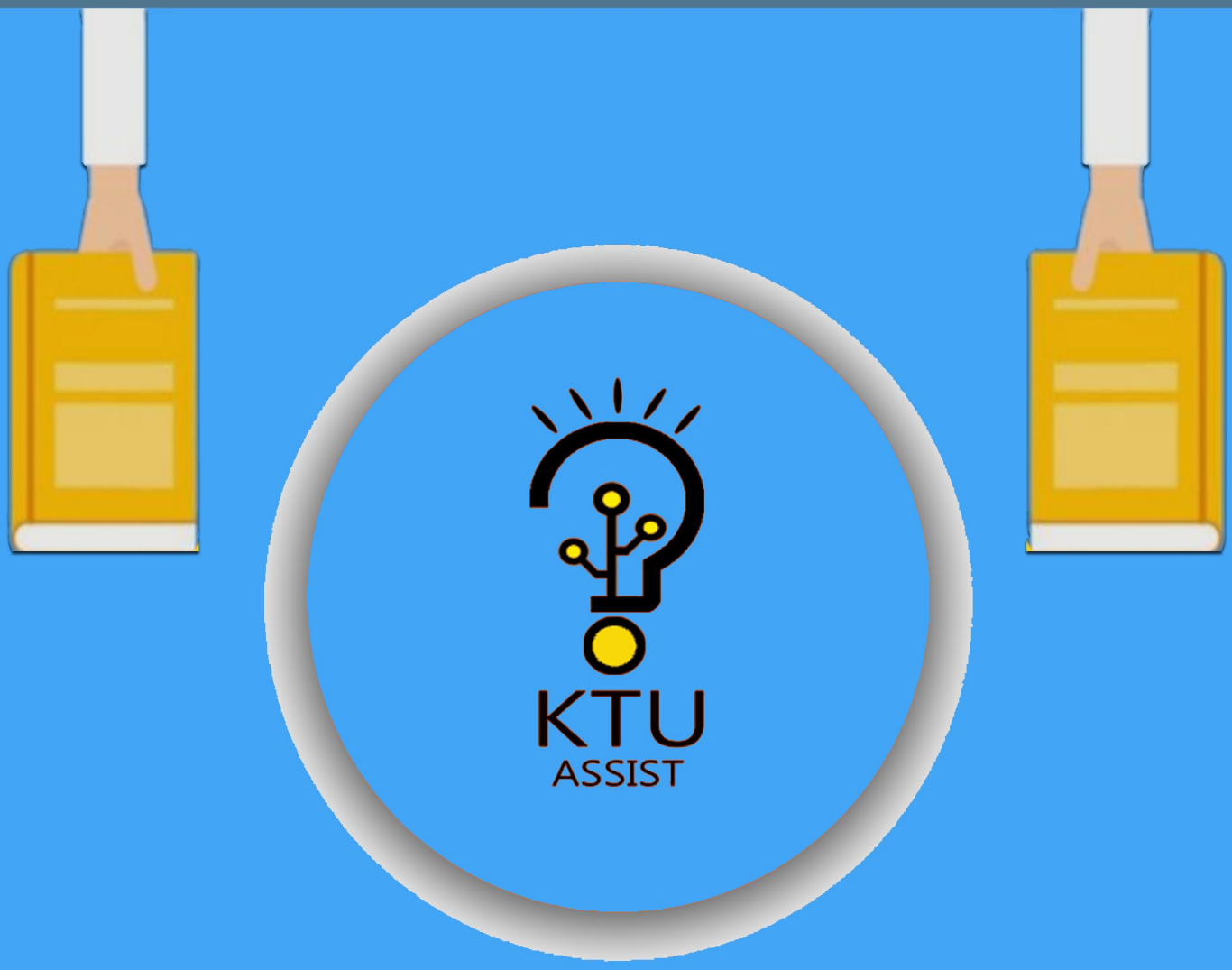
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

SIXTH SEMESTER B.TECH DEGREE MODEL EXAMINATION, MARCH 2018

Department: **Computer Science and Engineering**

Subject: - CS304: COMPILER DESIGN

Time: 3 hours Max. Marks: 100

PART A
Answer *all* questions

1. **What are the tools used for compiler construction?** **(3)**
2. **What is a symbol table?** **(3)**
3. **Differentiate regular expression and regular definition with an example.** **(3)**
4. **Differentiate tokens, patterns, lexeme** **(3)**

Total: (12)

PART B
Answer any *two* full questions

5. **With a neat diagram, explain the different phases of a compiler. Mention the input and output of each phase.** **(9)**
6. **Explain Input buffering with example.** **(9)**
7. **Explain the construction of Predictive parsing table for the grammar.**

E->TE'

E'->+TE'|e

T->FT'

T'->*FT' | e

F->(E) | id

(9)

Total: (18)

PART C
Answer *all* questions

8. **What is an operator precedence parser?** **(3)**
9. **Define viable prefix, kernel & non-kernel items.** **(3)**
10. **Give the syntax-directed definition for if-else statement** **(3)**
11. **Explain synthesized and inherited attributes.** (3)

Total: (12)

PART D

Answer any *two* full questions

12. Construct LR(0) items of following grammar

S-> L=R
S -> R
L-> *R
L-> ID
R-> L

(9)

13. Construct CLR parsing table for the following grammar **(9)**

S -> CC

C -> cC | d

14. Explain bottom up evaluation of S attributed definitions. (9)

Total: (18)

PART E

Answer any four full questions

15. Explain Storage- allocation strategies. **(10)**

16. a. Explain implementations of three address statements. **(5)**

b. Draw syntax tree and DAG for statement a:=b*-c+b*-c **(5)**

17. Explain Principal sources of code optimization **(10)**

18. Explain the issues in the design of code generator. (10)

Total: (40)

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

SIXTH SEMESTER B.TECH DEGREE MODEL EXAMINATION, MARCH 2018

Department: **Computer Science and Engineering**

Subject: - CS304: COMPILER DESIGN

ANSWER KEY

Time: 3 hours Max. Marks: 100

PART A

Answer *all* questions

1. **a)Parser Generators**

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

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

   b)Scanner Generators

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

   c)Syntax-directed Translation Engines

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

   d)Automatic Code Generators

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

   e)Data-flow Analysis Engines

   Data-flow analysis engine gathers the information, that is, the values transmitted from one part of a program to each of the other parts. Data-flow analysis is a key part of code optimization. **(3)**

2. **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. Whenever an identifier is detected by a lexical analyzer, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyzer.**

(3)

3. **Regular expression: It is *representator* of regular language. Regular expression is mathematically represent by some expression called regular expression. Regular expression is character sequence that define a search pattern.**
   Eg: Regular expression for identifier is letter(letter/digit)*
   Regular definition: is the name given to regular expression
   eg:Regular definition is id---->letter(letter/digit)*

(3)

4. **Tokens- Sequence of characters that have a collective meaning.**

Patterns- There is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token

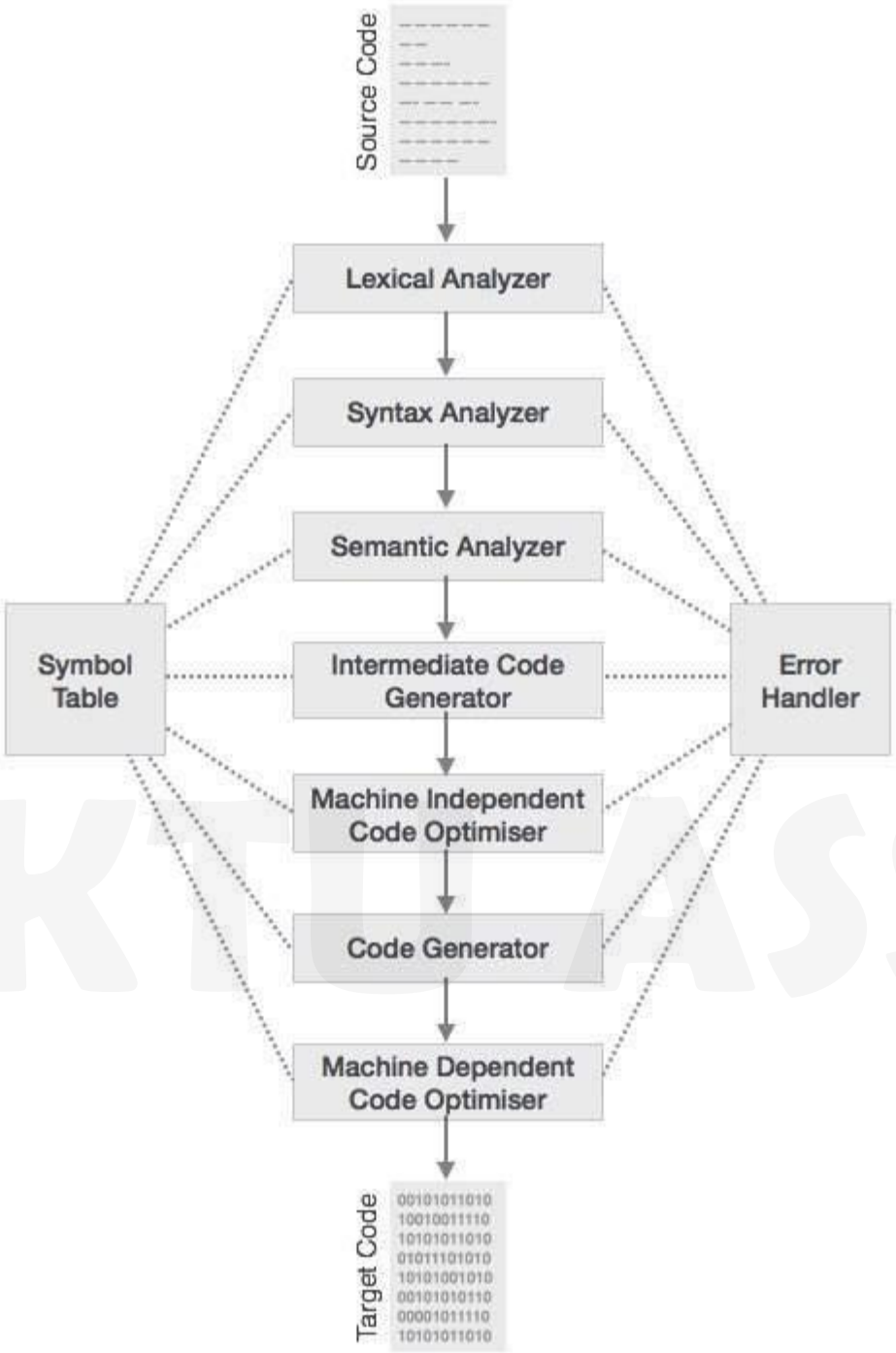
Lexeme- A sequence of characters in the source program that is matched by the pattern for a token.

(3)

Total: (12)

PART B
Answer any *two* full questions

5. **The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler.**



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

(9)

6. **Explain Input buffering with example.**

To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.

• Hence a two-buffer scheme is introduced to handle large lookaheads safely.

• Techniques for speeding up the process of lexical analyzer such as the use of sentinels to mark the buffer end have been adopted.

There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical-analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression based specification. In this, the generator provides routines for reading and buffering the input.

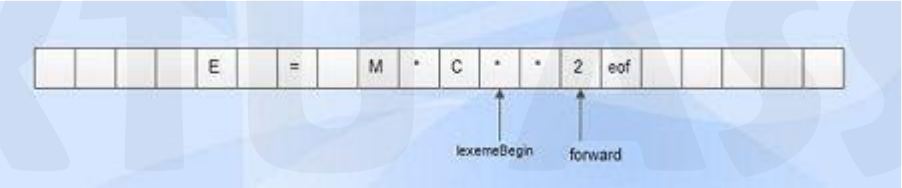
(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Buffer Pairs

Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Fig shows the buffer pairs which are used to hold the input data.



Scheme

• Consists of two buffers, each consists of N-character size which are reloaded alternatively.

• N-Number of characters on one disk block, e.g., 4096.

• N characters are read from the input file to the buffer using one system read command.

• *eof* is inserted at the end if the number of characters is less than N.

Pointers

Two pointers *lexemeBegin* and *forward* are maintained.

***lexeme Begin* points to the beginning of the current lexeme which is yet to be found.**

***forward* scans ahead until a match for a pattern is found.**

• Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.

• Current lexeme is the set of characters between two pointers.

Disadvantages of this scheme

• This scheme works well most of the time, but the amount of lookahead is limited.

• This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

(eg.) DECLARE (ARGl, ARG2, . . . , ARGn) in PL/1 program;

• It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Sentinels

• In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.

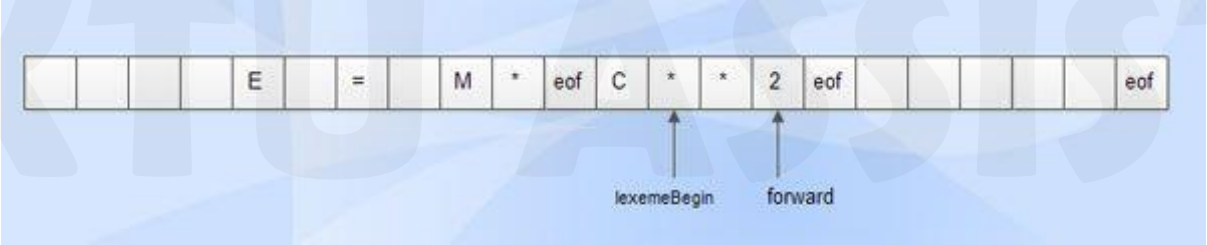
• Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

Test 1: **For end of buffer.**

Test 2: **To determine what character is read.**

• The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

• The sentinel is a special character that cannot be part of the source program. *(eof* **character is used as sentinel).**



Advantages

• Most of the time, It performs only one test to see whether forward pointer points to an *eof*.

• Only when it reaches the end of the buffer half or *eof,* it performs more tests.

• Since N input characters are encountered between *eofs,* the average number of tests per input character is very close to 1.

(9)

7.

Predictive parser can be implemented by recursive-descent parsing (may need to manipulate the grammar, e.g eliminating left recursion and left factoring).

Requirement: by looking at the first terminal symbol that a nonterminal symbol can derive, we should be able to choose the right production to expand the nonterminal symbol.

If the requirement is met, the parser easily be implemented using a non-recursive scheme by building a parsing table.

(9)

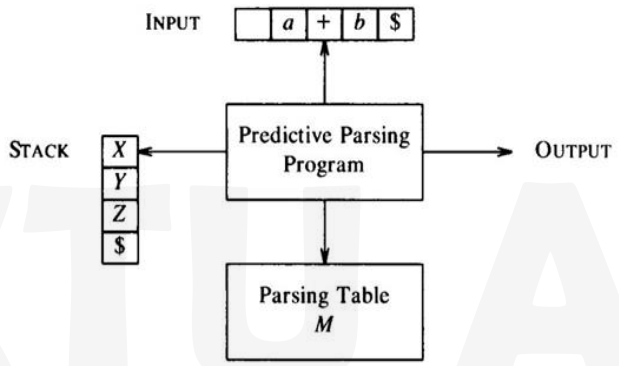


Fig. 4.13. Model of a nonrecursive predictive parser.

- **First(a) - the set of tokens that can appear as the first symbols of one or more strings generated from a. If a is empty string or can generate empty string, then empty string is also in First(a).**
- **Predictive parsing won't work on some type of grammars:**
  - **Left recursion: A->Aw**
  - **Have common left factor: A->aB | aC**

Predictive parsers can be constructed for a class of grammars called LL(1).

L->Left
L->Leftmost derivation
1->One input symbol at each step

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

<u>First(a):</u> Here, a is a string of symbols. The set of terminals that begin strings derived from a. If a is empty string or generates empty string, then empty string is in First(a).

<u>Follow(A):</u> Here, A is a nonterminal symbol. Follow(A) is the set of terminals that can immediately follow A in a sentential form

E->TE' First(E) = {(, id}, Follow(E)={), $}

- **E'->+TE'|e First(E')={+, e}, Follow(E') = {), $}**
- **T->FT' First(T) = {(, id}, Follow(T) = {+,), $}**
- **T'->*FT' | e First(T') = {*, e}, Follow(T') = {+,), $}**
- **F->(E) | id First(F) = {(, id}, Follow(F) = {*, +,), $}**

Algorithm for construction of parsing table

INPUT :- Grammar G
OUTPUT:- Parsing table M
For each production A -> α , do the following :
For each terminal 'a' in FIRST(A), add A-> α to M[A,α].
If ε is in FIRST(α) then for each terminal b in FOLLOW(A) add A -> α to M[A,b]. If b is $ then also add A -> α to M[A,$].
If there is no production in M[A,a] , then set M[A,a] to error.

Non Terminal			INPUT SYMBOLS			
	id	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E -> +TE'			E' -> ε	E' ->
T	T -> FT'			T -> FT'		
T'		T' -> ε	T' -> *FT'		T' -> ε	T' ->
F	F -> id			F -> (E)		

Step	Stack	Input	Next Action
1	$E	id*id$	E→TE'
2	$E'T	id*id$	T→FT'
3	$E'T'F	id*id$	F→id
4	$E'T'id	id*id$	match id

5	$E'T'	*id$	T'→*FT'
6	$T'F*	*id$	match *
7	$T'F	id$	F→id
8	$T'id	id$	match id
9	$T'	$	T'→ε
10	$	$	accept

(9)

8. A grammar is said to be operator precedence if it possess the following properties:

1. No production on the right side is ε.

2. There should not be any production rule possessing two adjacent non terminals at the right hand side.

(3)

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

Kernel items, whish include the initial item, S'→ .S, and all items whose dots are not at the left end.

Non-kernel items, which have their dots at the left end. **(3)**

10.

1. S → if E then S1

E.true := new_label()

E.false :=S.next

S1.next :=S.next

S.code :=E.code | | gen_code(E.true ': ') | | S1.code

2. S → if E then S1 else S2

E.true := new_label()

E.false := new_label()

S1.next :=S.next

S2.next :=S.next

S.code :=E.code | | gen_code(E.true ‘: ‘) | | S1.code| | gen_code(‘go to’,S.next) | |gen_code(E.false ‘:’) | |

S2.code **(3)**

11.

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

S → ABC

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

As in our previous example (E → E + T), the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

(3)

PART D
Answer any *two* full questions

12. LR Parsers

+ Can be constructed to recognize virtually all programming languages constructed from context-free grammars
+ Most general non-backtracking shiftreduce parsing method
+ Very fast at detecting syntactic errors - Too much work to construct LR parsing by hand

- Item: production rule with information about current parsing state

$A \to XYZ$ yields the four items

$A \to \cdot XYZ$
$A \to X \cdot YZ$
$A \to XY \cdot Z$
$A \to XYZ \cdot$

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

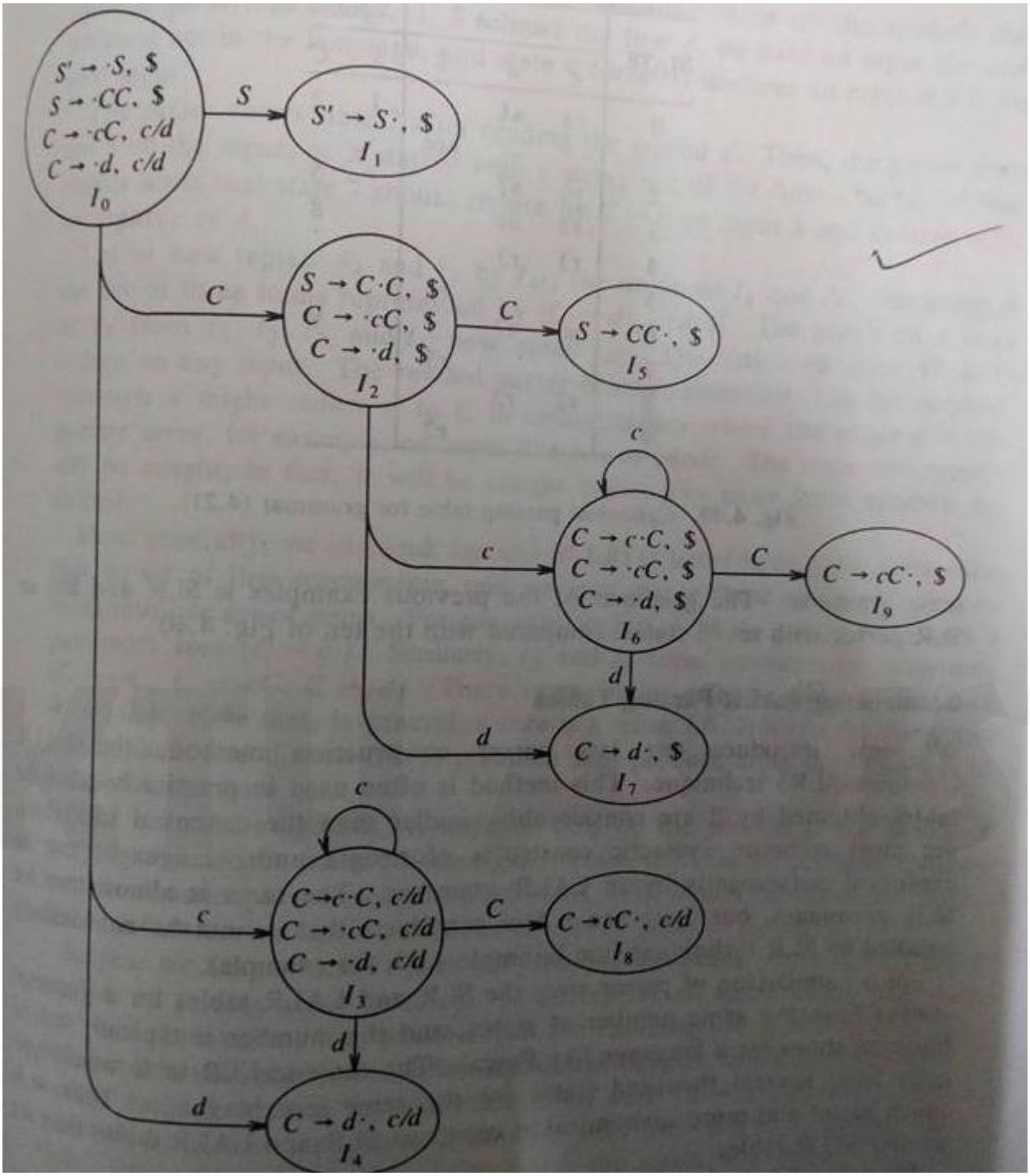
I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot * R$
$L \to \cdot \mathbf{id}$

I_7: $L \to * R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

13. (9)

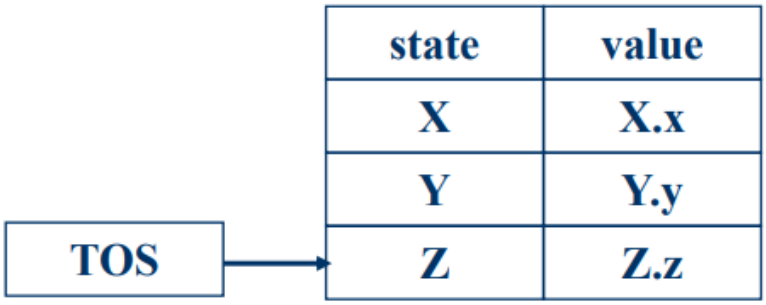


STATE	action			goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

(9)

14.

- Syntax-directed definitions with only synthesised attributes
- can be evaluated by a bottom up parser (BUP) as input is parsed
- values associated with the attributes can be kept on the stack as extra fields
- implementation using an LR parser (e.g. YACC)
- e.g. A => XYZ and A.a := f (X.x, Y.y, Z.z)



S-attributed Definitions: Parser Example

production		semantic rules
L	=> E \n	print(val[TOS])
E	=> E_1 + T	val[NTOS] := val[TOS-2] + val[TOS]
E	=> T	
T	=> T_1 * F	val[NTOS] := val[TOS-2] * val[TOS]
T	=> F	
F	=> (E)	val[NTOS] := val[TOS-1]
F	=> id	

- NTOS = TOS - r + 1 (r is # symbols reduced on RHS)

example of 3 * 5 + 4

S-attributed Definitions: Parser Example

input	state	value	production used
3 * 5 + 4 \n	-	-	
* 5 + 4 \n	3	3	
* 5 + 4 \n	F	3	F => digit
* 5 + 4 \n	T	3	T => F
5 + 4 \n	T *	3 :	
+ 4 \n	T * 5	3 : 5	
+ 4 \n	T * F	3 : 5	F => digit
+ 4 \n	T	15	T => T * F
+ 4 \n	E	15	E => T
4 \n	E +	15 :	
\n	E + 4	15 : 4	
\n	E + F	15 : 4	F => digit
\n	E + T	15 : 4	T => F
\n	E	19	E => E + T
	E \n	19 :	
	L	19	L => E \n

(9)

PART E
Answer any four full questions

15. **The different storage allocation strategies are;**

1. Static allocation lays out storage for all data objects at compile time.

2. Stack allocation manages the run-time storage as a stack.

3. Heap allocation: allocates and de-allocates storage as needed at runtime from a data known as the heap.

Static allocation

- **In static allocation, names bound to storage as the program is compiled, so there is no need for a run-time support package.time**
- **Since the bindings do not change at runtime, every time a procedure activated, its run-time, names bounded to the same storage location.**
- **Therefore values of local names retained across activations of a procedure. That is when control returns to a procedure the value of the local are the same as they were when control left the last time.**

Stack allocation : Storage allocation strategies

- **All compilers for languages that use procedures, functions or methods as units of user functions define actions manage at least part of their runtime memory as a stack run-time stack.**
- **Each time a procedure called, space for its local variables is pushed onto a stack, and when the procedure terminates, space popped off t stack.the**

Calling Sequences: Storage allocation strategies

- **Procedures called implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.**
- **A return sequence is similar to code to restore the state of a machine so the calling procedure can continue its execution after the call.**
- **The code is calling sequence of often divided between the calling procedure (caller) and a procedure is calls (callee)(callee).**
- **When designing calling sequences and the layout of activation record, the following principles are helpful:**
  1. Value communicated between caller and callee generally placed at the caller beginning of the callee's activation record, so they as close as possible to the
  caller's activation record.
  2. Fixed length items generally placed in the middle. Such items typically include the control link, the access link, and the machine status field.,
  3. Items whose size may not be known early enough placed at the end of the activation record.
  4. We must locate the top of the stack pointer judiciously. A common approach is to have it point to the end of fixed length fields in the activation is to have it point to fix the end of fixed length fields in the activation record. Fixed length data can then be accessed by fixed offsets, known to the intermediate code generator, relative to the top of the stack pointer.

Storage allocation strategies

- **The calling sequence and its division between caller and callee are as follows:**
  1. The caller evaluates the actual parameters.
  2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments the top_sp to the respective positions.
  3. The callee-saves the register values and other status information.
  4. The callee initializes its local data and begins execution.

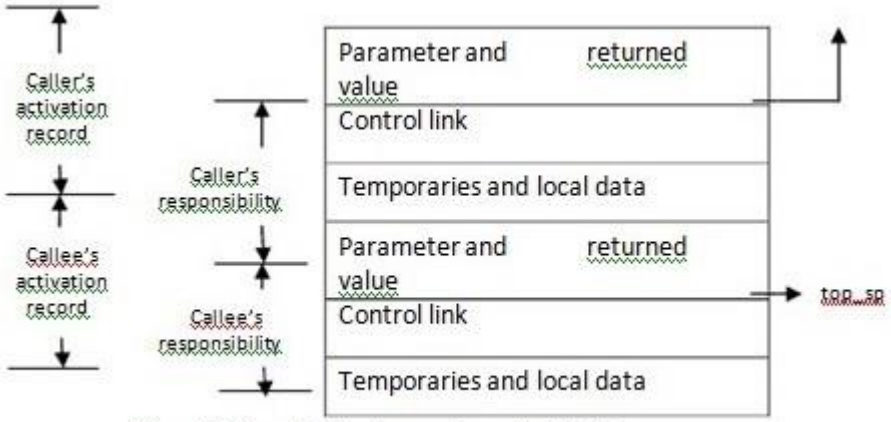


Fig. Division of task between caller and callee6.4

- **A suitable, corresponding return sequence is:**
  1. The callee places the return value next to the parameters.
  2. Using the information in the machine status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.
  3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller, therefore, may use that value.

Variable length data on stack: Storage allocation strategies

- **The runtime memory management system must deal frequently with the allocation of management objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.**
- **The same scheme works for objects of any type if they are local to the procedure called local have a size that depends on the parameter of the call.**

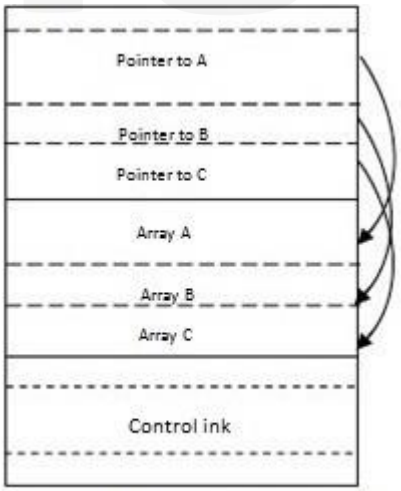


Fig Access to dynamically allocated arrays

Dangling Reference (Storage allocation strategies)

- **Whenever storage allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference to storage that has been allocated.**
- **It is a logical error to u dangling reference, since, the value of de use de-allocated storage is undefined according to the semantics of most languages.**
  - **Whenever storage allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference to storage that has been allocated.** (10)

16. a.

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples, triples, Indirect Triples

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3

=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

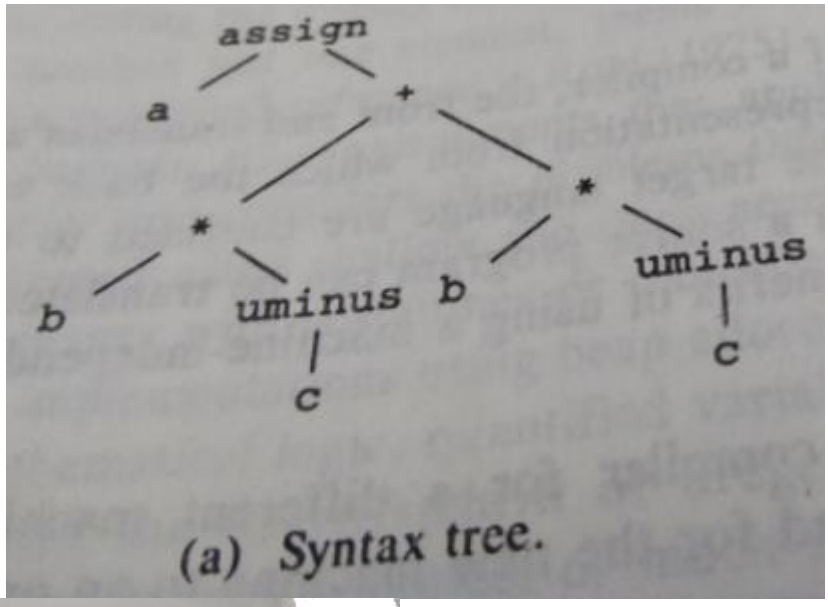
Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

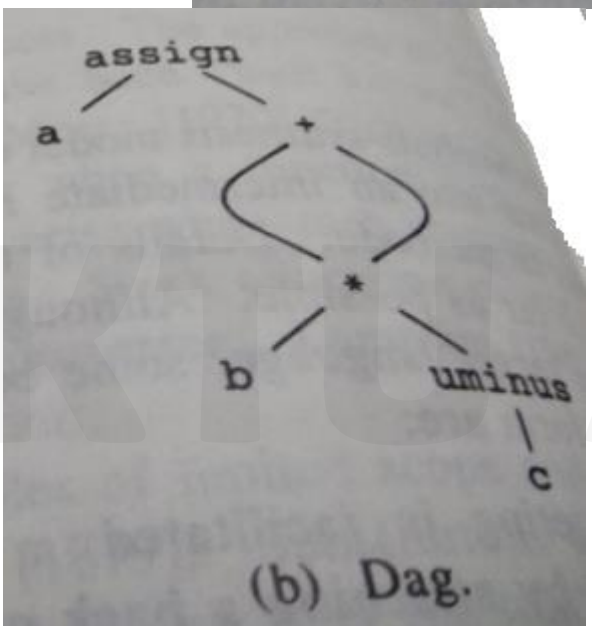
This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

(5)

b.



(a) Syntax tree.



(b) Dag.

(5)

17. **A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.**

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations examples:

- **Common sub expression elimination**
- **Copy propagation,**
- **Dead-code elimination**
- **Constant folding**

The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

***

Common Sub expressions elimination:

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

- **For example**

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
 t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

• For example:
x=Pi;

A=x*r*r;

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

Here the variable x is eliminated

Dead-Code Eliminations:

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

Example:

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

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

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

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- **Code motion, which moves code outside a loop;**
- **Induction-variable elimination, which we apply to replace variables from inner loop.**

Ø **Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.**

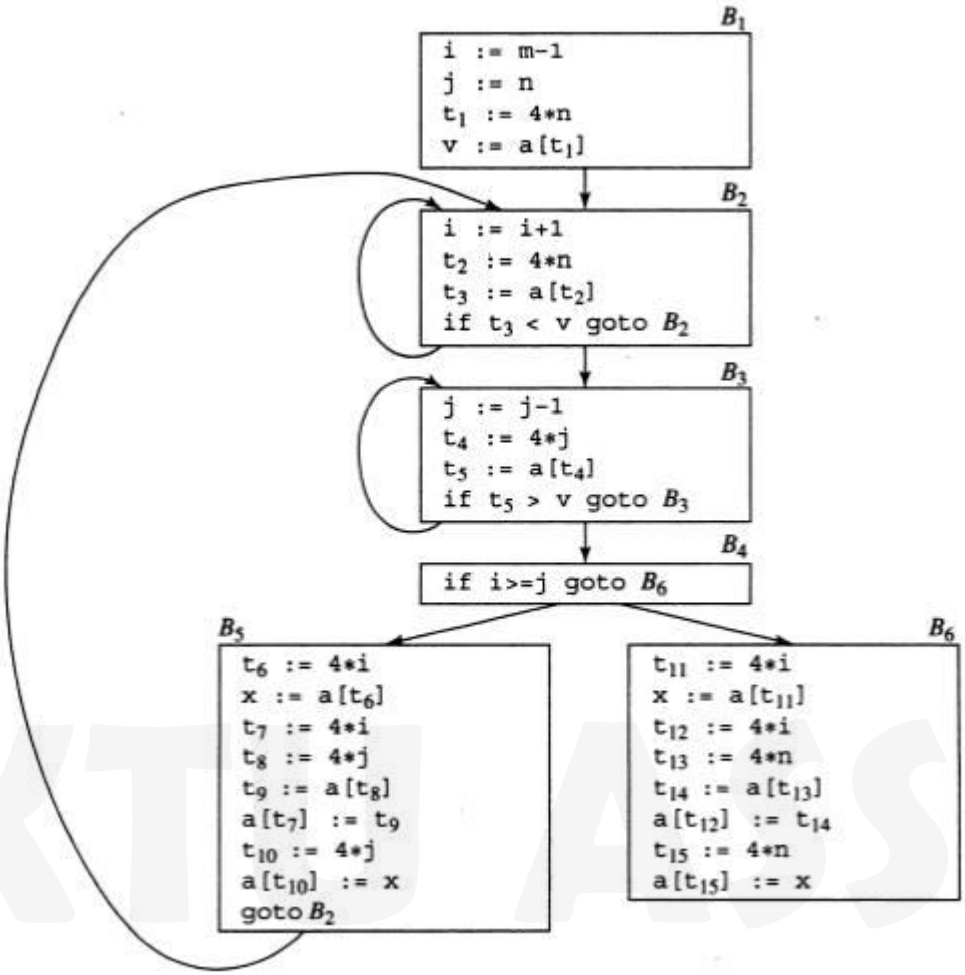


Fig. 5.2 Flow graph

Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /* statement does not change limit*/

Code motion will result in the equivalent of

```
t= limit-2;
while (i<=t) /* statement does not change limit or t */
```

Induction Variables :

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

When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Example:

As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Fig.5.3.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

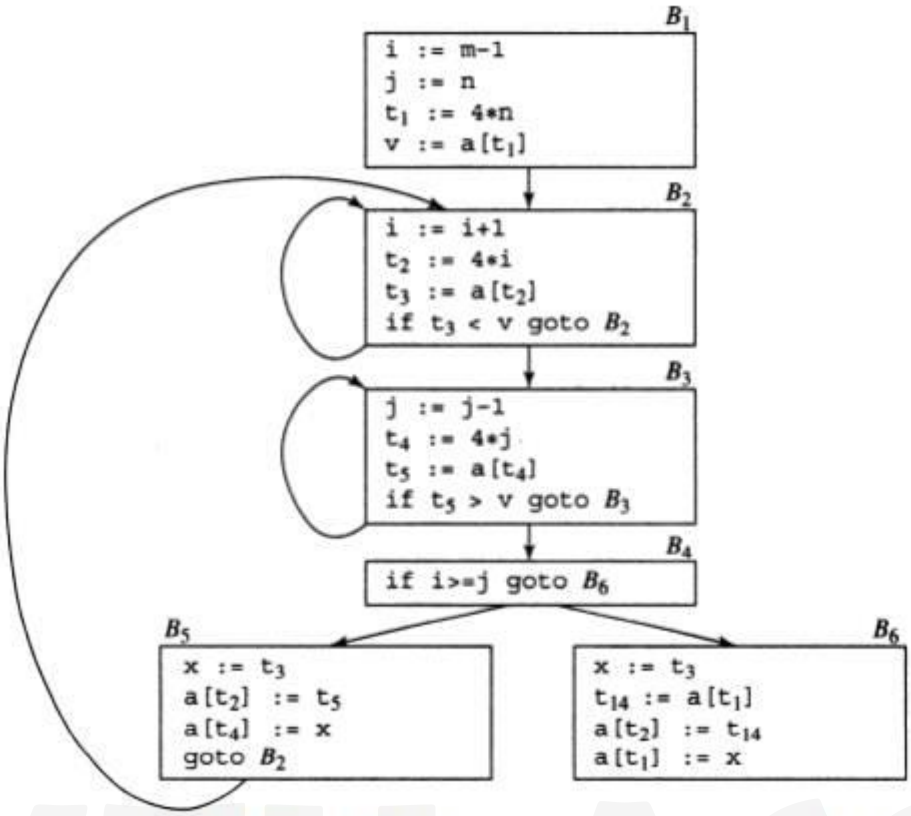


Fig. 5.3 B5 and B6 after common subexpression elimination

Fig. 5.3 B5 and B6 after common subexpression elimination

(10)

18. **Code generator phase generates the target code taking input as intermediate code.**

- **The output of intermediate code generator may be given directly to code generation or may pass through code optimization before generating code.**
- **Code generator main tasks:**
    1. **Instruction selection**
    2. **Register allocation and assignment**
    3. **Instruction ordering**

Issues in Design of Code generation:

- **Target code mainly depends on available instruction set and efficient usage of registers. The main issues in design of code generation are**

    1. Intermediate representation

- **Linear representation like postfix and three address code or quadruples and graphical representation like Syntax tree or DAG.**
- **Assume type checking is done and input in free of errors.**

2. Target Code

- **The code generator has to be aware of the nature of the target language for which the code is to be transformed.**
- **That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way.**
- **The target machine can have either CISC or RISC processor architecture.**
- **The target code may be absolute code, re-locatable machine code or assembly language code.**
- **Absolute code can be executed immediately as the addresses are fixed.**
- **But in case of re-locatable it requires linker and loader to place the code in appropriate location and map (link) the required library functions.**
- **If it generates assembly level code then assemblers are needed to convert it into machine level code before execution.**
- **Re-locatable code provides great deal of flexibilities as the functions can be compiled separately before generation of object code.**

1.1 Absolute machine language:

- **Producing an absolute machine language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed.**

1.2 Relocatable machine language:

- **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.**
- **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.**

3. Assembly language:

- **Producing an assembly language program as output makes the process of code generation somewhat easier.**

Example

MOV R0, R1

ADD R1, R2

Target Machine supports for the following addressing modes

a. Absolute addressing mode

Example: MOV R0, M where M is the address of memory location of one of the operands. MOV R0, M moves the contents of register R0 to memory location M.

b. Register addressing mode where both the operands are in register.

Example: ADD R0, R1

c. Immediate addressing mode – The operand value appears in the instruction.

Example: ADD # 1, R0

d. Index addressing mode- this is of the form C(R) where the address of operand is at the location C+Contents(R)

Example: MOV 4(R0), M the operand is located at address = contents (4+contents (R0)) Cost of instruction is defined as cost of execution plus the number of memory access.

3. Address mapping

- **Address mapping defines the mapping between intermediate representations to address in the target code.**
- **These addresses are based on the runtime environment used like static, stack or heap.**
- **The identifiers are stored in symbol table during declaration of variables or functions, along with type.**
- **Each identifier can be accessed in symbol table based on width of each identifier and offset. The address of the specific instruction (in three address code) can be generated using back patching**

4. Instruction Set

- **The instruction set should be complete in the sense that all operations can be implemented.**
- **Sometimes a single operation may be implemented using many instruction (many set of instructions).**
- **The code generator should choose the most appropriate instruction.**
- **The instruction should be chosen in such a way that speed is of execution is minimum or other machine related resource utilization should be minimum.**
- **The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set.**
- **One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.**

Example

The factors to be considered during instruction selection are:

- **The uniformity and completeness of the instruction set.**
- **Instruction speed and machine idioms.**
- **Size of the instruction set.**

Eg., for the following address code is:

a := b + c

d := a + e

inefficient assembly code is:

```
MOV b, R0        R0 ← b
ADD  c, R0       R0 ← c + R0
MOV R0, a        a  ← R0
MOV a, R0        R0 ← a
ADD  e, R0       R0 ← e + R0
MOV R0 , d       d  ← R0
```

Here the fourth statement is redundant, and so is the third statement if

'a' is not subsequently used.

5. Register Allocation

- **A program has a number of values to be maintained during the execution.**
- **The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers.**
- **Code generator decides what values to keep in the registers.**
- **Also, it decides the registers to be used to keep these values.**
- **Instructions involving register operands are usually shorter and faster than those involving operands in memory.**
- **Therefore efficient utilization of registers is particularly important in generating good code.**
- **During register allocation we select the set of variables that will reside in registers at each point in the program.**
- **During a subsequent register assignment phase, we pick the specific register that a variable will reside in.**

6. Memory Management

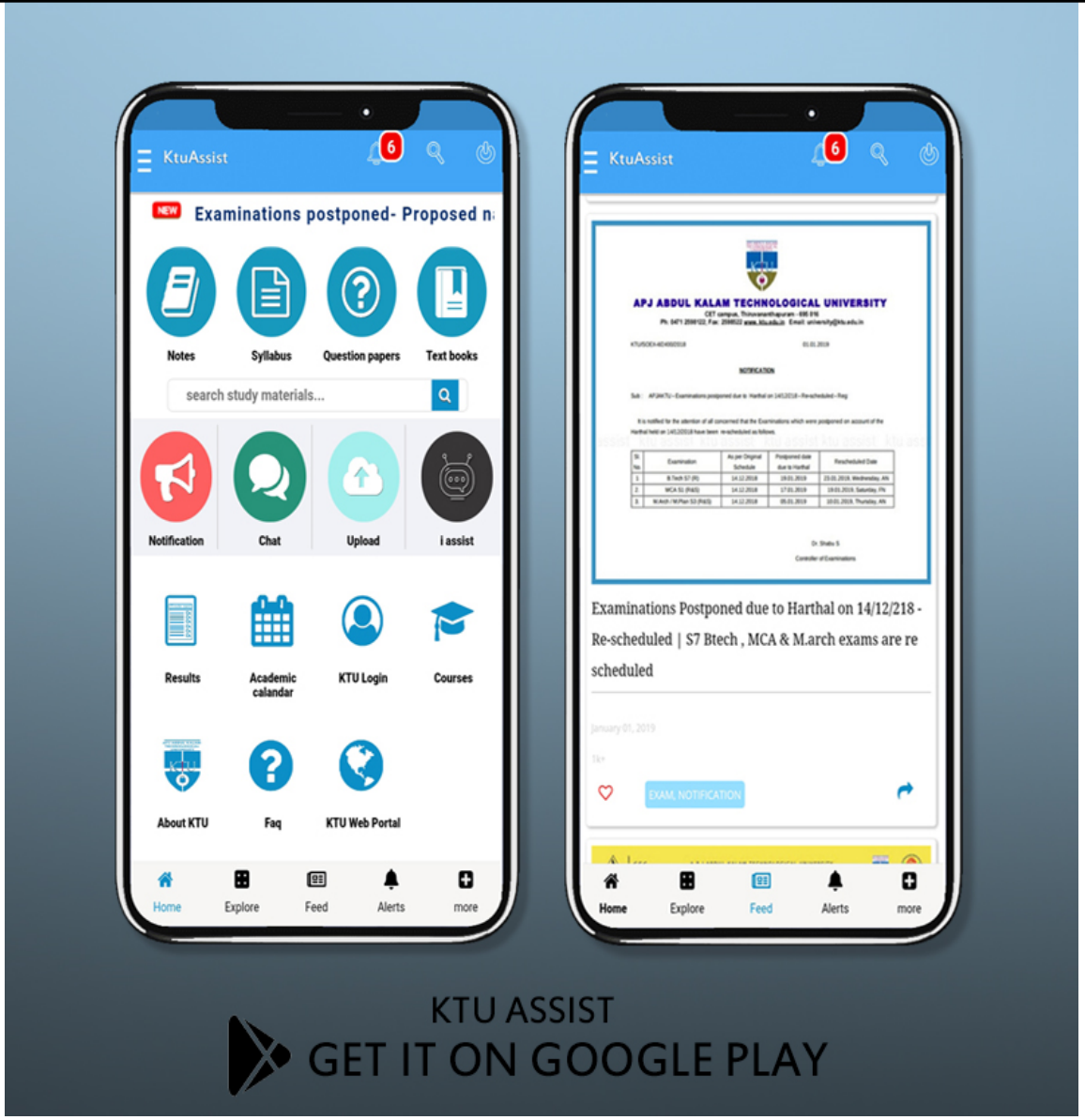
- **Mapping names in the source program to addresses of data objects in run-time memory is done cooperatively by the front end and the code generator.**
- **A name in a three- address statement refers to a symbol-table entry for the name.**
- **From the symbol-table information, a relative address can be determined for the name in a data area for the procedure.**

7. Evaluation of order

- **At last, the code generator decides the order in which the instruction will be executed.**
- **It creates schedules for instructions to execute them.**
- **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.**

(10)

Total: (40)



END

facebook.com/ktuassist

instagram.com/ktu_assist