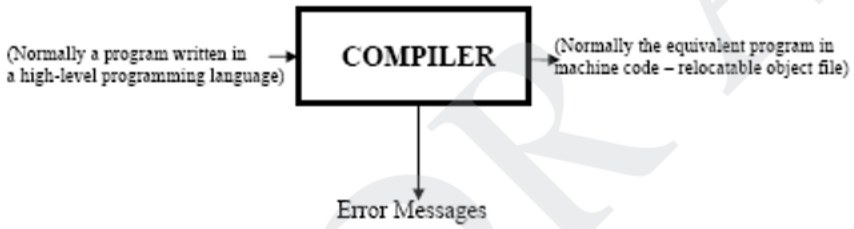
Anna University – 2017 Regulations
B.E. (COMPUTER SCIENCE AND ENGINEERING)
VI SEM CSE
CS8602 Compiler Design Question and answers
UNITS I & II Notes

---

1) What is a compiler?

COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Simply stated, a compiler is a program that reads a program written in one language-the source language-and translates it into an equivalent program in another language-the target language (see fig.1) As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform. Despite this apparent complexity, the basic tasks that any compiler must perform are essentially the same.

2) What are the phases of a compiler?

Phases of a Compiler

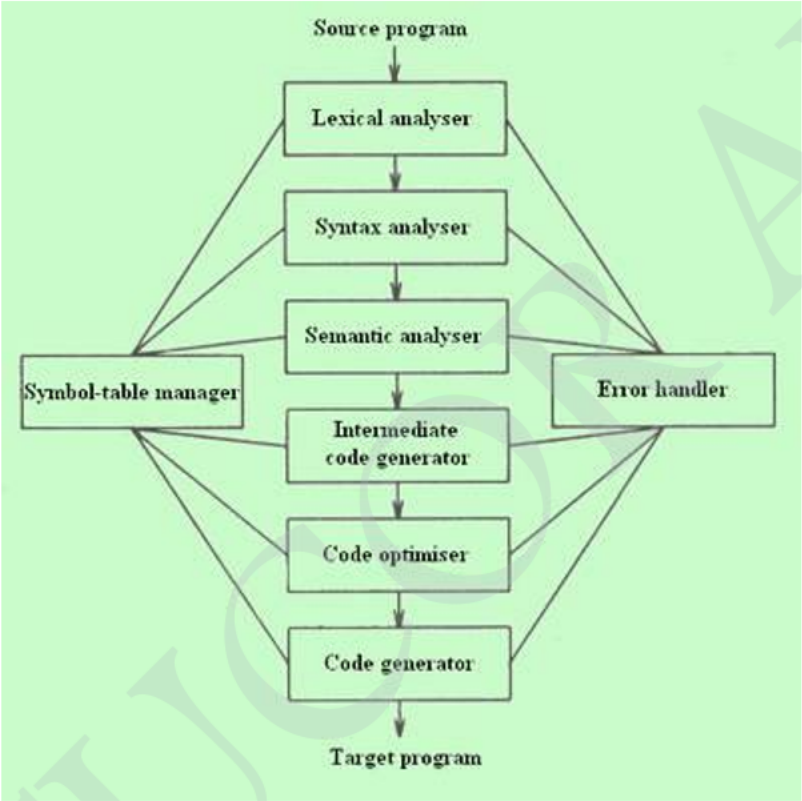
1. Lexical analysis ("scanning")
   - Reads in program, groups characters into "tokens"
2. Syntax analysis ("parsing")
   - Structures token sequence according to grammar rules of the language.
3. Semantic analysis
   - Checks semantic constraints of the language.
4. Intermediate code generation
   - Translates to "lower level" representation.
5. Program analysis and code optimization
   - Improves code quality.

6. Final code generation.

3) Explain in detail different phases of a compiler.

THE DIFFERENT PHASES OF A COMPILER

Conceptually, a compiler operates in *phases*, each of which transforms the source program from one representation to another.



The first three phases, forms the bulk of the analysis portion of a compiler. Symbol table management and error handling, are shown interacting with the six phases.

Symbol table management

An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. A *symbol table* is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that

record quickly. When an identifier in the source program is detected by the lex analyzer, the identifier is entered into the symbol table.

Error Detection and Reporting

Each phase can encounter errors. A compiler that stops when it finds the first error is not as helpful as it could be.

The syntax and semantic analysis phases usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Errors when the token stream violates the syntax of the language are determined by the syntax analysis phase. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.

The Analysis phases

As translation progresses, the compiler's internal representation of the source program changes. Consider the statement,

position := initial + rate * 10

The lexical analysis phase reads the characters in the source pgm and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword etc. The character sequence forming a token is called the *lexeme* for the token. Certain tokens will be augmented by a 'lexical value'. For example, for any identifier the lex analyzer generates not only the token id but also enter s the lexeme into the symbol table, if it is not already present there. The lexical value associated this occurrence of id points to the symbol table entry for this lexeme. The representation of the statement given above after the lexical analysis would be:

id1: = id2 + id3 * 10

Syntax analysis imposes a hierarchical structure on the token stream, which is shown by syntax trees (fig 3).

Intermediate Code Generation

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation can have a variety of forms.

In three-address code, the source pgm might look like this,

temp1: = inttoreal (10)

temp2: = id3 * temp1

temp3: = id2 + temp2

id1: = temp3

Code Optimisation

The code optimization phase attempts to improve the intermediate code, so that faster running machine codes will result. Some optimizations are trivial. There is a great variation in the amount of code optimization different compilers perform. In those that do the most, called 'optimising compilers', a significant fraction of the time of the compiler is spent on this phase.

Code Generation

The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program. Then, intermediate instructions are each translated into a sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

4) What is grouping of phases?

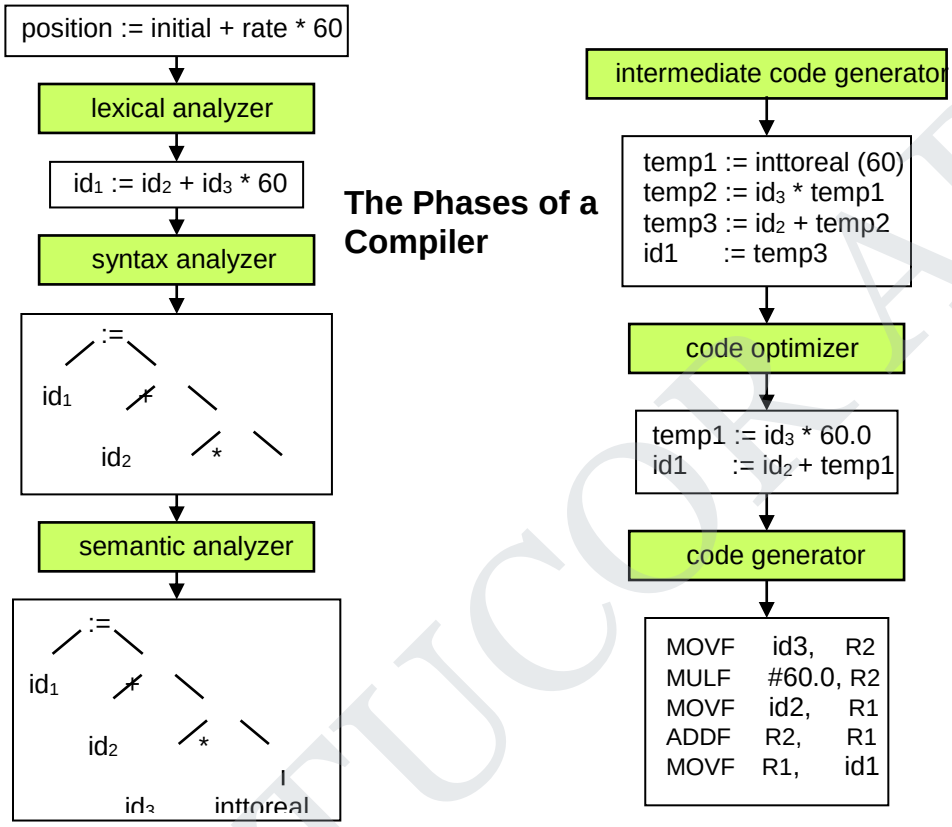
Grouping of Phases

- *Front end* : machine independent phases

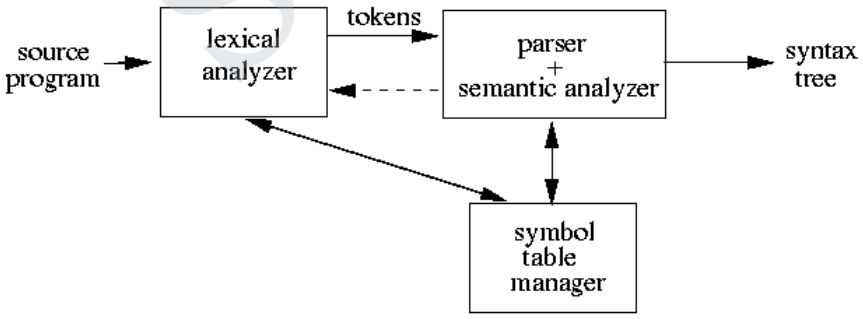
- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generation
- Some code optimization

- *Back end* : machine dependent phases
  - Final code generation
  - Machine-dependent optimizations

5) Explain with diagram how a statement is compiled .



6)**What are roles and tasks of a lexical analyzer?**



Main Task: Take a token sequence from the scanner and verify that it is a syntactically correct program.
Secondary Tasks:

- Process declarations and set up symbol table information accordingly, in preparation for semantic analysis.
- Construct a syntax tree in preparation for intermediate code generation.

- **Define Context free grammar.**
- **Context-free Grammars**
- A *context-free grammar* for a language specifies the syntactic structure of programs in that language.
- Components of a grammar:
  - a finite set of tokens (obtained from the scanner);
  - a set of variables representing "related" sets of strings, e.g., *declarations*, *statements*, expressions.
  - a set of rules that show the structure of these strings.
  - an indication of the "top-level" set of strings we care about.
- **Context-free Grammars: Definition**
- Formally, a context-free grammar G is a 4-tuple $G = (V, T, P, S)$, where:
  - V is a finite set of *variables* (or *nonterminals*). These describe sets of "related" strings.
  - T is a finite set of *terminals* (i.e., tokens).
  - P is a finite set of *productions*, each of the form
- $A \rightarrow \alpha$
- where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals.
  - S ∈ V is the *start symbol*.

Example of CFG :

E ==>EAE | (E) | -E | id

A==> + | - | * | / |

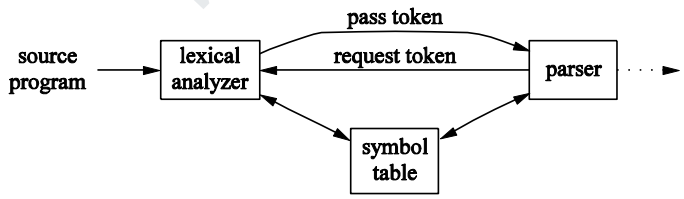
Where E,A are the non-terminals while id, +, *, -, /,(,) are the terminals.

6) What are parsers?

Parser

- Accepts string of tokens from lexical analyzer (usually one token at a time)
- Verifies whether or not string can be generated by grammar
- Reports syntax errors (recovers if possible)

THE ROLE OF A PARSER



Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion.

The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)

2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing.

7) What are parse trees?

Parse Trees

- Nodes are non-terminals.
- Leaves are terminals.
- Branching corresponds to rules of the grammar.
- The leaves give a sentence of the input language.
- For every sentence of the language there is at least one parse tree.
- Sometimes we have more then one parse tree for a sentence.
- Grammars which allow more than one parse tree for some sentences are called ambiguous and are usually not good for compilation.

8) What are different kinds of errors encountered during compilation?

Compiler Errors

- Lexical errors (e.g. misspelled word)
- Syntax errors (e.g. unbalanced parentheses, missing semicolon)
- Semantic errors (e.g. type errors)
- Logical errors (e.g. infinite recursion)

Error Handling

- Report errors clearly and accurately
- Recover quickly if possible
- Poor error recover may lead to avalanche of errors

9) What are different error recovery strategies?

Error Recovery strategies

- **Panic mode**: discard tokens one at a time until a synchronizing token is found
- **Phrase-level recovery**: Perform local correction that allows parsing to continue
- **Error Productions**: Augment grammar to handle predicted, common errors
- **Global Production**: Use a complex algorithm to compute least-cost sequence of changes leading to parseable code

10) Explain Recursive descent parsing.

Recursive descent parsing: corresponds to finding a leftmost derivation for an input string

Equivalent to constructing parse tree in pre-order

Example:

Grammar: S ! cAd A ! ab j a

Input: cad

Problems:

1. backtracking involved ()buffering of tokens required)
2. left recursion will lead to infinite looping
3. left factors may cause several backtracking steps

Compiler Construction: Parsing – p. 3/31

11) Give an example of ambiguous grammar.

Examples

Ambigous grammar:

E ::= E "_" E | E "+" E | "1" | "(" E ")"

Unambigous grammar

E ::= E "+" T | T

T ::= T "_" F | F

F ::= "1" | "(" E ")"

8) What is left recursion? How it is eliminated?

Left recursion: G istb left recursive if for some non-terminal A, $A \stackrel{+}{\Rightarrow} A\alpha$

Simple case I: $A \rightarrow A\alpha \mid \beta \quad \Rightarrow \quad A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \epsilon$

Simple case II:

$$\begin{array}{lcl} A & \rightarrow & A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \\ & & \beta_1 \mid \ldots \mid \beta_n \end{array}$$

$$\Downarrow$$

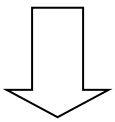
$$\begin{array}{lcl} A & \rightarrow & \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' & \rightarrow & \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon \end{array}$$

12) What is left factoring?

Left Factoring

- Rewriting productions to delay decisions
- Helpful for predictive parsing
- Not guaranteed to remove ambiguity

A → αβ1 | αβ2



A → αA'

A' → β1 | β2

Left factoring:

Example: *stmt* → **if (** *expr* **)** *stmt*
| **if (** *expr* **)** *stmt* **else** *stmt*

Algorithm:
while left factors exist **do**
for each non-terminal A **do**
Find longest prefix α common to ≥ 2 rules
Replace $A \to \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid (\ldots)$
by $A \to \alpha A' \mid (\ldots)$
$A' \to \beta_1 \mid \ldots \mid \beta_n$
end for
end while

13) What is top down parsing?

Top Down Parsing

- Can be viewed two ways:
  - Attempt to find leftmost derivation for input string
  - Attempt to create parse tree, starting from at root, creating nodes in preorder
- General form is recursive descent parsing
  - May require backtracking
  - Backtracking parsers not used frequently because not needed

14) What is predictive parsing?

- A special case of recursive-descent parsing that does not require backtracking
- Must always know which production to use based on current input symbol
- Can often create appropriate grammar:
  - removing left-recursion
  - left factoring the resulting grammar

15) Define LL(1) grammar.

LL(1) Grammars

- Algorithm covered in class can be applied to any grammar to produce a parsing table
- If parsing table has no multiply-defined entries, grammar is said to be "LL(1)"
  - First "L", left-to-right scanning of input
  - Second "L", produces leftmost derivation
  - "1" refers to the number of lookahead symbols needed to make decisions

16) What is shift reduce parsing?

Shift-Reduce Parsing

- One simple form of bottom-up parsing is shift-reduce parsing
- Starts at the bottom (leaves, terminals) and works its way up to the top (root, start symbol)
- Each step is a "reduction":
  - Substring of input matching the right side of a production is "reduced"
  - Replaced with the nonterminal on the left of the production
- If all substrings are chosen correctly, a rightmost derivation is traced in reverse

Shift-Reduce Parsing Example

```
S → aABe
A → Abc | b
B -> d
```

```
abbcde
aAbcde
aAde
aABe
S
```

```
S rm=> aABe rm=>aAde rm=>aAbcde rm=> abbcde
```

17) Define Handle.

Handles

- Informally, a "**handle**" of a string:
  - Is a substring of the string
  - Matches the right side of a production
  - Reduction to left side of production is one step along **reverse of rightmost derivation**
- Leftmost substring matching right side of production is not necessarily a handle
  - Might not be able to reduce resulting string to start symbol
  - In example from previous slide, if reduce aAbcde to aAAcde, can not reduce this to S
- **Formally, a handle of a right-sentential form γ:**
  - Is a production A ◊ β and a position of γ where β may be found and replaced with A
  - Replacing A by β leads to the previous right-sentential form in a rightmost derivation of γ
- So if S rm*=> αAw rm=> αβw then A ◊ β in the position following α is a handle of αβw
- The string w to the right of the handle contains only terminals
- Can be more than one handle if grammar is ambiguous (more than one rightmost derivation)

18) What is handle pruning?

- Repeat the following process, starting from string of tokens until obtain start symbol:
  - Locate handle in current right-sentential form
  - Replace handle with left side of appropriate production
- Two problems that need to be solved:
  - How to locate handle
  - How to choose appropriate production

19) Explain stack implementation of shift reduce parsing.

Shift-Reduce Parsing

- Data structures include a stack and an input buffer
  - Stack holds grammar symbols and starts off empty
  - Input buffer holds the string w to be parsed
- Parser shifts input symbols onto stack until a handle β is on top of the stack
  - Handle is reduced to the left side of appropriate production
  - If stack contains only start symbol and input is empty, this indicates success

Actions of a Shift-Reduce Parser

- **Shift** – the next input symbol is shifted onto the top of the stack
- **Reduce** – The parser reduces the handle at the top of the stack to a nonterminal (the left side of the appropriate production)
- **Accept** – The parser announces success
- **Error** – The parser discovers a syntax error and calls a recovery routine

Shift Reduce Parsing Example

Stack	Input	Action
$	id1 + id2 * id3$	shift
$id1	+ id2 * id3$	reduce by E → id
$E	+ id2 * id3$	shift
$E +	id2 * id3$	shift
$E + id2	* id3$	reduce by E → id
$E + E	* id3$	shift
$E + E *	id3$	shift
$E + E * id3	$	reduce by E → id
$E + E * E	$	reduce by E → E * E
$E + E	$	reduce by E → E + E
$E	$	accept

20) What are viable prefixes?

Viable Prefixes

- Two definitions of a viable prefix:
  - A prefix of a right sentential form that can appear on a stack during shift-reduce parsing
  - A prefix of a right-sentential form that does not continue past the right end of the rightmost handle
- Can always add tokens to the end of a viable prefix to obtain a right-sentential form

21) Explain conflicts in shift reduce parsing.

Conflicts in Shift-Reduce Parsing

- There are grammars for which shift-reduce parsing can not be used

- **Shift/reduce conflict**: can not decide whether to shift or reduce
- **Reduce/reduce conflict**: can not decide which of multiple possible reductions to make
- Sometimes can add rule to adapt for use with ambiguous grammar

22) What is operator precedence parsing?

Operator-Precedence Parsing

- A form of shift-reduce parsing that can apply to certain simple grammars
  - No productions can have right side ε
  - No right side can have two adjacent nonterminals
  - Other essential requirements must be met
- Once the parser is built (often by hand), the grammar can be effectively ignored

23) What are precedence relations?

Precedence Relations

Relation	Meaning
a <· b	**a "yields precedence to" b**
a ·= b	**a "has the same precedence as" b**
a ·> b	**a "takes precedence over" b**

24) How precedence relations are used?

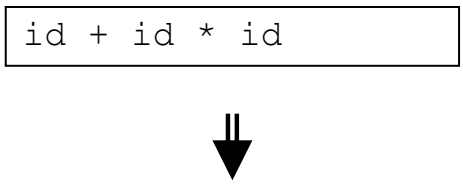
Using Precedence Relations (1)

- Can be thought of as delimiting handles:
  - <· Marks left end of handle
  - ·> Appears in the interior of handle
  - ·= Marks right end of handle
- Consider right-sentential β0a1β1β1…anBn:
  - Each βi is either single nonterminal or ε
  - Each ai is a single token
  - Suppose that exactly one precedence relation will hold for each ai, ai+1 pair

Using Precedence Relations (2)

- Mark beginning and end of string with $
- Remove the nonterminals
- Insert correct precedence relation between each pair of terminals

	i		*	$
i		.	.	.
+	<	.	<	.
*	<	.	.	.
$	<	<	<	



Using Precedence Relations (3)

- To find the current handle:
  - Scan the string from the left until the first ·> is encountered
  - Scan backwards (left) from there until a <· is encountered
  - Everything in between, including intervening or surrounding nonterminals, is the handle
- The nonterminals do not influence the parse!

25) Explain operator precedence parsing algorithm.

```
set ip to point to the first symbol in w$
initialize stack to $
repeat forever
 if $ on top of stack in ip points to $
  return success
 else
  let a be topmost symbol on stack
  let b be symbol pointed to by ip
  if a <· b or a ·= b
   push b onto stack
   advance ip to next input symbol
  else if a ·> b
   repeat
    pop x
   until top symbol on stack <· x
  else
   error()
```

26) Explain a heuristic to produce a proper set of precedence relations.

Precedence and Associativity (1)

- For grammars describing arithmetic expressions:
  - Can construct table of operator-precedence relations automatically
  - Heuristic based on precedence and associativity of operators
- Selects proper handles, even if grammar is ambiguous

The following rules are designed to select the proper handles to reflect a given set of associativity and precedence rules for binary operators :

- If operator θ1 has higher precedence than operator θ2, make θ1 ·> θ2 and θ2 <· θ1

- If θ1 and θ2 are of equal precedence:
  - If they are left associative, make θ1 ·> θ2 and θ2 ·> θ1
  - If they are right associative, make θ1 <· θ2 and θ2 <· θ1

27) What is an operator grammar?

A grammar having the property (among other essential requirements) that no production right side is ε or has two adjacent nonterminals is called an **operator grammar**.

Operator Grammar Example

E → E + E | E – E | E * E | E / E
| E ^ E | (E) | -E | id

Where

- ^ is of highest precedence and is right-associative
- * and / are of next highest precedence and are left-associative
- + and – are of lowest precedence and are left-associative

28) What are precedence functions?

Precedence Functions (1)

- Do not need to store entire table of precedence relations
- Select two precedence functions f and g :
  - f(a) < g(b) whenever a <· b
  - f(a) = g(b) whenever a ·= b
  - f(a) > g(b) whenever a ·> b

	+	-	*	/	^	(	)	i	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3		5	5	0	5	0

29) How precedence functions are constructed?

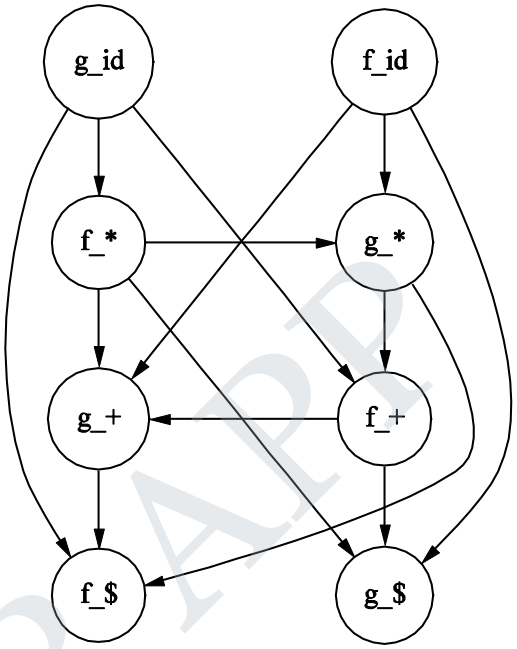
Precedence Functions Algorithm

- Create symbols fa and fg for all tokens and $
- If a ·= b then fa and gb must be in same group
- Partition symbols into as many groups as possible
- For all cases where a <· b, draw edge from group of gb to group of fa
- For all cases where a ·> b, draw edge from group of fa to group of gb
- If graph has cycles, no precedence functions exist
- Otherwise:
  - f(a) is the length of the longest path beginning at group of fa
  - g(a) is the length of the longest path beginning at group of ga

Precedence Functions Example

	i	+	*	$
i		·	·	·
+	<	·	<	·
*	<	·	·	·
$	<	<	<	

	+	*	i	$
f	2	4	4	0
g	1	3	5	0



30) How error recovery is enforced in operator precedence parsers?

Detecting and Handling Errors

- Errors can occur at two points:
  - If no precedence relation holds between the terminal on top of stack and current input
  - If a handle has been found, but no production is found with this handle as right side
- Errors during reductions can be handled with diagnostic message
- Errors due to lack of precedence relation can be handled by recovery routines specified in table

31) What are LR parsers?

LR Parsers

- LR Parsers us an efficient, bottom-up parsing technique useful for a large class of CFGs
- Too difficult to construct by hand, but automatic generators to create them exist (e.g. Yacc)
- LR(k) grammars
  - "L" refers to left-to-right scanning of input

- "R" refers to rightmost derivation (produced in reverse order)
- "k" refers to the number of lookahead symbols needed for decisions (if omitted, assumed to be 1)

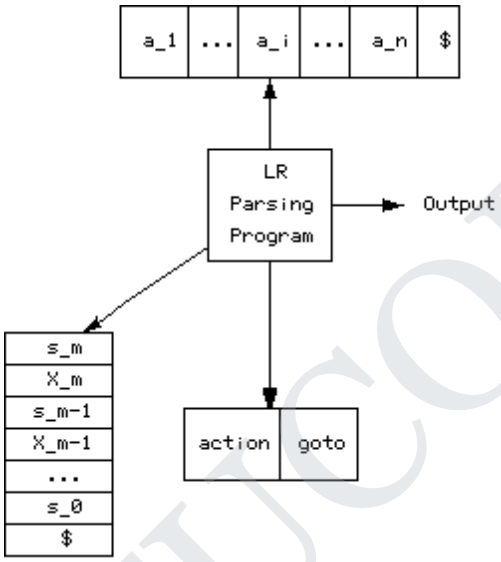
32) What are the benefits of LR parsers?

Benefits of LR Parsing

- Can be constructed to recognize virtually all programming language construct for which a CFG can be written
- Most general non-backtracking shift-reduce parsing method known
- Can be implemented efficiently
- Handles a class of grammars that is a superset of those handled by predictive parsing
- Can detect syntactic errors as soon as possible with a left-to-right scan of input

33) Explain LR parsing Algorithm with diagrams. For a given grammar and parsing table Tabulate the moves of LR parser for a given input string id * id + id.

Model of LR Parser



The LR parsing program works as follows :

The schematic of LR parser consists of an input,an output,a stack,a driver program,a parsing table that has two parts (actions and goto)

- Driver program is the same for all LR Parsers
- Stack consists of states (si) and grammar symbols (Xi)
  - Each state summarizes information contained in stack below it
  - Grammar symbols do not actually need to be stored on stack in most implementations
- State symbol on top of stack and next input symbol used to determine shift/reduce decision
- Parsing table includes action function and goto function
- **Action function**
  - Based on state and next input symbol
  - Actions are shift, reduce, accept or error
- **Goto function**
  - Based on state and grammar symbol
  - Produces next state
- Configuration (s0X1s1…Xm sm,ai ai+1…an$) indicates right-sentential form X1X2…Xmaiai+1…an

- If action[sm,ai] = shift s, enter configuration (s0X1s1…Xmsmais,ai+1…an$)
- If action[sm,ai] = reduce A ◊ B, enter configuration (s0X1s1…Xm-rsm-rAs, ai+1…an$), where s = goto[sm-r,A] and r is length of B
- If action[sm,ai] = accept, signal success
- If action[sm,ai] = error, try error recovery

LR Parsing Algorithm

set ip to point to the first symbol in w$
initialize stack to s0
repeat forever
  let s be topmost state on stack
  let a be symbol pointed to by ip
  if action[s,a] = shift s'
    push a then s' onto stack
    advance ip to next input symbol
  else if action[s,a] = reduce A → B
    pop 2*|B| symbols of stack
    let s' be state now on top of stack
    push A then goto[s',A] onto stack
    output production A → B
  else if action[s,a] == accept
    return success
  else
    error()

LR Parsing Table Example

```
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id
```

state	action						goto		
	i	+	*	(	)	$	E	T	F
0	s			s			1	2	3
1		s				a			
2		r	s		r	r			
3		r	r		r	r			
4	s			s			8	2	3
5		r	r		r	r			
6	s			s				9	3
7									1
8		s			s				

LR Parsing Example
Moves of LR parser on id * id + id

Stack	Input	Action
(1) s0	id * id + id $	shift
(2) s0 id s5	* id + id $	reduce by F → id
(3) s0 F s3	* id + id $	reduce by T → F
(4) s0 T s2	* id + id $	shift
(5) s0 T s2 * s7	id + id $	shift
(6) s0 T s2 * s7 id s5	+ id $	reduce by F → id
(7) s0 T s2 * s7 F s10	+ id $	reduce by T → T * F
(8) S0 T s2	+ id $	reduce by E → T
(9) s0 E s1	+ id $	shift
(10) s0 E s1 + s6	id $	shift
(11) s0 E s1 + s6 id s5	$	reduce by F → id
(12) s0 E s1 + s6 F s3	$	reduce by T → F
(13) s0 E s1 + s6 T s9	$	reduce by E → E + T
(14) s0 E s1	$	accept

34) What are three types of LR parsers?

Three methods:

a. **SLR (simple LR)**
   i. Not all that simple (but simpler than other two)!
   ii. Weakest of three methods, easiest to implement

b. **Constructing canonical LR parsing tables**
   i. Most general of methods
   ii. Constructed tables can be quite large

c. **LALR parsing table (lookahead LR)**
   i. Tables smaller than canonical LR
   ii. Most programming language constructs can be handled

35) Explain the non-recursive implementation of predictive parsers.

Non-recursive implementation:

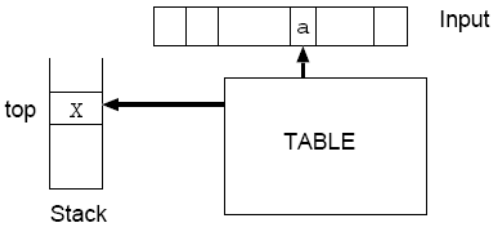


Table: 2-d array s.t. $M[A, a]$ specifies A-production to be used if input symbol is a

Algorithm:

0. Initially: stack contains $\langle$EOF $S\rangle$, input pointer is at start of input
1. if $X = a =$ EOF, done
2. if $X = a \neq$ EOF, pop stack and advance input pointer
3. if X is non-terminal, lookup $M[X, a] \Rightarrow X \rightarrow UVW$ pop X, push W, V, U

FIRST(α): set of terminals that begin strings derived from α

if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then $\epsilon \in FIRST(\alpha)$

FOLLOW (A): set of terminals that can appear immediately to the right of A in some sentential form

$$FOLLOW(A) = \{a \mid S \stackrel{*}{\Rightarrow} \alpha A a \beta\}$$

if A is the rightmost symbol in any sentential form, then EOF $\in FOLLOW(A)$

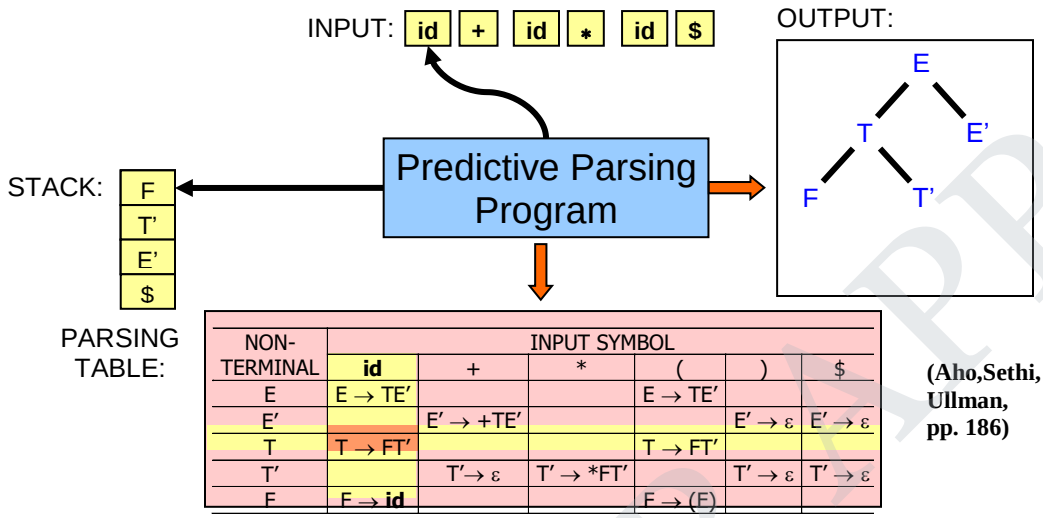
FIRST:

1. if X is a terminal, then $FIRST(X) = \{X\}$
2. if $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$
3. if $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production:
   if $a \in FIRST(Y_i)$ and $\epsilon \in FIRST(Y_1), \ldots, FIRST(Y_{i-1})$, add a to $FIRST(X)$
   if $\epsilon \in FIRST(Y_i)\ \forall i$, add ϵ to $FIRST(X)$

FOLLOW:

1. Add EOF to $FOLLOW(S)$
2. For each production of the form $A \rightarrow \alpha B \beta$
   (i) add $FIRST(\beta) \backslash \{\epsilon\}$ to $FOLLOW(B)$
   (ii) if $\beta = \epsilon$ or $\epsilon \in FIRST(\beta)$, then add everything in $FOLLOW(A)$ to $FOLLOW(B)$

A Predictive Parser



NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	E → TE′			E → TE′		
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′			T → FT′		
T′		T′→ ε	T′ → *FT′		T′ → ε	T′ → ε
F	F → id			F → (F)		

(Aho,Sethi, Ullman, pp. 186)

36) What are the advantages of using an intermediate language?

Advantages of Using an Intermediate Language

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

2. ***Optimization*** - reuse intermediate code optimizers in compilers for different languages and different machines.

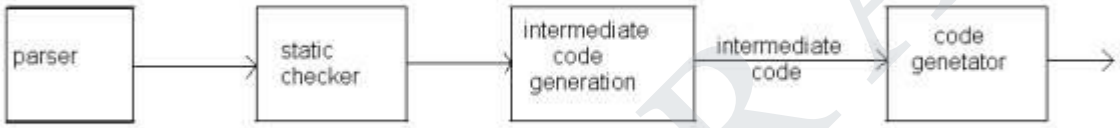
Note: the terms "intermediate code", "intermediate language", and "intermediate representation" are all used interchangeably.

Anna University – 2017 Regulations
B.E. (COMPUTER SCIENCE AND ENGINEERING)
VI SEM CSE
CS8602 Compiler Design Question and answers
UNITS III Notes

---

1) **What is intermediate code?**

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



intermediate code is:

- the output of the Parser and the input to the Code Generator.
- relatively machine-independent: allows the compiler to be *retargeted*.
- relatively easy to manipulate (optimize).

2) What are the advantages of using an intermediate language?

Advantages of Using an Intermediate Language
Retargeting - Build a compiler for a new machine by attaching a new code generator to an existing front-end.
2. ***Optimization*** - reuse intermediate code optimizers in compilers for different languages and different machines.

Note: the terms “intermediate code”, “intermediate language”, and “intermediate representation” are all used interchangeably.

3) What are the types of intermediate representations?

There are three types of intermediate representation:-

1. Syntax Trees

2. Postfix notation

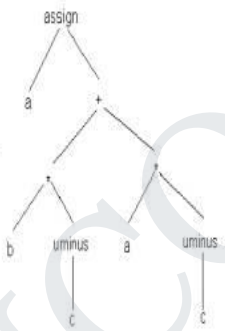
3. Three Address Code

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

Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified. A syntax tree for

the assignment statement a:=b*-c+b*-c appear in the figure.



.

fig8.2

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

a b c uminus + b c uminus * + assign

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

4) Construct a syntax directed definition for constructing a syntax tree for assignment statements.

Syntax tree for assignment statements are produced by the syntax directed definition in fig.

Production	Semantic Rule
S → **id** := E	S.nptr := mknode('assign', mkleaf(**id**, **id**.place), E.nptr)
E → E1 + E2	E.nptr := mknode('+', E1.nptr ,E2.nptr)
E → E1 * E2	E.nptr := mknode('* ', E1.nptr ,E2.nptr)
E → - E1	E.nptr := mkunode('uminus', E1.nptr)
E → (E1)	E.nptr := E1.nptr
E → **id**	E.nptr := mkleaf(**id**, **id**.place)

5) How a syntax tree is represented?

This same syntax-directed definition will produce the dag if the functions mkunode(op, child) and mknode(op, left, right) return a pointer to an existing node whenever possible, instead of constructing new nodes. The token id has an attribute place that points to the symbol-table entry for the identifier id.name, representing the lexeme associated with that occurrence of id. lf the lexical analyzer holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme. Two representations of the syntax tree in Fig8.2 appear in Fig.8.4. Each node is represented as a record with a field for its operator and additional fields for pointers to its children. In Fig 8.4(b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position IO.

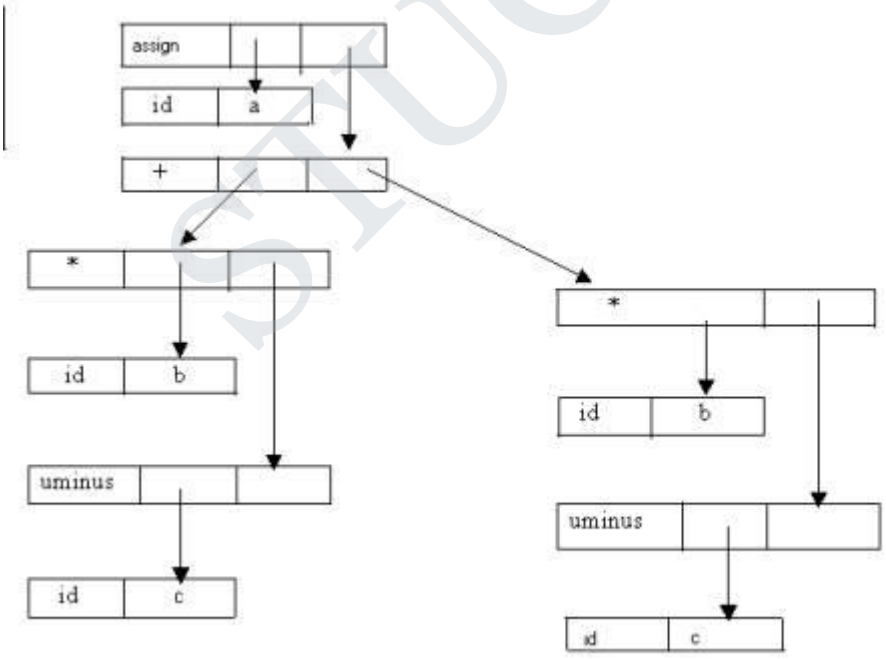


fig8.4(a)

0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	……		

fig8.4(b)

6) What is three address code?

Three-address code is a sequence of statements of the general form

X:= Y Op Z

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

t1 := y * z

t2 : = x + t1

where t1 and t2 are compiler-generated temporary names. This unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization. The use of names for the intermediate values computed by a program allow- three-address code to be easily rearranged – unlike postfix notation. three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag in Fig. 8.2 are represented by the three-address code sequences in Fig. 8.5. Variable names can appear directly in three-address statements, so Fig. 8.5(a) has no statements corresponding to the leaves in Fig. 8.4.

Code for syntax tree

t1 := -c

t2 := b * t1

t3 := -c

t4 := b * t3

t5 := t2 + t4

a := t5

Code for DAG

t1 := -c

t2 := b * t1

t5 := t2 + t2

a := t5

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

7) What are the types of three address statements?

Types Of Three-Address Statements

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

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

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

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

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

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

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

param x1

param x2

param xn

call p, n

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

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

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

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

8) Explain the process of syntax directed translation of three address code.

Syntax-Directed Translation into Three-Address Code

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of non-terminal E on the left side of E → E1 + E will be computed into a new temporary t. In general, the three- address code for id: = E consists of code to evaluate E into some

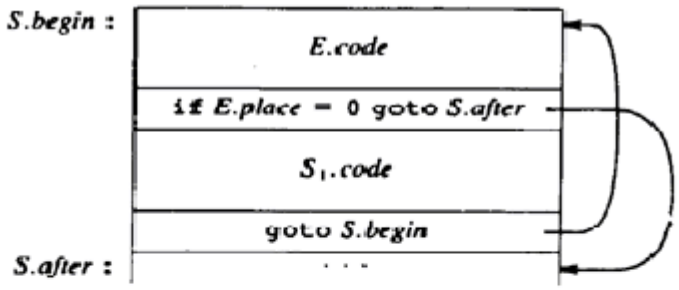
temporary t, followed by the assignment id.place: = t. If an expression is a single identifier, say y, then y itself holds the value of the expression. For the moment, we create a new name every time a temporary is needed; techniques for reusing temporaries are given in Section S.3. The S-attributed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

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

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

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

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



PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **while** E **do** S_1	*S.begin* := *newlabel*; *S.after* := *newlabel*; *S.code* := *gen*(*S.begin* ':') ‖ *E.code* ‖ *gen*('if' *E.place* '=' '0' 'goto' *S.after*) ‖ S_1.*code* ‖ *gen*('goto' *S.begin*) ‖ *gen*(*S.after* ':')

These attributes represent labels created by a function new label that returns a new label every time it is called. Note that S.after becomes the label of the statement that comes after the code for the while statement. We assume that a non-zero expression represents true; that is, when the value of F becomes zero, control leaves the while statement. f:expressions that govern the flow of control may in general be Boolean expressions containing relational and logical operators. The semantic rules for while statements in Section 8.6 differ from those in Fig. 8.7 to allow for flow of contro1 within Boolean expressions. Postfix notation -an be obtained by adapting the semantic rules in Fig. 8.6 (or see Fig. 2.5). 1he postfix notation for an identifier is the identifier itself. The rules for the other productions concatenate only the operator after the code for the operands. For example, associated with the production E – E, is the semantic rule

E.code:= E1.code || 'uminus'

1n general, the intermediate form produced by the syntax-directed translations in this chapter can he changed by making similar modifications to the semantic rules.

9) Explain in detail the implementation of three address statements.

Implementations of three-Address Statements

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

Quadruples

A quadruple is a record structure with four fields, which we call op, arg l, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unary operators like x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result. The quadruples in Fig. H.S(a) are for the assignment a: = b+ – c + b i – c. They are obtained from the three-address code in Fig. 8.5(a). The contents of fields arg 1, arg 2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples

To avoid entering temporary names into the symbol table. we might refer to a temporary value bi the position of the statement that computes it. If we do so, three-address statements can be represented by records with only three fields: op, arg 1 and arg2, as in Fig. 8.8(b). The fields arg l and arg2, for the arguments of op, are either pointers to the symbol table (for programmer- defined names or constants) or pointers into the triple structure (for temporary values). Since three fields are used, this intermediate code format is known as triples.' Except for the treatment of programmer-defined names, triples correspond to the representation of a syntax tree or dag by an array of nodes, as in Fig. 8.4.

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

fig8.8(a) Qudraples

	op	Arg1	Arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

fig8.8(b) Triples

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves. In practice, the information needed to interpret the different kinds of entries in the arg 1 and arg2 fields can be encoded into the op field or some additional fields. The triples in Fig. 8.8(b) correspond to the quadruples in Fig. 8.8(a). Note that the copy statement a:= t5 is encoded in the triple representation by placing a in the arg 1 field and using the operator assign. A ternary operation like x[i]: = y requires two entries in the triple structure, as shown in Fig. 8.9(a), while x: = y[i] is naturally represented as two operations in Fig. 8.9(b).

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

(a) x[i] := y

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

(b) x := y[i]

Fig. 8.9. More triple representations.

Indirect Triples

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

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

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

Fig. 8.10. Indirect triples representation of three-address statements.

10) How declarations are translated into intermediate code?

DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record. When the front end generates addresses, it may have a target machine in mind. Suppose that addresses of consecutive integers differ by 4 on a byte- addressable machine. The address calculations generated by the front end may therefore include multiplications by 4. The instruction set of the target machine may also favor certain layouts of data objects, and hence their addresses. We

ignore alignment of data objects here, Example 7.3 shows how data objects are aligned by two compilers.

Declarations in a Procedure

The syntax of languages such as C, Pascal, and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say offset, can keep track of the next avai1able relative address. In the translation scheme of Fig. S.I1 non-terminal P generates a sequence of declarations of the form id: T. Before '.he first declaration is considered, offset is set to 0. As each new name is seen, that name is entered in the symbol table with offset equal to the current value of offset, and offset is incremented by the width of the data object denoted by that name. The procedure enter(name, type, offset) creates a symbol-table entry for name, gives it type and relative address offset in its data area. We use synthesized attributes type and width for non-terminal T to indicate the type and width, or number of memory units taken by objects of that type. Attribute type represents a type expression constructed from the basic types integer and real by applying the type constructors pointer and array, as in Section 6.l. If type expressions are represented by graphs, then attribute type might be a pointer to the node representing a type expression. In Fig. 8. I , integers have width 4 and real have width 8. The width of an array is obtained by multiplying the width of each element by the number of elements in the array.- The width of each pointer is assumed to be 4.

P → D

D → D ; D

D → id : T {enter (id.name, T.type, offset);

Offset:= offset + T.width }

T → integer {T.type :=integer;

T.width :=4}

T → real {T.type := real;

T.width := 8}

T → array [num] of T1 {T.type :=array(num.val, T1.type);

T.width :=num.val X T1.width}

T → ^T1 {T.type :=pointer (T.type);

T.width:=4}

In Pascal and C, a pointer may be seen before we learn the type of the object pointed to Storage allocation for such types is simpler if all pointers have the same width. The initialization of offset in the translation scheme of Fig. 8.1 is more evident if the first production appears on one line as:

P →{offset:= 0 } D

Non-terminals generating a. called marker non-terminals in Section 5.6, can be used to rewrite productions so that all actions appear at the ends of right sides. Using a marker non-terminal M, (8.2) can be restated as:

P → M D

M →ε (offset:= 0}

Keeping Track of Scope Information

In a language with nested procedures, names local to each procedure can be assigned relative addresses using the approach of Fig. 8.11 . When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will he illustrated by adding semantic rules to the following language.

P → D

D → D;D | id: T proc id; D;S

The production for non-terminals S for statements and T for types are not shown because we focus on declarations. The non-terminal T has synthesized attributes type and width, as in the translation scheme of Fig. For simplicity, suppose that there is a separate symbol table for each procedure in the language (8.3). One possible implementation of a symbol table is a linked list of entries for names. Clever implementations can be substituted if desired. A new symbol table is created when a procedure declaration D proc id D~; 5 is seen, and entries for the declarations in D~ are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations in Fig. 8.11 is that the procedure enter is told which symbol table to make an entry in. For example, symbol tables for five procedures are shown in Fig. 8. l2. The nesting structure of the procedures can be deduced from the links between the symbol tables; the program is in Fig. 7.22. The symbol tables for procedures readarray, exchange, and quicksort point back to that for the containing procedure sort, consisting of the entire program. Since partition is declared within quicksort, its table points to that of quick sort.

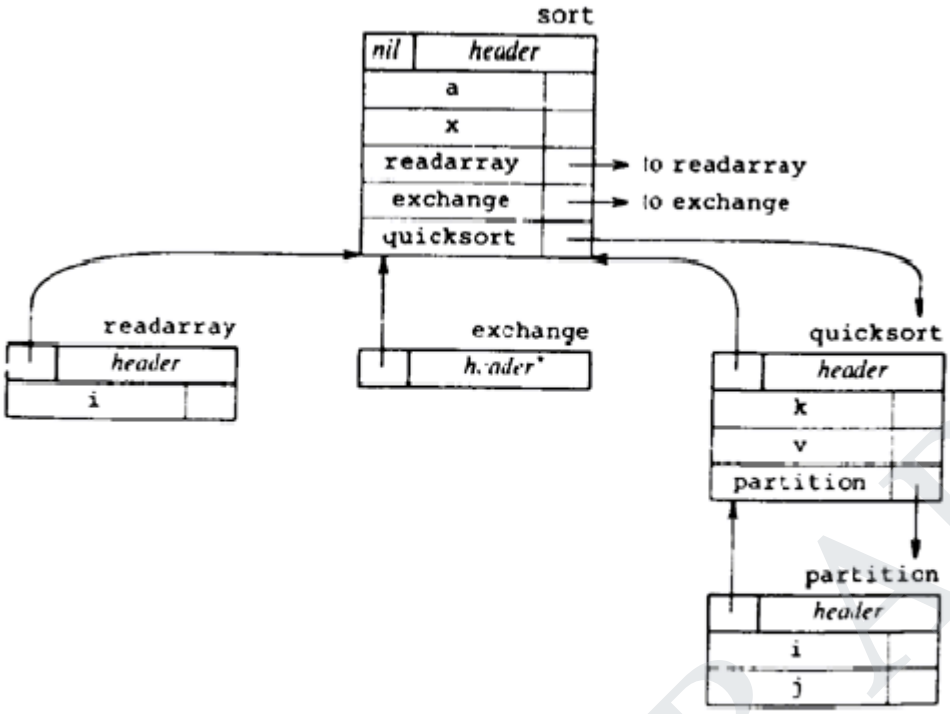


Fig. 8.12. Symbol tables for nested procedures

The semantic rules are defined in terms of the following operations:

I. mktable(previous) creates a new symbol table and returns a pointer to the new table. The argument previous points to a previously created symbol table, presumably that for the enclosing procedure. The pointer previous is placed in a header for the new symbol table, along with additional information such as the nesting depth of a procedure. We can also number the procedures in the order they are declared and keep this number in the header.

2. enter(table, name, type, offset) creates a new entry for name *name* in the symbol table pointed to by table. Again, enter places type and relative address offset in fields within the entry.

3. addwidth(table, width) records the cumulative width of all the entries table in the header associated with this symbol table.

4. enterproc (table, name, newtable) creates a new entry for procedure name in the symbol table pointed to by table. The argument newtable points to the symbol table for this procedure name.

The translation scheme in Fig. S. l3 shows how data can be laid out in one pass, using a stack tblptr to hold pointers to symbol tables of the enclosing procedures. With the symbol tables ot' Fig. 8.12, tblptr will contain pointers to the tables for -ort, quicksort, and partition when the declarations in partition are considered. The pointer to the current symbol table is on top. The other stack offset is the natural generalization to nested procedures of attribute offset in Fig. 8. l I. The top element of offset is the next available relative address for a local of the current procedure. All semantic actions in the sub-trees for B and C in

A B C { actionA}

are done before actionA the end of the production occurs. Hence, the action associated with the marker M in Fig. 8.l3 is the first to be done. The action for non-terminal M initializes stack tblptr with a symbol table for the outermost scope, created by operation mktable(nil). The action also pushes relative address 0 onto stack offset. The non-terminal V plays a similar role when a procedure declaration appears. Its action uses the operation mktable(top(tblptr)) to create a new symbol table. Here the argument top(tblptr) gives the enclosing scope of the new table. A pointer to the new table is pushed above that for the enclosing scope. Again, 0 is pushed onto offset.

For each variable declaration id: T. an entry is created for id in the current symbol table. This declaration leaves the stack pointer unchanged; the top of stack offset is incremented by T.width. when the action on the right side of D proc id: N D,; S occurs. the width of all

declarations generated by D1 is on top of stack offset.', it is recorded using addwidth. and offset are then popped, and we revert to examining the declarations in the closing procedure. At this point, the name of the enclosed procedure is entered into the symbol table of its enclosing procedure.

P → M D {addwidth(top(tblptr), top(offset));
Pop(tblptr); pop(offset)}

M → ε { t := mktable(nil);
Push(t,tblptr); push(0,offset)}

D → D1 ;D2

D → proc id ; N D1 ;S { t := top(tblptr);
addwidth(t.top(offset));
pop(tblptr); pop(offset);
enterproc(top(tblptr), id.name, t)}

D → id : T {enter(top(tblptr),id.name,T.type,top(offset));

top(offset) := top(offset) +T.width }

N → ε { t := mktable(top(tblptr));

Push(t, tblptr); push(0, offset)}

Field Names in Records

The following production allows non-terminal T to generate records in addition to basic types, pointers, and arrays:

T → record D end

The actions in the translation scheme of Fig. S.I4 emphasize the similarity between the layout of records as a language construct and activation records. Since procedure definitions do not affect the width computations in Fig. 8.13, we overlook the fact that the above production also allows procedure definitions to appear within records.

T → record L D end {T.type := record(top(tblptr));

T.width := top(offset);

Pop(tblptr); pop(offset) }

L→ ε { t:= mktable(nil);

Push(t, tblptr); push (0, offset) }

After the keyword record is seen, the acting associated with the marker

creates a new symbol table for the field names. A pointer to this symbol table is pushed onto stack tblptr and relative address 0 is pushed onto stack . The action for D → id: T in Fig. 8.13 therefore enters information about the field name id into the symbol table for the record. Furthermore, the top of stack will hold the width of all the data objects within the record after the fields have been examined. The action following end in Fig. 8. 14 returns the width as synthesized attribute T.width. The type T.type is obtained by applying the constructor *record* to the pointer to the symbol table for this record.

Anna University – 2017 Regulations
B.E. (COMPUTER SCIENCE AND ENGINEERING)
VI SEM CSE
CS8602 Compiler Design Question and answers
UNITS IV Notes

UNIT IV CODE GENERATION 9

Issues in the design of code generator – The target machine – Runtime Storage management – Basic Blocks and Flow Graphs – Next-use Information – A simple Code generator – DAG representation of Basic Blocks – Peephole Optimization.

1) What is the role of code generator in a compiler?

CODE GENERATION

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

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

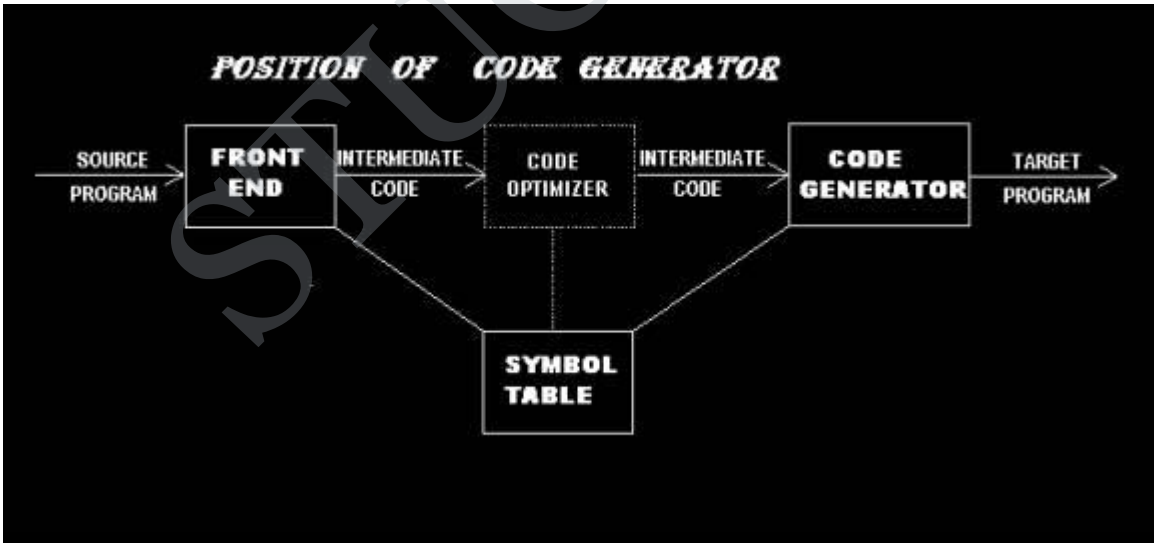


fig. 1

2) Write in detail the issues in the design of code generator.

ISSUES IN THE DESIGN OF A CODE GENERATOR

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

INPUT TO THE CODE GENERATOR

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

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

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

TARGET PROGRAMS

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

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

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

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

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

MEMORY MANAGEMENT

Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator. We assume that a name in a three-address statement refers to a symbol table entry for the name.

If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the “back patching”. Suppose that labels refer to quadruple numbers in a quadruple array. As we scan each quadruple in turn we can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as j: *goto i* is encountered, and i is less than j, the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple i. If, however, the jump is forward, so i exceeds j, we must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. Then we process quadruple i, we fill in the proper machine location for all instructions that are forward jumps to i.

INSTRUCTION SELECTION

The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three- address statement we can design a code skeleton that outlines the target code to be generated for that construct.

For example, every three address statement of the form x := y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
MOV y, R0   /* load y into register R0  */

ADD z, R0    /* add z to R0 */

MOV R0, x   /* store R0 into x */
```

Unfortunately, this kind of statement – by - statement code generation often produces poor code. For example, the sequence of statements

```
a := b + c

d := a + e
```

would be translated into

```
MOV   b, R0

ADD    c, R0

MOV   R0, a

MOV   a, R0

ADD    e, R0

MOV   R0, d
```

Here the fourth statement is redundant, and so is the third if 'a' is not subsequently used.

The quality of the generated code is determined by its speed and size.

A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost differences between different implementations may be significant, a naive translation of the intermediate code may lead to correct, but unacceptably inefficient target code. For example if the target machine has an "increment" instruction (INC), then the three address statement a := a+1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, add one to the register, and then stores the result back into a.

```
MOV   a, R0

ADD    #1,R0

MOV   R0, a
```

Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain. Deciding which machine code sequence is best for a

given three address construct may also require knowledge about the context in which that construct appears.

REGISTER ALLOCATION

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

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

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

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

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

```
M    x, y
```

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

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

The division instruction is of the form

```
D    x, y
```

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

Now consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c).

Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which ‘a’ is to be loaded depends on what will ultimately happen to e.

```
t  :=  a + b          t := a + b

t  :=  t * c          t  :=  t + c
```

```
t := t / d                    t := t / d
```

(a) (b)

fig. 2 Two three address code sequences

```
L   R1, a                     L    R0, a
A   R1, b                     A    R0, b
M   R0, c                     A    R0, c
D   R0, d                     SRDA R0, 32
ST  R1, t                     D    R0, d
                              ST   R1, t
```

(a) **(b)**

fig.3 Optimal machine code sequence

CHOICE OF EVALUATION ORDER

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

—

APPROCHES TO CODE GENERATION

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator

must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

3) What are basic blocks and flowgraphs?

BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

BASIC BLOCKS

A **basic block** is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. The following sequence of three-address statements forms a basic block:

t1 := a*a

t2 := a*b

t3 := 2*t2

t4 := t1+t3

t5 := b*b

t6 := t4+t5

A three-address statement x := y+z is said to *define* x and to *use* y or z. A name in a basic block is said to *live* at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 1: Partition into basic blocks.

Input: A sequence of three-address statements.

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the set of **leaders**, the first statements of basic blocks.

The rules we use are the following:

I) The first statement is a leader.

II) Any statement that is the target of a conditional or unconditional goto is a leader.

III) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example 3: Consider the fragment of source code shown in fig. 7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i+1;
     end
      while  i<= 20
   end
```

fig 7: program to compute dot product

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic blocks. statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block.

```
(1)  prod := 0
(2)  i := 1
(3)  t1 := 4*i
(4)  t2 := a [ t1 ]
```

(5) t3 := 4*i

(6) t4 :=b [t3]

(7) t5 := t2*t4

(8) t6 := prod +t5

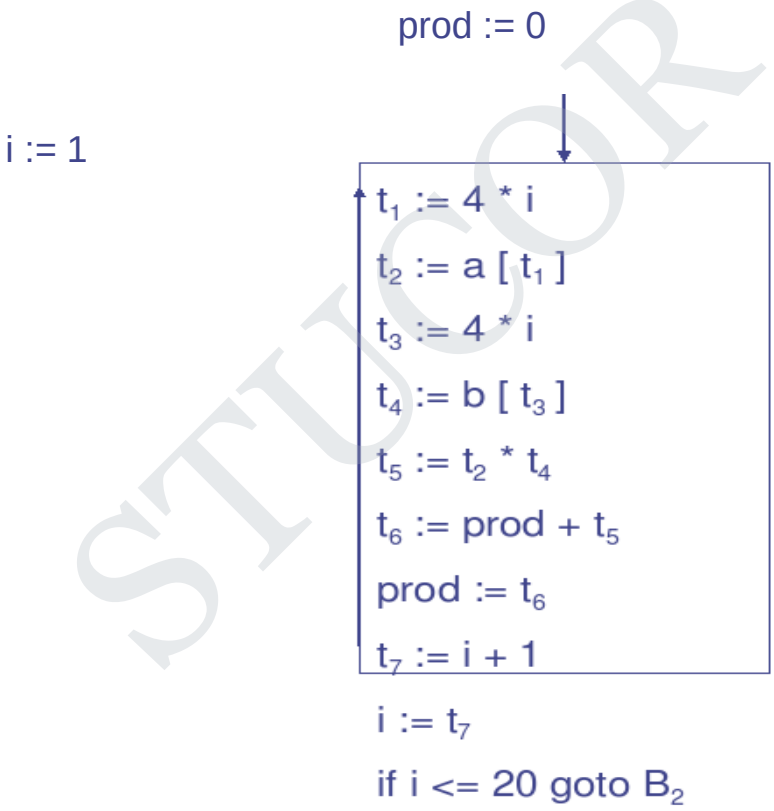
(9) prod := t6

(10) t7 := i+1

(11) i := t7

(12) if i<=20 goto (3)

fig 8. Three-address code computing dot product



4) What are the structure preserving transformations on basic blocks?

TRANSFORMATIONS ON BASIC BLOCKS

A basic block computes a set of expressions. These expressions are the values of the names live on exit from block. Two basic blocks are said to be *equivalent* if they compute the same set of expressions.

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraic transformations.

STRUCTURE-PRESERVING TRANSFORMATIONS

The primary structure-preserving transformations on basic blocks are:

1. common sub-expression elimination
2. dead-code elimination
3. renaming of temporary variables
4. interchange of two independent adjacent statements

We assume basic blocks have no arrays, pointers, or procedure calls.

1. Common sub-expression elimination

Consider the basic block

a:= b+c

b:= a-d

c:= b+c

d:= a-d

The second and fourth statements compute the same expression,

namely b+c-d, and hence this basic block may be transformed into the equivalent block

a:= b+c

b:= a-d

c:= b+c

d:= b

Although the 1st and 3rd statements in both cases appear to have the same expression on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

2. **Dead-code elimination**

Suppose x is dead, that is, never subsequently used, at the point where the statement x:= y+z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

3. **Renaming temporary variables**

Suppose we have a statement t:= b+c, where t is a temporary. If we change this statement to u:= b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed. In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary. We call such a basic block a *normal-form* block.

4. **Interchange of statements**

Suppose we have a block with the two adjacent statements

t1:= b+c

t2:= x+y

Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2. A normal-form basic block permits all statement interchanges that are possible.

5) What are the instructions and address modes of the target machine?

The target machine characteristics are

- Byte-addressable, 4 bytes/word, *n* registers
  - Two operand instructions of the form
  - op source, destination
  - Example opcodes: MOV, ADD, SUB, MULT
  - Several addressing modes
  - An instruction has an associated *cost*
    - Cost corresponds to length of instruction

Addressing Modes & Extra Costs

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	*k*(R)	*k* + *contents*(R)	1
Indirect register	*R	*contents*(R)	0
Indirect indexed	**k*(R)	*contents*(k + *contents*(R))	1

6) Generate target code for the source language statement "(a-b) + (a-c) + (a-c);"

The 3AC for this can be written as

```
t := a – b
u := a – c
v := t + u
d := v + u      //d live at the end
```

Show the code sequence generated by the simple code generation algorithm

What is its cost? Can it be improved?

Statements	Generated Code	*RegDes*	*AdDes*
		Registers empty	
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Total cost =12

activation record?

Information needed by a single execution of procedure is managed using a contiguous block of storage called an activation record or frame. It is customary to push the activation record of a procedure on the run time stack when the procedure is called and to pop the activation record off the stack when control returns to the caller.

8) What are the contents of activation record?

Contents of A.R.

- Returned value for a function.
- Parameters:
  - **Formal parameters:** the declaration of parameters.
  - **Actual parameters:** the values of parameters for this activation.
- Links: where variables can be found.
  - **Control (or dynamic) link:** a pointer to the activation record of the caller.
  - **Access (or static) link:** a pointer to places of non-local data,
- Saved machine status.
- Local variables.
- Temporary variables.
  - Evaluation of expressions.
  - Evaluation of arguments.
  - Evaluation of array indexes.
  - ...

Anna University – 2017 Regulations
B.E. (COMPUTER SCIENCE AND ENGINEERING)
VI SEM CSE
CS8602 Compiler Design Question and answers
UNITS V Notes

UNIT V CODE OPTIMIZATION AND RUN TIME ENVIRONMENTS 9

Introduction– Principal Sources of Optimization – Optimization of basic Blocks – Introduction to Global Data Flow Analysis – Runtime Environments – Source Language issues – Storage Organization – Storage Allocation strategies – Access to non-local names – Parameter Passing.

1) What is code optimization?

The code produced by the straight forward compiling algorithms can often be made to run faster or take less space,or both. This improvement is achieved by program transformations that are traditionally called optimizations. Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine. Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

2) List the criteria for code improvement transformations.

Simply stated ,the best program transformations are those that yield the most benefit for the least effort.

First,the transformation must preserve the meaning of programs. That is,the optimization must not change the output produced by a program for a given input,or cause an error.

Second,a transformation must,on the average,speed up programs by a measurable amount.

Third,the transformation must be worth the effort.

Some transformations can only be applied after detailed,often time-consuming analysis of the source program,so there is little point in applying them to programs that will be run only a few times.

Code Improvement Criteria

A transformation must preserve meaning of a program (correctness)

- A transformation must improve (e.g., speed-up) programs by a measurable amount on average
- A transformation must worth the effort

3) Indicate the places for potential improvements can be made by the user and the compiler.

Programmer

Profiles, change algorithm, transform loops

Compiler: (on intermediate code)

Improve loops, procedure calls, various transformations

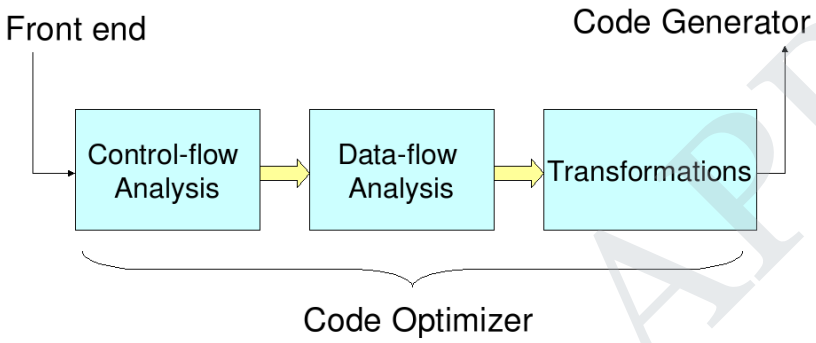
Compiler: (on target code)

Use of registers, instruction selection, peephole optimization

4) What are the phases of code improvement?

The code improvement phase consists of control-flow and data-flow analysis followed by the application of transformations

Optimizing Compiler: Organization

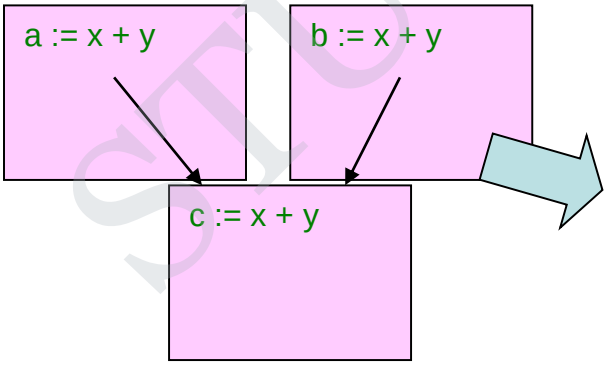


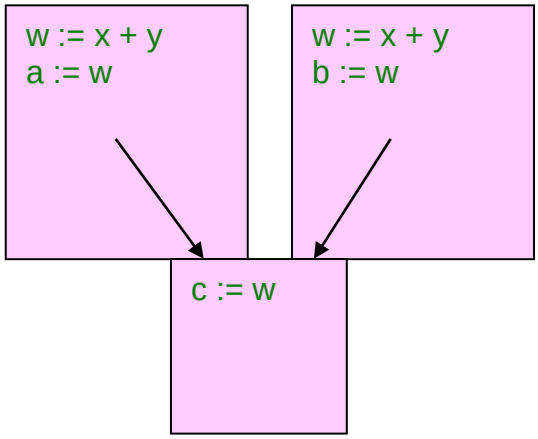
5) Discuss in detail the principal sources of optimization.

Principal Sources of Optimization

Function-preserving transformations

- Common sub-expression elimination
- Copy propagation





Dead-code elimination

- Dead (or useless) code: statements that evaluate values that never get used

- Dead-code may appear after transformations

- *Constant folding*
- Deducing at compile-time value of an expression is a constant and using the constant instead
- **Loop optimizations (especially inner loops)**

  Programs tend to spend most of their time in inner loops
- We may improve running time by *decreasing the number of instructions in an inner loop* even while *increasing the amount of code outside the loop*

Code motion

Places *loop-invariant computation* before the loop

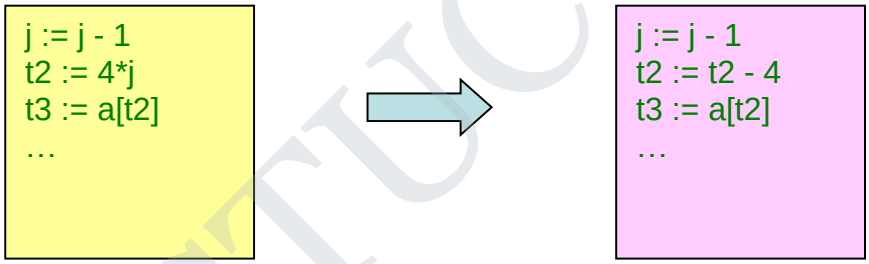
E.g.,

```
while ( i <= limit-2 )  /* limit is loop invariant */


t = limit – 2;
while ( i <= t )    /* limit, t are loop-invariant */
```

Strength reduction

E.g., The replacement of multiplication by a subtraction or addition might be possible when induction vars are modified in a loop



6) Discuss optimization of basic blocks.

Optimizing Basic Blocks

Local transformations that preserve structure

- Common sub-expression elimination
- Dead-code elimination

- **Local transformations that are algebraic**
  - Arithmetic identities; e.g., x+0 = x, x/1 = x
  - Strength reduction; e.g., x**2 = x*x, x/2=x*0.5
  - Constant folding

7) What is dataflow analysis?

An optimizing compiler needs to

- Collect info about the program as a whole
- Distribute info to each block in the flow graph
- E.g., knowing which vars are live on exit from each block and using it for register allocation
- This info is *dataflow information* and a compiler collects this by *dataflow analysis*

- Info can be collected by setting up and solving systems of equations
  - They relate info at various parts in a program
  - A *dataflow equation* is of the form:
  - $out[S] = gen[S] \cup (in[S] - kill[S])$
    - Meaning: "*information at the end of statement (or block)* S *is either generated within it, or enters at the beginning and is not killed as control flows through it* "

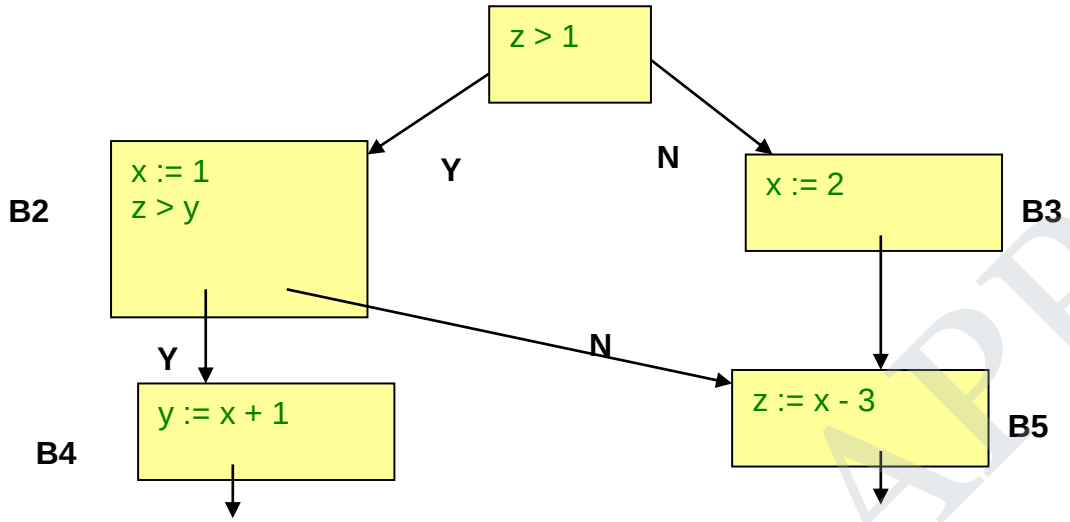
To set up and solve equations, may have to proceed forward or backwards

- i.e., backwards: define *in* [S] in terms of *out* [S]
- Equations are normally set up at basic block level

- Need to handle complexities involved in function calls and use of pointers
- Dataflow Analysis: Examples
- *Reaching Definitions (ud-chains)*
  - Determines which *definitions* of (assignments to) a variable may reach each *use* of it
- *Available Expressions*
  - Determines which expressions are *available* at each point in a program (on every path from the entry to the point, there is an evaluation of the expression and none of the vars in it are assigned values between the last evaluation and the point)
- *Liveness (Live-Variable) Analysis*
  - Determines for a given var and a given point whether there is a *use* of the var along some path from the point to the exit
- *Upwards Exposed Uses (du-chains)*
  - Determines what *uses* of vars at certain points are reached by particular *definitions* (this is the dual of reaching definitions)

Example

Reaching definitions and its ***dual***:
- *Use* of x in B5 is reached by the *definitions* in B2, B3
- *Definition* of x in B2 reaches the *uses* in B4, B5



Dataflow Analysis: Examples

Copy-Propagation Analysis

Determines that on every path from a copy assignment such as x:= y, to a use of var x there is no assignments to y

Constant-Propagation Analysis

Determines that on every path from an assignment of a constant to a variable, x := k, to a use of x the only assignment to x assign the value k

8) What is an activation tree? Explain its functions.

Every execution of a procedure is called an ACTIVATION.

We can use a tree,called an activation tree to depict the way control enters and leaves activations.

The LIFETIME of an activation of procedure P is the sequence of steps between the first and last steps of P's body, including any procedures called while P is running.

Normally, when control flows from one activation to another, it must (eventually) return to the same activation.

When activations are thusly nested, we can represent control flow with ACTIVATION TREES.

In an activation tree ,

1. *Eac*

2. *h node represents an activation of a procedure.*
3. *The root represents the activation of the main program.*
4. *The node for a is the parent of the node b if and only if control flows from activation a to b.*
5. *The node for 'a' is to the left of the node for 'b' if and only if the life time of 'a' occurs before the life time of 'b'.*

8) What are control stacks?

The flow of control in a program corresponds to a depth first traversal of the activation treethat starts at the root,visits anode before its children,and recursively visits children at each node in a left-to-right order.

We can use a stack,called a control stack to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack and to pop the node when the activation ends.

When node 'n' is at the top of the control stack,the stack contains the nodes along the path from 'n' to the root.

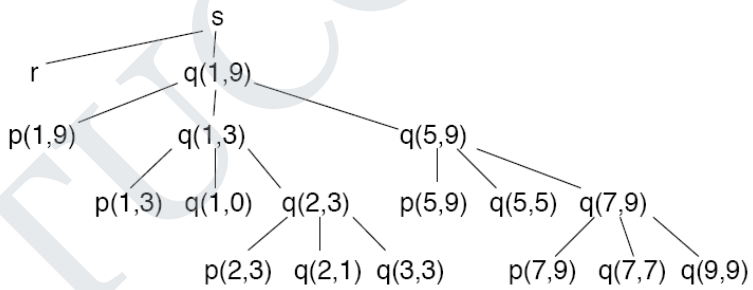
9) What are the storage allocation strategies used at Run time?

Storage Allocation

Static allocation

Storage allocation was fixed during the entire execution of a program

Stack allocation



Space is pushed and popped on a run-time stack during program execution such as procedure calls and returns.

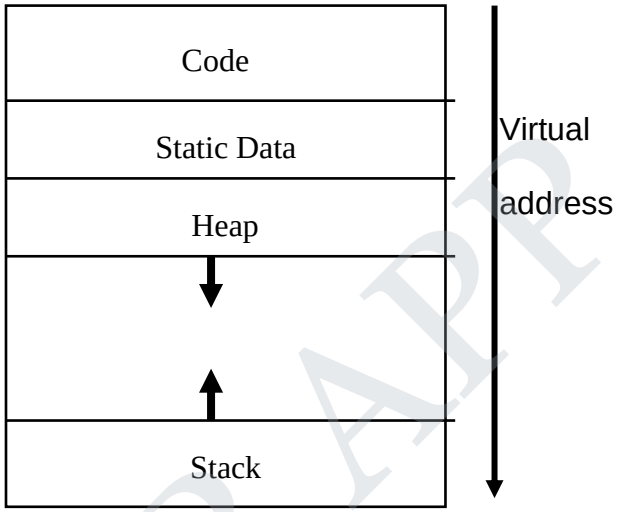
Heap allocation

Allow space to be allocated and freed at any time.

10) Explain with a diagram the run time storage organization.

Run Time Sorage Organization

Code
Static data
Stack
Heap



11) What is static allocation?

Static Allocation

.Bindings between names and storage are fixed

The values of local names are retained across activations of a procedure.

Addressing is efficient

Limitations:

No recursive procedures

No dynamic data structures

12) What is stack allocation?

Stack Allocation

.Recursive procedure require stack allocation •

- Activation records (AR) are pushed and popped as activations begin and end.
- The offset of each local data relative to the beginning of AR is stored in the symbol table.

```
float f(int k)
{
float c[10],b;
b = c[k]*3.14;
return b;
```

Return value	offset = 0
Parameter k	offset = 4

13) What are calling sequences?

Procedure calls are implemented by generating what are known as calling sequences in the target code. A call sequence allocates an activation record and enters information into its fields. A return sequence restores the state of the machine so the calling procedure can continue execution.

Calling Sequences

.A call sequence allocates an activation record and enters information into its fields

parameters, return address, old stack top, saving registers, local data initialization

A return sequence restores the state of the machine

return value, restore registers, restore old stack top, branch to return address

The code in calling sequence is often divided between the caller and the callee.

14) What are return sequences?

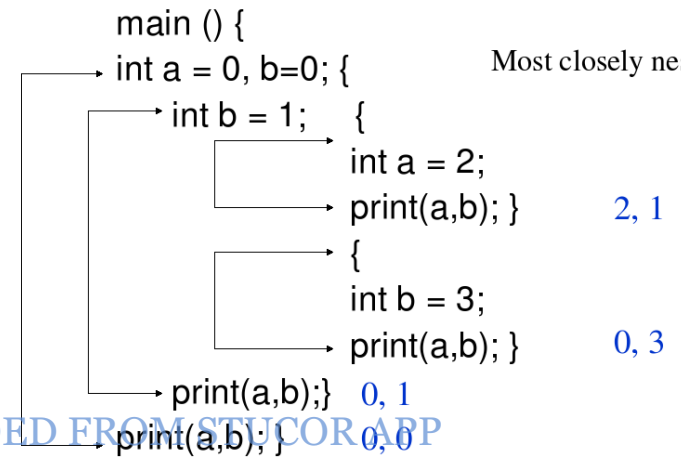
Possible return sequence

- Callee places a return value next to the AR of the caller.
- Callee restores top-sp and other registers
- Callee branches to the return address
- Caller can copy return value into its own AR

15) What are non-local names?

In a language with nested procedures (or blocks) and static scope (lexical scope), some names are neither local nor global, they are non-local names.

```
procedure A
real a;
    procedure B
    real b;
     reference a;  non-local
    end B
end A;
```



Most closely nested rule

2, 1

0, 3

0, 1

0, 0

Example: Non-local names in C

16) Explain in detail parameter passing.

Parameter Passing

Parameters

Names that appear in the declaration of a procedure are formal parameters.

Variables and expressions that are passed to a procedure are actual parameters (or arguments)

Parameter passing modes

Call by value

Call by reference

Copy-restore

Call by name

Call-by-Value

.The actual parameters are evaluated and their *r-values* are passed to the called procedure

A procedure called by value can affect its caller either through nonlocal names or through pointers.

Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.

Pascal uses pass by value by default, but *var* parameters are passed by reference.

Call-by-Reference

Also known as call-by-address or call-by-location. The caller passes to the called procedure the *l-value* of the parameter.

If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.

Parameters in Fortran are passed by reference

an old implementation bug in Fortran

```
func(a,b) { a = b};
call func(3,4); print(3);
```

Copy-Restore

A hybrid between call-by-value and call-by reference.

The actual parameters are evaluated and their r-values are passed as in call-by-value. In addition, l-values are determined before the call.

When control returns, the current r-values of the formal parameters are copied back into the l-values of the actual parameters.

Call-by-Name

The actual parameters literally substituted for the formals. This is like a macro-expansion or in-line .expansion

Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used.

In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.