2) Load x, R_0

Second instruction can be deleted from both examples 3 and 4.

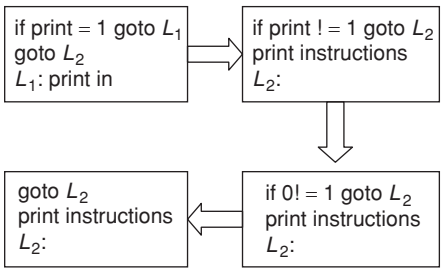
Example 5: Store R_0, x
Load x, R_0
Here load instruction can be deleted.

Eliminating Unreachable code

An unlabeled instruction immediately following and unconditional jump may be removed.

- May be produced due to debugging code introduced during development.
- May be due to updates in programs without considering the whole program segment.

Example: Let print = 0



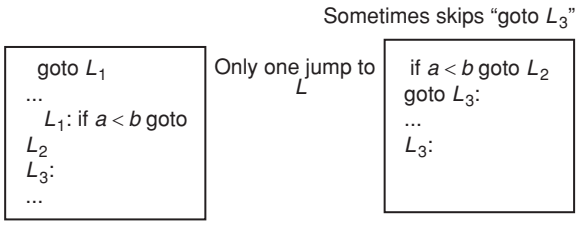
In all of the above cases print instructions are unreachable.
∴ Print instructions can be eliminated.

Example: goto L_2
…
L_2:

Flow of control optimizations The unnecessary jumps can be eliminated.
Jumps like:
Jumps to jumps,
Jumps to conditional jumps,
Conditional jumps to jumps.

Example 1: we can replace the jump sequence
goto L_1
…
L_1: got L_2
By the sequence
Got L_2
L_1: got L_2,
…
If there are no jumps to L_1 then it may be possible to eliminate the statement L_1: goto L_2.

Example 2:



Reduction in strength

- x^2 is cheaper to implement as $x * x$ than as a call to exponentiation routine.
- Replacement of multiplication by left shift.

Example: $x * 2^3 \Rightarrow x << 3$

- Replace division by right shift.

Example: $x >> 2$ (is $x/2^2$)

Use of machine Idioms

- Auto increment and auto decrement addressing modes can be used whenever possible.

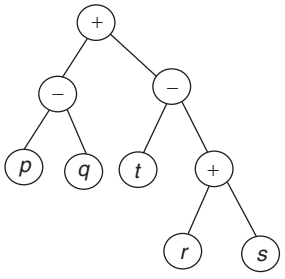
Example: replace add #1, R by INC R

EXERCISES

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices

1. Consider the following expression tree on a machine with bad store architecture in which memory can be accessed only through load and store instructions. The variables p, q, r, s and t are initially stored in memory. The binary operators used in this expression tree can be evaluated by the machine only when the operands are in registers. The instructions produce result only in a register if no intermediate results can be stored in memory, what is the minimum number of registers needed to evaluate this expression?



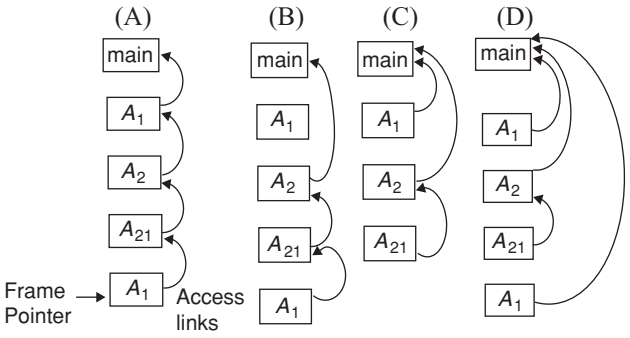
(A) 2 (B) 9
(C) 5 (D) 3

2. Consider the program given below with lexical scoping and nesting of procedures permitted.

```
Program main ( )
{
Var …
Procedure A1 ( )
{
Var …
call A2;
}
Procedure A2 ( )
{
Var..
Procedure A21 ( )
{
Var…
call A21 ( );
}
Call A1;
}
Call A1;
}
```

Consider the calling chain: main ( ) → A_1 ( ) → A_2 ( ) → A_{21} ( ) → A_1 ( ).

The correct set of activation records along with their access links is given by



3. Consider the program fragment:

```
sum = 0;
For (i = 1; i < = 20; i++)
sum = sum + a[i] +b[i];
```

How many instructions are there in the three-address code for this?

(A) 15 (B) 16
(C) 17 (D) 18

4. Suppose the instruction set of the processor has only two registers. The code optimization allowed is code motion. What is the minimum number of spills to memory in the complied code?

```
c = a + b;
d = c*a;
e = c + a;
x = c*c;
If (x > a)
{
y = a*a;
Else
{
d = d*d; e = e*e;
}
```

(A) 0 (B) 1
(C) 2 (D) 3

5. What is the minimum number of registers needed to compile the above problem's code segment without any spill to memory?

(A) 3 (B) 4
(C) 5 (D) 6

6. Convert the following expression into postfix notation:

$a = (-a + 2*b)/a$

(A) $aa - 2b\ *+a/=$ (B) $a - 2ba\ */+=$
(C) $a2b\ *\ a/+$ (D) $a2b - *\ a/+$

7. In the quadruple representation of the following program, how many temporaries are used?

int $a = 2$, $b = 8$, $c = 4$, d;

For ($j = 0$; j< = 10; j++)

$a = a\ *\ (j*\ (b/c))$;

$d = a\ *\ (j*\ (b/c))$;

(A) 4 (B) 7
(C) 8 (D) 10

8. Let $A = 2$, $B = 3$, $C = 4$ and $D = 5$, what is the final value of the prefix expression: $+\ *\ AB - CD$

(A) 5 (B) 10
(C) –10 (D) –5

9. Which of the following is a valid expression?

(A) $BC\ *\ D - +$ (B) $*\ ABC -$
(C) $BBB\ ***- +$ (D) $-*/bc$

10. What is the final value of the postfix expression $B\ C\ D\ A\ D - + - +$ where $A = 2$, $B = 3$, $C = 4$, $D = 5$?

(A) 5 (B) 4
(C) 6 (D) 7

11. Consider the expression $x = (a + b)*\ -C/D$. In the quadruple representation of this expression in which instruction '/' operation is used?

(A) 3rd (B) 4th
(C) 5th (D) 8th

12. In the triple representation of $x = (a + b)*- c/d$, in which instruction $(a + b)\ *\ - c/d$ result will be assigned to x?

(A) 3rd (B) 4th
(C) 5th (D) 8th

13. Consider the three address code for the following program:

While ($A < C$ and $B > D$) do

If ($A = = 1$) then $C = C + 1$;

Else

While ($A < = D$) do

$A = A + 3$;

How many temporaries are used?

(A) 2 (B) 3
(C) 4 (D) 0

14. Code generation can be done by

(A) DAG (B) Labeled tree
(C) Both (A) and (B) (D) None of these

15. Live variables analysis is used as a technique for

(A) Code generation (B) Code optimization
(C) Type checking (D) Run time management

Practice Problems 2

Directions for questions 1 to 19: Select the correct alternative from the given choices

1. Match the correct code optimization technique to the corresponding code:

(i) $i = i\ *\ 1$ $j = 2\ *\ i$	$\Rightarrow\ j = 2\ *\ i$	(p) Reduction in strength
(ii) $A = B + C$ $D = 10 + B + C$	$\Rightarrow\ A = B + C$ $D = 10 + A$	(q) Machine Idioms
(iii) For $i = 1$ to 10 $A[i] = B + C$	$\Rightarrow$ for $i = 1$ to 10 $t = B + C$ $A[i] = t$;	(r) Common sub expression elimination.
(iv) $x = 2\ *\ y \Rightarrow$	$y << 2$;	(s) Code motion

(A) i – r, iii – s, iv – p, ii – q
(B) i – q, ii – r, iii – s, iv –p
(C) i – s, iii – p, iii – q, iv – r
(D) i – q, ii – p, iii – r, iv – s

2. What will be the optimized code for the following expression represented in DAG?

$a = q * -r + q * -r$

(A) $t_1 = -r$
$t_2 = q * t_1$
$t_3 = a * t_1$
$t_4 = t_2 + t_3$
$a = t_4$

(B) $t_1 = -r$
$t_2 = q * t_1$
$t_3 = t_2 + t_2$
$a = t_3$

(C) $t_1 = -r$
$t_2 = q$
$t_3 = t_1 * t_2$
$t_4 = t_3 + t_3$
$a = t_4$

(D) All of these

3. In static allocation, names are bound to storage at _____ time.
(A) Compile
(B) Runtime
(C) Debugging
(D) Both (A) and (B)

4. The actual parameters are evaluate d and their r-values are passed to the called procedure is known as
(A) call-by-reference
(B) call-by-name
(C) call-by-value
(D) copy-restore

5. If the expression $-(a + b) *(c + d) + (a + b + c)$ is translated into quadruple representation, then how many temporaries are required?
(A) 5
(B) 6
(C) 7
(D) 8

6. If the above expression is translated into triples representation, then how many instructions are there?
(A) 6
(B) 10
(C) 5
(D) 8

7. In the indirect triple representation for the expression $A = (E/F) * (C - D)$. The first pointer address refers to
(A) $C - D$
(B) E/F
(C) Both (A) and (B)
(D) $(E/F) * (C - D)$

8. For the given assembly language, what is the cost for it?

MOV b, a

ADD c, a

(A) 3
(B) 4
(C) 6
(D) 2

9. Consider the expression

((4 + 2 * 3 + 7) + 8 * 5). The polish postfix notation for this expression is
(A) 423* + 7 + 85*+
(B) 423* + 7 + 8 + 5*
(C) 42 + 37 + *85* +
(D) 42 + 37 + 85** +

Common data for questions 10 to 15: Consider the following basic block, in which all variables are integers, and ** denotes exponentiation.

$a: = b + c$
$z: = a * * 2$
$x: = 0 * b$
$y: = b + c$
$w: = y * y$
$u: = x + 3$
$v: = u + w$

Assume that the only variables that are live at the exit of this block are v and z. In order, apply the following optimization to this basic block.

10. After applying algebraic simplification, how many instructions will be modified?
(A) 1
(B) 2
(C) 4
(D) 5

11. After applying common sub expression elimination to the above code. Which of the following are true?
(A) $a: = b + c$
(B) $y: = a$
(C) $z = a + a$
(D) None of these

12. Among the following instructions, which will be modified after applying copy propagation?
(A) $a: = b + c$
(B) $z: = a * a$
(C) $y: = a$
(D) $w: = y * y$

13. Which of the following is obtained after constant folding?
(A) $u: = 3$
(B) $v: = u + w$
(C) $x: = 0$
(D) Both (A) and (C)

14. In order to apply dead code elimination, what are the statements to be eliminated?
(A) $x = 0$
(B) $y = b + c$
(C) Both (A) and (B)
(D) None of these

15. How many instructions will be there after optimizing the above result further?
(A) 1
(B) 2
(C) 3
(D) 4

16. Consider the following program:

L_0: $e: = 0$
$b: = 1$
$d: = 2$
L_1: $a: = b + 2$
$c: = d + 5$
$e: = e + c$
$f: a*a$
If $f < c$ goto L_3
L_2: $e: = e + f$
goto L_4
L_3: $e: = e + 2$
L_4: $d: = d + 4$
$b: = b - 4$
If $b! = d$ goto 4
L_5:

How many blocks are there in the flow graph for the above code?

(A) 5
(B) 6
(C) 8
(D) 7

17. A basic block can be analyzed by
(A) Flow graph
(B) A graph with cycles
(C) DAG
(D) None of these

18. In call by value the actual parameters are evaluated. What type of values is passed to the called procedure?
(A) l-values
(B) r-values
(C) Text of actual parameters
(D) None of these

19. Which of the following is FALSE regarding a Block?
(A) The first statement is a leader.
(B) Any statement that is a target of conditional / unconditional goto is a leader.
(C) Immediately next statement of goto is a leader.
(D) The last statement is a leader.

Previous Years' Questions

1. The least number of temporary variables required to create a three-address code in static single assignment form for the expression $q + r/3 + s - t * 5 + u * v/w$ is _____ **[2015]**

2. Consider the intermediate code given below.

(1) $i = 1$
(2) $j = 1$
(3) $t_1 = 5 * i$
(4) $t_2 = t_1 + j$
(5) $t_3 = 4 * t_2$
(6) $t_4 = t_3$
(7) $a[t_4] = -1$
(8) $j = j + 1$
(9) if $j <= 5$ goto (3)
(10) $i = i + 1$
(11) if $i < 5$ goto (2)

The number of nodes and edges in the control-flow-graph constructed for the above code, respectively, are **[2015]**

(A) 5 and 7 (B) 6 and 7
(C) 5 and 5 (D) 7 and 8

3. Consider the following code segment. **[2016]**

$x = u - t;$
$y = x * v;$
$x = y + w;$
$y = t - z;$
$y = x * y;$

The minimum number of total variables required to convert the above code segment *to static single assignment form is* _____ .

4. What will be the output of the following pseudo-code when parameters are passed by reference and dynamic scoping is assumed? **[2016]**

$a = 3;$
void $n(x)$ { $x = x * a$; print (x);}
void $m(y)$ {$a = 1$; $a = y - a$; $n(a)$; print (a)}
void main( ) {$m(a)$;}

(A) 6,2 (B) 6,6
(C) 4,2 (D) 4,4

5. Consider the following intermediate program in three address code

$$p = a - b$$
$$q = p * c$$
$$p = u * v$$
$$q = p + q$$

Which one of the following corresponds to a *static single assignment* form of the above code? **[2017]**

(A) $p_1 = a - b$
$q_1 = p_1 * c$
$p_1 = u * v$
$q_1 = p_1 + q_1$

(B) $p_3 = a - b$
$q_4 = p_3 * c$
$p_4 = u * v$
$q_5 = p_4 + q_4$

(C) $p_1 = a - b$
$q_1 = p_2 * c$
$p_3 = u * v$
$q_2 = p_4 + q_3$

(D) $p_1 = a - b$
$q_1 = p * c$
$p_2 = u * v$
$q_2 = p + q$

Answer Keys

Exercises

Practice Problems 1

1. D	**2.** D	**3.** C	**4.** C	**5.** B	**6.** A	**7.** B	**8.** A	**9.** A	**10.** A
11. B	**12.** C	**13.** A	**14.** C	**15.** B					

Practice Problems 2

1. B	**2.** B	**3.** A	**4.** B	**5.** B	**6.** A	**7.** B	**8.** C	**9.** A	**10.** A
11. B	**12.** D	**13.** A	**14.** C	**15.** C	**16.** A	**17.** C	**18.** B	**19.** D	

Previous Years' Questions

1. 8	**2.** B	**3.** 10	**4.** D	**5.** B

Chapter 4

Code Optimization

LEARNING OBJECTIVES

- *Code optimization basics*
- *Principle sources of optimization*
- *Loop invariant code motion*
- *Strength reduction on induction variables*
- *Loops in flow graphs*
- *Pre-header*
- *Global data flow analysis*
- *Definition and usage of variables*
- *Use-definition (u-d) chaining*
- *Data flow equations*

Code Optimization Basics

The process of improving the intermediate code and the target code in terms of both speed and the amount of memory required for execution is known as code optimization.

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

Properties of the transformations of an optimizing compiler are

1. A transformation must preserve the meaning of programs.
2. It must speed up programs by a measurable amount.
3. A transformation must be worth the effort.

Places for improvements

1. Source Code:
   User can – profile a program
   – change an algorithm
   – transform loops
2. Intermediate code can be improved by improving
   – Loops
   – Procedure calls
   – Address calculations
3. Target code can be improved by
   – Using registers
   – Selecting instructions
   – Peephole transformations

Optimizing compiler organization

This applies
- Control flow analysis
- Data flow analysis
- Transformations

Issues in design of code optimization The issues in the design of code optimization are
1. Target machine characteristics
2. Target CPU architecture
3. Functional units

Target machine Optimization is done, according to the target machine characteristics. Altering the machine description parameters, one can optimize single piece of compiler code.

Target CPU architecture The issues to be considered for the optimization with respect to CPU architecture
1. Number of CPU registers
2. RISC Instruction set
3. CISC instruction set
4. Pipelining

Functional units Based on number of functional units, optimization is done. So that instructions can be executed simultaneously.

Principle Sources of Optimization

Some code improving transformation is Local transformations and some are Global transformations.

Local Transformations can be performed by looking only at a statement in a basic block. Otherwise it is global transformation.

Function Preserving Transformations

These transformations improve the program without changing the function it computes. Some of these transformations are

1. Common sub expression elimination
2. Copy propagation
3. Dead-code elimination
4. Loop optimization
   - Code motion
   - Induction variable elimination
   - Reduction in strength

Common sub expression elimination The process of identifying common sub expressions and eliminating their computation multiple times is known as common sub expression elimination.

Example: Consider the following program segment:

```
int sum_n, sum_n2, sum_n3;
int sum (int n)
{
Sum_n = ((n)*(n+1))/2;
sum_n2 = ((n)*(n+1)*(2n+1))/6;
sum_n3 = (((n)*(n+1))/2)*(((n)*(n+1))/2;
}
```

Three Address code for the above input is

(0) Proc-begin sum
(1) t_0: = n + 1
(2) t_1: = n * t_0
(3) t_2: = t_1/2
(4) sum_n = t_2
(5) t_3: = n + 1
(6) t_4: = n * t_3
(7) t_5: 2 * n
(8) t_6: = t_5 + 1
(9) t_7: = t_4 * t_6
(10) t_8: = t_7/6
(11) sum_n_2: = t_8
(12) t_9: = n + 1
(13) t_{10}: n * t_9
(14) t_{11}: t_{10}/2
(15) t_{12}: = n + 1
(16) t_{13}: = n * t_{12}
(17) t_{14}: = t_{13}/2
(18) t_{15}: = t_{11} * t_{14}
(19) sum_n_3: = t_{15}
(20) label L_o
(21) Proc end sum

The computations made in quadruples

(1) – (3), (12) – (14), (15) – (17) are essentially same. That is, ((n)*(n + 1))/2 is computed.

It is the common sub expression.

This common sub expression is computed four times in the above example.

It is possible to optimize the code to have common sub expressions computed only once and then reuse the computed values further.

∴ Optimized intermediate code will be

(0) proc-begin sum
(1) t_0: = n + 1
(2) t_1: = n * t_0
(3) sultan: = t_1/2
(4) t_5: = 2 * n
(5) t_6: = t_5 + 1
(6) t_7: = t_1 * t_6
(7) sum_n2: = t_7/6
(8) sum_n3: sum_n * sum_n
(9) proc-end sum

Constant folding The constant expressions in the input source are evaluated and replaced by the equivalent values at the time of compilation.

For example 10*3, 6 + 101 are constant expressions and they are replaced by 30, 107 respectively.

Example: Consider the following '*C*' code:

```
int arr1 [10];
int main ( )
{
   arr1 [0] = 3;
   arr1 [1] = 4;
}
```

Unoptimized three address code equivalent to the above '*C*' code is

(0) proc-begin main
(1) t_0: = 0*4
(2) t_1: = &arr1
(3) t_1 [t_0]: = 3
(4) t_2: = 1*4
(5) t_3: = &arr1
(6) t_3 [t_2]: = 4
(7) Label L_0
(8) Proc – end main

In the above code, 0*4 is a constant expression its value = 0. 1*4 is a constant expression, its value = 4.

∴ After applying constant folding, optimized code will be

(0) proc-begin main
(1) t_0: = 0
(2) t_1: = &arr1
(3) t_1 [t_0]: = 3
(4) t_2: = 4

(5) t_3: = &arr1
(6) t_3 [t_2]: = 4
(7) label L_0
(8) proc – end main

Copy propagation In copy propagation, if there is an expression $x = y$ then use the variable 'y' instead of 'x'. This propagated in the statements following $x = y$.

Example: In the previous example, there are two copy statements.

(1) $t_0 = 0$
(2) $t_2 = 4$

After applying copy propagation, the optimized code will be

(0) proc-begin main
(1) t_0: = 0
(2) t_1: = &arr1
(3) t_1 [0]: = 3
(4) t_2: = 4
(5) t_3: = &arr1
(6) t_3 [4]: = 4
(7) Label L_0
(8) proc-end main

In the three address code shown above, quadruples (1) and (4) are no longer used in any of the following statements.

∴ (1) and (4) can be eliminated.

Three address code after dead store elimination

(0) proc-begin main
(1) t_1: = &arr1
(2) t_1 [0]: = 3
(3) t_3: = &arr1
(4) t_3 [4]: = 4
(5) Label L_0
(6) proc-end main

In the above example, we are propagating constant values. It is also known as constant propagation.

Variable propagation Propagating another variable instead of the existing one is known as variable propagation.

Example: int func(int *a*, int *b*, int *c*)

```
{
    int d, e, f;
    d = a;
    If (a > 10)
    {
       e = d + b;
    }
    Else
    {
       e = d + c;
    }
    f = d*e;
    return (f);
}
```

Three address code (unoptimized):

(0) proc-begin func
(1) $d := a$
(2) if a >10 goto L_0
(3) goto L_1
(4) label : L_0
(5) $e := d + b$
(6) goto L_2
(7) label : L_1
(8) $e := d + c$
(9) label : L_2
(10) $f := d * e$
(11) return f
(12) goto L_3
(13) label : L_3
(14) proc-end func

Three address code after variable (copy) propagation:

(0) proc-begin func
(1) d: = a
(2) If a >10 goto .L_0
(3) goto L_1
(4) label: L_0
(5) e: = $a + b$
(6) goto L_2
(7) label: L_1
(8) e: = $a + c$
(9) label: L_2
(10) f: = a*e
(11) return f
(12) goto L_3
(13) label: L_3
(14) proc-end func

After dead store elimination:
In the above code (1) d: = a is no more used
∴ Eliminate the dead store d: = a

(0) proc-begin func
(1) If a > 10 goto L_0
(2) goto L_1
(3) label: L_0
(4) e: = $a + b$
(5) goto L_2
(6) label: L_1
(7) e: $a + c$
(8) label: L_2
(9) f: = a*e
(10) return f
(11) goto L_3
(12) label: L_3
(13) proc-end func

Dead code elimination Eliminating the code that never gets executed by the program is known as Dead code elimination. It reduces the memory required by the program

Example: Consider the following Unoptimized Intermediate code:

(0) proc-begin func
(1) debug: = 0
(2) If debug = = 1 goto L_0
(3) goto L_1
(4) label: L_0
(5) param c
(6) param b
(7) param a
(8) param lc1
(9) call printf 16
(10) retrieve to
(11) label: L_1
(12) t_1: = $a + b$
(13) t_2: = $t_1 + c$
(14) v_1: = t_2
(15) Return v_1
(16) goto L_2
(17) label: L_2
(18) proc-end func

In copy propagation, debug is replaced with 0, wherever debug is used after that assignment.

∴ Statement 2 will be changed as

If 0 = = 1 goto L_0

0 = = 1, always returns false.

∴ The control cannot flow to label: L_0

This makes the statements (4) through (10) as dead code. (2) Can also be removed as part of dead code elimination. (1) Cannot be eliminated, because 'debug' is a global variable. The optimized code after elimination of dead code is shown below.

(0) proc-begin func
(1) debug: = 0
(2) goto L_1
(3) label: L_1
(4) t_1: = $a + b$
(5) t_2: = $t_1 + c$
(6) v_1: = t_2
(7) return v_1
(8) goto L_2
(9) label: L_2
(10) proc-end func

Algebraic transformations We can use algebraic identities to optimize the code further. For example

Additive Identity: $a + 0 = a$

Multiplicative Identity: a*1 = a

Multiplication with 0: a*0 = 0

Example: Consider the following code fragment:

```
struct mystruct
{
int a [20];
int b;
} xyz;
int func(int i)
{
xyz.a[i] = 34;
}
```

The Unoptimized three address code:

(0) proc-begin func
(1) t_0: = &xyz
(2) t_1: = 0
(3) t_2: = i*4
(4) t_1: = $t_2 + t_1$
(5) t_0 [t_1] = 34
(6) label: L_0
(7) proc-end func

Optimized code after copy propagation and dead code elimination is shown below:

The statement t_1: = 0 is eliminated.

(0) proc-being func
(1) t_0 =: = &xyz
(2) t_2: = i*4
(3) t_1 : = $t_2 + 0$
(4) t_0 [t_1]: = 34
(5) label: L_0
(6) proc-end func

After applying additive identity:

(0) proc-begin func
(1) t_0: = &xyz
(2) t_2: = i*4
(3) t_1 : = t_2
(4) t_0 [t_1]: = 34
(5) label: L_0
(6) proc-end func

After copy propagation and dead store elimination:

(0) proc-begin func
(1) t_0: = &xyz
(2) t_2 : = i*4
(3) t_0 [t_2]: = 34
(4) label: L_0
(5) proc-end func

Strength reduction transformation This transformation replaces expensive operators by equivalent cheaper ones on the target machine.

For example $y := x*2$ is replaced by $y := x + x$ as addition is less expensive than multiplication.

Similarly
Replace $y := x*32$ by $y := x << 5$
Replace $y := x/8$ by $y := x >> 3$

Loop optimization We can optimize loops by

(1) Loop invariant code motion transformation.
(2) Strength reduction on induction variable transformation.

Loop invariant code motion

The statements within a loop that compute value, which do not vary throughout the life of the loop are called loop invariant statements.

Consider the following program fragment:

```
int a [100];
int func(int x, int y)
{
int i;
int n1, n2;
i = 0;
n1 = x*y;
n2 = x - y;
while (a[i] > (n1*n2))
i = i + 1;
return(i);
}
```

The Three Address code for above program is

(0) proc-begin func
(1) $i := 0$
(2) $n_1 := x*y$
(3) $n_2 := x - y$
(4) label : L_0
(5) $t_2 := i*4$
(6) $t_3 := \&arr$
(7) $t_4 := t_3[t_2]$
(8) $t_5 := n_1 * n_2$
(9) if $t_4 > t_5$ goto L_1
(10) goto L_2
(11) label : L_1
(12) $i := i + 1$
(13) goto L_0
(14) label : L_2
(15) return i
(16) goto L_3
(17) label : L_3
(18) proc-end func

In the above code statements (6) and (8) are invariant.

After loop invariant code motion transformation the code will be

(0) proc-begin func
(1) $i := 0$
(2) $n_1 := x*y$
(3) $n_2 := x-y$
(4) $t_3 := \&arr$
(5) $t_5 := n_1*n_2$
(6) label : L_0
(7) $t_2 := i*4$
(8) $t_4 := t_3[t_2]$
(9) if $t_4 > t_5$ goto L_1
(10) goto L_2
(11) label : L_1
(12) $i := i + 1$
(13) goto L_0
(14) label : L_2
(15) return i
(16) goto L_3
(17) label : L_3
(18) proc-end func

Strength reduction on induction variables

Induction variable: A variable that changes by a fixed quantity on each of the iterations of a loop is an induction variable.

Example: Consider the following code fragment:

```
int i;
int a[20];
int func( )
{
    while(i<20)
    {
    a[i] = 10;
    i = i + 1;
    }
}
```

The three-address code will be

(0) proc-begin func
(1) label : L_0
(2) if $i < 20$ goto L_1
(3) goto L_2
(4) label : L_1
(5) $t_0 := i*4$
(6) $t_1 := \&a$
(7) $t_1[t_0] := 10$
(8) $i := i + 1$
(9) goto L_0
(10) label : L_2
(11) label : L_3
(12) proc-end func

After reduction of strength the code will be
Here (5) $t_0 = i*4$ is moved out of the loop and (8) is followed by $t_0 = t_0 + 4$.

(0) proc-begin func
(0a) $t_0 := i*4$
(1) label : L_0
(2) if $i < 20$ goto L_1
(3) goto L_2
(4) label:L_1
(5)
(6) $t_1 := \&a$
(7) $t_1[t_0] := 10$
(8) $i := i + 1$
(8a) $t_0 := t_0 + 4$
(9) goto L_0
(10) label : L_2
(11) label : L_3
(12) proc-end func

Loops in Flow Graphs

Loops in the code are detected during the data flow analysis by using the concept called 'dominators' in the flow graph.

Dominators

A node d of a flow graph dominates node n, if every path from the initial node to 'n' goes through 'd'.

It is represented as d dom n.

Notes:

1. Each and every node dominates itself.
2. Entry of the loop dominates all nodes in the loop.

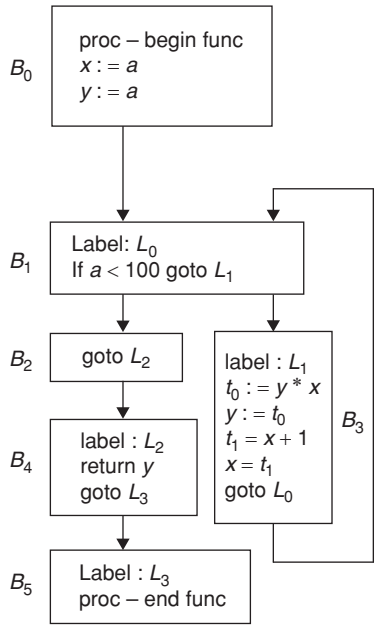
Example: Consider the following code fragment:

```
int func(int a)
{
int x, y;
x = a;
y = a;
While (a < 100)
{
y = y*x;
x = x+1;
}
return(y);
}
```

The Three Address code after local optimization will be

(0) proc-begin func
(1) $x := a$
(2) $y := a$
(3) label: L_0
(4) if $a < 100$ goto L_1
(5) goto L_2
(6) label: L_1
(7) $t_0 := y*x$
(8) $y := t_0$
(9) $t_1 := x + 1$
(10) $x := t_1$
(11) goto L_0
(12) label: L_2
(13) return y
(14) goto L_3
(15) label: L_3
(16) proc-end func

The Flow Graph for above code will be:



To reach B_2, it must pass through B_1
$\therefore$ B_1 dominates B_2. Also B_0 dominates B_2.
dominators $[B_1] = \{B_0, B1\}$ (or) dominators $[1] = \{0, 1\}$

The dominators for each of the nodes in the flow graph are

dominators [0] = {0}
dominators [1] = {0, 1}
dominators [2] = {0, 1, 2}
dominators [3] = {0, 1, 3}
dominators [4] = {0, 1, 2, 4}
dominators [5] = {0, 1, 2, 4, 5}

Edge

An edge in a flow graph represents a possible flow of control.

In the flow graph, B_0 to B_1 edge is represented as $0 \rightarrow 1$.

Head and tail: In the edge $a \rightarrow b$, the node b is called head and the node a is called as tail.

Back edges: There are some edges in which dominators [tail] contains the head.

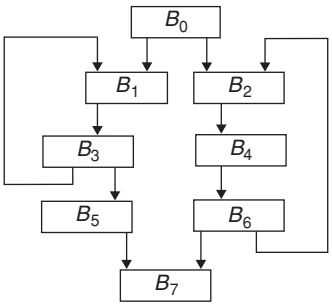
The presence of a back edge indicates the existence of a loop in a flow graph.

In the previous graph, $3 \rightarrow 1$ is a back edge.

Consider the following table:

Edge	Head	Tail	Dominators [head]	Dominators [tail]
$0 \rightarrow 1$	1	0	{0, 1}	{0}
$1 \rightarrow 2$	2	1	{0, 1, 2}	{0, 1}
$1 \rightarrow 3$	3	1	{0, 1, 3}	{0, 1}
$3 \rightarrow 1$	1	3	{0, 1}	{0, 1, 3}
$2 \rightarrow 4$	4	2	{0, 1, 2, 4}	{0, 1, 2}
$4 \rightarrow 5$	5	4	{0, 1, 2, 4, 5}	{0, 1, 2, 4}

Example: Consider below flow graph:



The dominators of each node are

dominators [0] = {0}
dominators [1] = {0, 1}
dominators [2] = {0, 2}
dominators [3] = {0, 1, 3}
dominators [4] = {0, 2, 4}
dominators [5] = {0, 1, 3, 5}
dominators [6] = {0, 2, 4, 6}
dominators [7] = {0, 7}

Edge	Head	Tail	Dominators [head]	Dominators [tail]
$0 \rightarrow 1$	1	0	{0, 1}	{0}
$0 \rightarrow 2$	2	0	{0, 2}	{0}
$1 \rightarrow 3$	3	1	{0, 1, 3}	{0, 1}
$3 \rightarrow 1$	1	3	{0, 1}	{0, 1, 3}
$3 \rightarrow 5$	5	3	{0, 1, 3, 5}	{0, 1, 3} Backedge
$5 \rightarrow 7$	7	5	{0, 7}	{0, 1, 3, 5}
$2 \rightarrow 4$	4	2	{0, 2, 4}	{0, 2}
$6 \rightarrow 2$	2	6	{0, 2}	{0, 2, 4, 6} Backedge
$4 \rightarrow 6$	6	4	{0, 2, 4, 6}	{0, 2, 4}
$6 \rightarrow 7$	7	6	{0, 7}	{0, 2, 4, 6}

Here $\{B_6, B_2, B_4\}$ form a loop (L_1), $\{B_3, B_1\}$ form another loop (L_2)

In a loop, the entry of the loop dominates all nodes in the loop.

Header of the loop The entry of the loop is also called as the header of the loop.

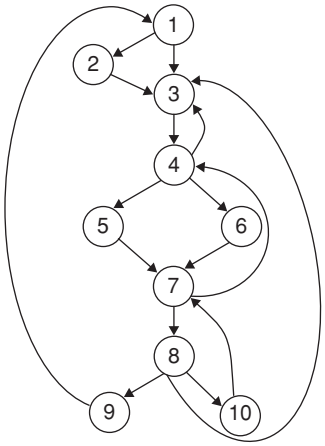
Loop exit block In loop L_1 can be exited from the basic block B_6. It is called loop exit block. The block B_3 is the loop exit block for the loop L_2. It is possible to have multiple exit blocks in a loop.

Dominator tree

A tree, which represents dominate information in the form of tree is a dominator tree. In this,

- The initial node is the root.
- Each node d dominates only its descendents in the tree.

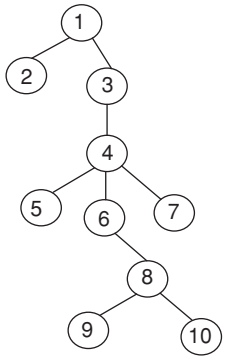
Consider the flow graph



The dominators of each node are

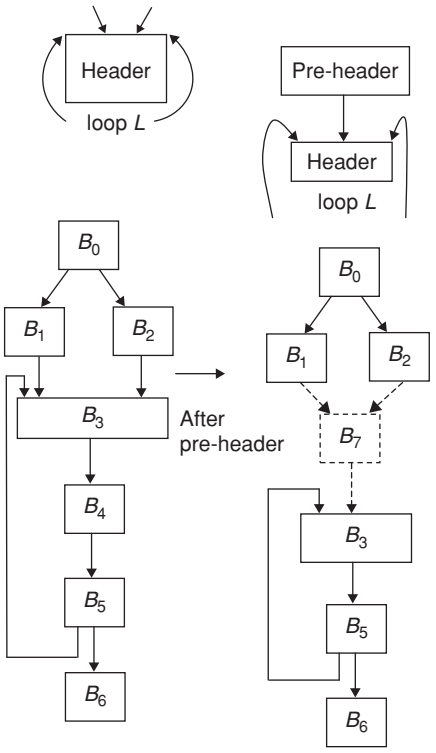
dominators [1] = {1}
dominators [2] = {1, 2}
dominators [3] = {1, 3}
dominators [4] = {1, 3, 4}
dominators [5] = {1, 3, 4, 5}
dominators [6] = {1, 3, 4, 6}
dominators [7] = {1, 3, 4, 7}
dominators [8] = {1, 3, 4, 7, 8}
dominators [9] = {1, 3, 4, 7, 8, 9}
dominators [10] = {1, 3, 4, 7, 8, 10}

The dominator tree will be:



Pre-header

A pre-header is a basic block introduced during the loop optimization to hold the statements that are moved from within the loop. It is a predecessor to the header block.



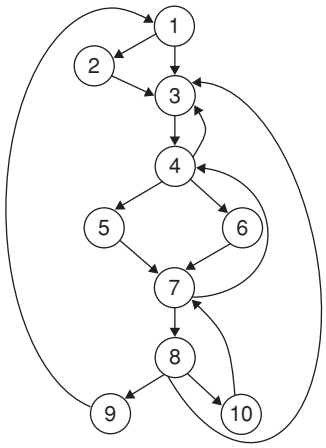
Reducible Flow Graphs

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups:

(1) Forward edges
(2) Backward edges with the following properties.

(i) The forward edges form an acyclic graph in which every node can be reached from the initial node of G.
(ii) The back edges consist only of edges whose heads dominates their tails.

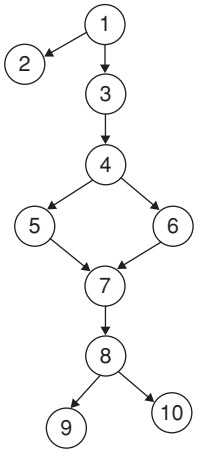
Example: Consider previous flow graph



In the above flow graph, there are five back edges

$$4 \to 3,\ 7 \to 4,\ 8 \to 3,\ 9 \to 1 \text{ and } 10 \to 7$$

Remove all backedges.
The remaining edges must be the forward edges.
The remaining graph is acyclic.



$\therefore$ It is reducible.

Global Dataflow Analysis

Point: A point is a place of reference that can be found at

1. Before the first statement in a basic block.
2. After the last statement in a basic block.
3. In between two adjacent statements within a basic block.

Example 1:

$$\boxed{\begin{array}{l} a = 10 \\ b = 20 \\ c = a * b \end{array}}\ B_1$$

Here, In B_1 there are 4 points

Example 2:

B_1
- $P_1 - B_1$ proc-begin func
- $P_2 - B_1$ $v_3 = v_1 + v_2$
- $P_3 - B_1$ if $c > 100$ goto L_0
- $P_4 - B_1$

There is 4 point in the basic block B_1, given by $P_1 - B_1$, $P_2 - B_1$, $P_3 - B_1$ and $P_4 - B_1$.

Path: A path is a sequence of points in which the control can flow.

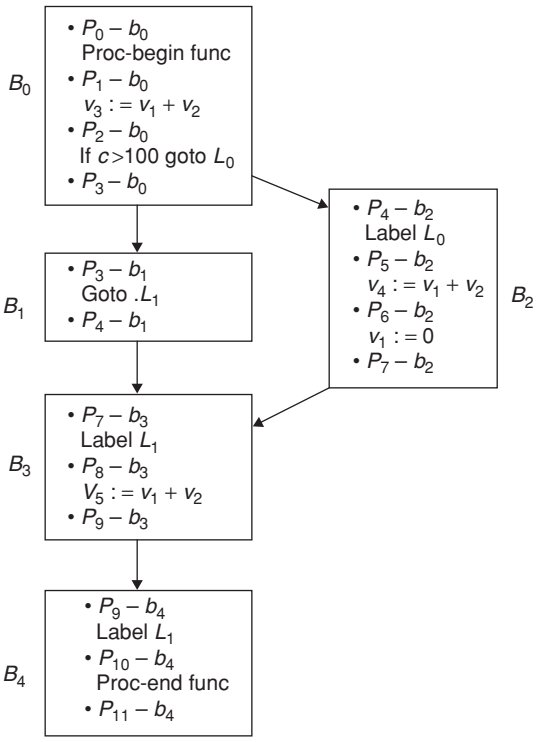
A path from P_1 to P_n is a sequence of points $P_1, P_2, \ldots, P_n$ such that for each i between 1 and n-1, either

(a) P_i is the point immediately preceding a statement and P_{i+1} is the point immediately following that statement in the same block.

(OR)

(b) P_i is the end of some block and P_{i+1} is the beginning of a successor block.

Example:



Path is between the points $P_0 - b_0$ and $P_6 - b_2$:

The sequence of points $P_0 - b_0$, $P_1 - b_0$, $P_2 - b_0$, $P_3 - b_0$, $P_4 - b_2$, $P_5 - b_2$ and $P_6 - b_2$.

Path between $P_3 - b_1$ and $P_6 - b_2$: There is no sequence of points.

Path between $P_0 - b_0$ and $P_7 - b_3$: There are two paths.

(1) Path 1 consists of the sequence of points, $P_0 - b_0, P_1 - b_0$, $P_2 - b_0, P_3 - b_0, P_3 - b_0, P_4 - b_1$ and $P_7 - b_3$.

(2) Path 2 consists of the sequence of points $P_0 - b_0, P_1 - b_0$, $P_2 - b_0, P_3 - b_0, P_4 - b_2, P_5 - b_2, P_6 - b_2, P_7 - b_2$ and $P_7 - b_3$

Definition and Usage of Variables

Definitions

It is either an assignment to the variable or reading of a value for the variable.

Use

Use of identifier x means any occurrence of x as an operand.

Example: Consider the statement

$$x = y + z;$$

In this statement some value is assigned to x. It defines x and used y and z values.

Global Data-Flow-Analysis

Data Flow Analysis (DFA) is a technique for gathering information about the possible set of values calculated at various points in a program.

- An example of a data-flow analysis is reaching definitions.
- A single way to perform data-flow analysis of program is to setup data flow equations for each node of the control flow graph.

Use definition (U-d) chaining

The use of a value is any point where that variable or constant is used in the right hand side of an assignment or is evaluating an expression.

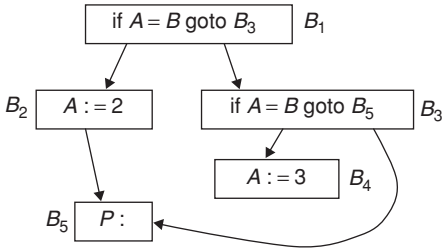
The definition of a value occurs implicitly at the beginning of the whole program for a variable.

A point is defined either prior to or immediately after a statement.

Reaching definitions

A definition of a variable A reaches a point P if there is a path in the flow graph from that definition to P, such that no other definitions of A appear on the path.

Example:



The definition $A: = 3$ can reach point p in B_5.

To determine the definitions that can reach a given program first assign distinct numbers to each definition, since it is associated with a unique quadruple.

- For each simple variable A, make a list of all definitions of A anywhere in the program.
- Compute two sets for each basic block B.

Gen [B] is the set of generated definitions within block B and that reach the end of the block.

1. Kill [B], which is the set of definitions outside of B that define identifiers that also have definitions within B.
2. IN [B], which are all definitions reaching the point just before B's first statement.

Once this is known, the definitions reaching any use of A within B are found by:

Let u be the statement being examined, which uses A.

1. If there are definitions of A within B before u, the last is the only one reaching u.
2. If there is no definition of A within B prior to u, those reaching u are in IN [B].

Data Flow Equations

1. For all blocks B,

$$\text{OUT}[B] = (\text{IN}[B] - \text{KILL}[B]) \cup \text{GEN}[B]$$

A definition d, reaches the end of B if

(a) $d \in$ IN [B] and is not killed by B.
(or)
(b) d is generated in B and is not subsequently redefined here.

2. IN [B] = U OUT [P]
$\forall P$ preceding B
A definition reaches the beginning of B iff it reaches the end of one of its predecessors.

Computing U-d Chains

If a use of variable 'a' is preceded in its block by a definition of 'a', this is the only one reaching it.

If no such definition precedes its use, all definitions of 'a' in IN [B] are on its chain.

Uses of U-d Chains

1. If the only definition of 'a' reaching this statement involves a constant, we can substitute that constant for 'a'.
2. If no definitions of 'a' reaches this point, a warning can be given.
3. If a definition reaches nowhere, it can be eliminated. This is part of dead code elimination.

EXERCISES

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Replacing the expression 2 * 3.14 by 6.28 is
(A) Constant folding
(B) Induction variable
(C) Strength reduction
(D) Code reduction

2. The expression (a*b)*c op … where 'op' is one of '+', '*' and '↑' (exponentiation) can be evaluated on CPU with a single register without storing the value of (a*b) if
(A) 'op' is '+' or '*'
(B) 'op' is '↑' or '+'
(C) 'op' is '↑' or '*'
(D) not possible to evaluate without storing

3. Machine independent code optimization can be applied to
(A) Source code
(B) Intermediate representation
(C) Runtime output
(D) Object code

4. In block B if S occurs in B and there is no subsequent assignment to y within B, then the copy statement $S : x = y$ is
(A) Generated (B) Killed
(C) Blocked (D) Dead

5. If E was previously computed and the value of variable in E have not changed since previous computation, then an occurrence of an expression E is
(A) Copy propagation
(B) Common sub expression
(C) Dead code
(D) Constant folding

6. In block B, if x or y is assigned there and s is not in B, then $s : x = y$ is
(A) Generated (B) Killed
(C) Blocked (D) Dead

7. Given the following code
$A = x + y$;
$B = x + y$;
Then the corresponding optimized code as
_____
_____
$C = x + y$;
_____
$A = C$;
_____
$B = C$;
When will be optimized code pose a problem?
(A) When C is undefined.
(B) When memory is consideration.
(C) C may not remain same after some statements.
(D) Both (A) and (C).

8. Can the loop invariant $X = A - B$ from the following code be moved out?
For i = 1 to 10
```
{
A = B * C;
X = A - B;
}
```
(A) No
(B) Yes
(C) $X = A - B$ is not invariant
(D) Data insufficient

9. If every path from the initial node goes through a particular node, then that node is said to be a
(A) Header (B) Dominator
(C) Parent (D) Descendant

Common data for questions 10 and 11: Consider the following statements of a block:
$a: = b + c$
$b: = a - d$
$c: = b + c$
$d: = a - d$

10. The above basic block contains, the value of b in 3rd statement is
(A) Same as b in 1st statement
(B) Different from b in 1st statement
(C) 0
(D) 1

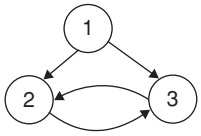
11. The above basic block contains
(A) Two common sub expression
(B) Only one common sub expression
(C) Dead code
(D) Temporary variable

12. Find the induction variable from the following code:
$A = -0.2$;
$B = A + 5.0$;
(A) A
(B) B
(C) Both A and B are induction variables
(D) No induction variables

13. The analysis that cannot be implemented by forward operating data flow equations mechanism is
(A) Interprocedural
(B) Procedural
(C) Live variable analysis
(D) Data

14. Which of the following consist of a definition, of a variable and all the uses, U, reachable from that definition without any other intervening definitions?
(A) Ud-chaining (B) Du-chaining
(C) Spanning (D) Searching

15. Consider the graph



The graph is
(A) Reducible graph
(B) Non-reducible graph
(C) Data insufficient
(D) None of these

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. In labeling algorithm, let n is a binary node and its children have L_1 and L_2, if $L_1 = L_2$ then LABEL (n):
(A) $L_1 - 1$ (B) $L_2 + 1$
(C) $L_1 + L_1$ (D) $L_1 + 1$

2. The input for the code generator is a:
(A) Tree at lexical level
(B) Tree at semantic level
(C) Sequence of assembly language instructions
(D) Sequence of machine idioms

3. In labeling algorithm, let n is a binary node and its children have i_1 and i_2, LABEL (n) if $i_1 \neq i_2$ is
(A) Max (i_1, i_2)
(B) $i_2 + 1$
(C) $i_2 - 1$
(D) $i_2 - i_1$

4. The following tries to keep frequently used value in a fixed register throughout a loop is:
(A) Usage counts
(B) Global register allocation
(C) Conditional statement
(D) Pointer assignment

5. Substitute y for x for copy statement $s : x = y$ if the following condition is met
(A) Statements s may be the only definition of x reaching u
(B) x is dead
(C) y is dead
(D) x and y are aliases

6. Consider the following code

```
for (i=0; i<m; i++)
{
for (j=0; j<m; j ++)
If (i%2)
{
a = a + (14*j+5*i);
b = b + (9 + 4*j);
}
}
```

Which of the following is false?
(A) There is a scope of common reduction in this code
(B) There is a scope of strength reduction in this code.
(C) There is scope of dead code elimination in this code
(D) Both (A) and (C)

7. S_1: In dominance tree, the initial node is the root.
S_2: Each node d dominates only its ancestors in the tree.
S_3: if $d \neq n$ and d dom n then d dom m.
Which of the statements is/are true?
(A) S_1, S_2 are true
(B) S_1, S_2 and S_3 are true

(C) Only S_3 is true
(D) Only S_1 is true

8. The specific task storage manager performs:
(A) Allocation/Deallocation of storage to programs
(B) Protection of storage area allocated to a program from illegal access by other programs in the system
(C) The status of each program
(D) Both (A) and (B)

9. Concept which can be used to identify loops is:
(A) Dominators
(B) Reducible graphs
(C) Depth first ordering
(D) All of these

10. A point cannot be found:
(A) Between two adjacent statements
(B) Before the first statement
(C) After the last statement
(D) Between any two statements

11. In the statement, $x = y*10 + z$; which is/are defined?
(A) x (B) y
(C) z (D) Both (B) and (C)

12. Consider the following program:

```
void main ( )
{
    int x, y;
    x = 3; y = 7;
    --------
    --------
    if (x<y)
    {
    int x;
    {
    int y;
    y = 9;
    ------
    x = 2*y;
    }
    -------
    -------
    x = x + y;
    printf ("%d", x);
    }
    ------
    printf ("%d", x);
    }
```

The output is
(A) 3 – 25 (B) 25 – 3
(C) 3 – 3 (D) 25 – 25

13. The evaluation strategy which delays the evaluation of an expression until its value is needed and which avoids repeated evaluations is:
(A) Early evaluation (B) Late evaluation
(C) Lazy evaluation (D) Critical evaluation

14. If two or more expressions denote same memory address, then the expressions are:
(A) Aliases (B) Definitions
(C) Superiors (D) Inferiors

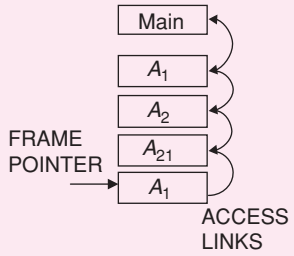
15. Operations that can be removed completely are called:
(A) Strength reduction
(B) Null sequences
(C) Constant folding
(D) None of these

Previous Years' Questions

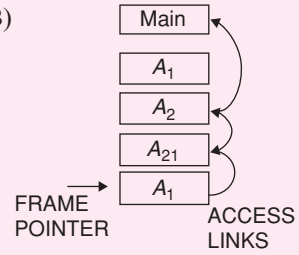
1. In a compiler, keywords of a language are recognized during: **[2011]**
(A) parsing of the program
(B) the code generation
(C) the lexical analysis of the program
(D) dataflow analysis

2. Consider the program given below, in a block structured pseudo-language with lexical scoping and nesting of procedures permitted. **[2012]**
Program main;
Var …
Procedure A_1;
Var …
Call A_2;
End A_1
Procedure A_2;
Var …
Procedure A_{21};
Var …
Call A_1;
End A_{21}
Call A_{21};
End A_2
Call A_1;
End main
Consider the calling chain: Main $\rightarrow A_1 \rightarrow A_2 \rightarrow A_{21} \rightarrow A_1$
The correct set of activation records along with their access links is given by:

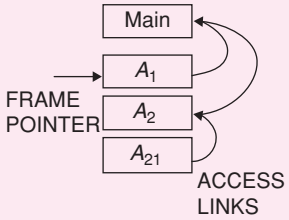
(A)



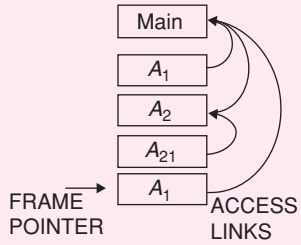
(B)



(C)



(D)



Common data for questions 3 and 4: The following code segment is executed on a processor which allows only register operands in its instructions. Each instruction can have atmost two source operands and one destination operand. Assume that all variables are dead after this code segment.

```
c = a + b;
d = c * a;
e = c + a;
x = c * c;
If (x > a) {
   y = a * a;
}
Else {
      d = d * d;
      e = e * e;
}
```

3. What is the minimum number of registers needed in the instruction set architecture of the processor to compile this code segment without any spill to memory? Do not apply any optimization other than optimizing register allocation. **[2013]**

(A) 3 (B) 4
(C) 5 (D) 6

4. Suppose the instruction set architecture of the processor has only two registers. The only allowed compiler optimization is code motion, which moves statements from one place to another while preserving correctness. What is the minimum number of spills to memory in the compiled code? **[2013]**

(A) 0 (B) 1
(C) 2 (D) 3

5. Which one of the following is NOT performed during compilation? **[2014]**

(A) Dynamic memory allocation
(B) Type checking
(C) Symbol table management
(D) Inline expansion

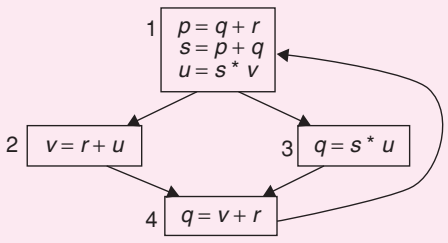
6. Which of the following statements are CORRECT? **[2014]**

(i) Static allocation of all data areas by a compiler makes it impossible to implement recursion.
(ii) Automatic garbage collection is essential to implement recursion.
(iii) Dynamic allocation of activation records is essential to implement recursion.
(iv) Both heap and stack are essential to implement recursion.

(A) (i) and (ii) only (B) (ii) and (iii) only
(C) (iii) and (iv) only (D) (i) and (iii) only

7. A variable x is said to be live at a statement S_i in a program if the following three conditions hold simultaneously: **[2015]**

1. There exists a statement S_j that uses x
2. There is a path from S_i to S_j in the flow graph corresponding to the program.
3. The path has no intervening assignment to x including at S_i and S_j.



The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph are

(A) p, s, u (B) r, s, u
(C) r, u (D) q, v

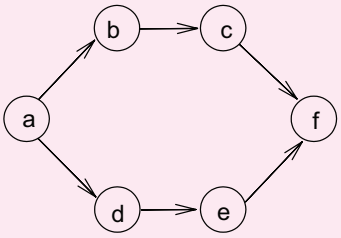
8. Match the following **[2015]**

P. Lexical analysis	1. Graph coloring
Q. Parsing	2. DFA minimization
R. Register allocation	3. Post-order traversal
S. Expression evaluation	4. Production tree

(A) P–2, Q–3, R–1, S–4 (B) P–2, Q–1, R–4, S–3

(C) P–2, Q–4, R–1, S–3 (D) P–2, Q–3, R–4, S–1

9. Consider the following directed graph:



The number of different topological orderings of the vertices of the graph is _____ . **[2016]**

10. Consider the following grammar:

stmt – > **if** expr **then** expr **else** expr; stmt | Ò
expr – > term **relop** term | term
term – > id | number
id – > **a** | **b** | **c**
number – > [**0** – **9**]

where **relop** is a relational operator (e.g., <, >,…), Ò refers to the empty statement, and **if, then, else** are terminals.

Consider a program *P* following the above grammar containing ten **if** terminals. The number of control flow paths in *P* is_____. For example, the program

if e_1 **then** e_2 **else** e_3

has 2 control flow paths, $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$. **[2017]**

11. Consider the expression $(a—1) * (((b + c)/3) + d)$. Let X be the minimum number of registers required by an *optimal* code generation (without any register spill) algorithm for a load/store architecture, in which *(i) only load and store instruction can have memory operands and (ii) arithmetic instructions can have only register or immediate operands*. The value of X is _____. **[2017]**

12. Match the following according to input (from the left column) to the compiler phase (in the right column) that processes it: **[2017]**

(P) Syntax tree	(i) Code generator
(Q) Character stream	(ii) Syntax analyzer
(R) Intermediate representation	(iii) Semantic analyzer
(S) Token stream	(iv) Lexical analyzer

(A) P → (ii), Q → (iii), R → (iv), S → (i)

(B) P → (ii), Q → (i), R → (iii), S → (iv)

(C) P → (iii), Q → (iv), R → (i), S → (ii)

(B) P → (i), Q → (iv), R → (ii), S → (iii)

Answer Keys

Exercises

Practice Problems 1

1. A **2.** C **3.** B **4.** A **5.** B **6.** B **7.** C **8.** B **9.** B **10.** B
11. B **12.** D **13.** C **14.** B **15.** B

Practice Problems 2

1. D **2.** B **3.** A **4.** B **5.** A **6.** D **7.** D **8.** D **9.** D **10.** D
11. A **12.** B **13.** C **14.** A **15.** B

Previous Years' Questions

1. C **2.** D **3.** B **4.** B **5.** A **6.** D **7.** C **8.** C **9.** 6 **10.** 1024
11. 2 **12.** C