**UNIT IV**

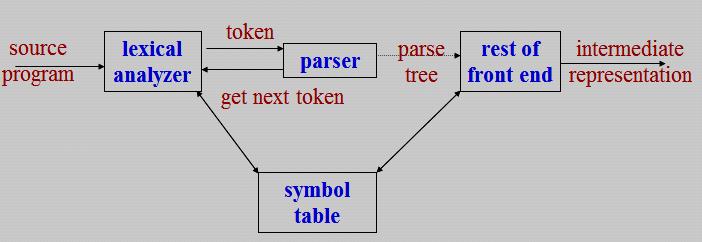
**PARSING**

**Parsing:** Role of Parser–Context free Grammars–Writing a Grammar–Predictive Parser–LRParser. **Intermediate Code Generation:** Intermediate Languages – Declarations – Assignment Statements – Boolean Expressions – Case Statements – Back Patching – Procedure Calls.

P a g e | **1** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**2 MARKS**

1. **What is the role of parser?** 
   * In compiler model, parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source program.
   * The parser should report any syntax errors in an intelligible fashion.



**2. What is meant by parser?**

A parser for grammar G is a program that takes as input a string ‘w’ and produces as output either a parse tree for ’w’, if ‘w’ is a sentence of G, or an error message indicating that w is not a sentence of G. It obtains a string of tokens from the lexical analyzer, verifies that the string generated by the grammar for the source language.

**3. What are the types of Parser?**

There are three general types of parsers for grammars.

1. Universal parsing methods
   * Cocke-Younger –Kasami algorithm and
   * Earley’s algorithm
2. Top down parser
3. Bottom up parser
4. **What are the different levels of syntax error handler?** 
   * Lexical, such as misspelling an identifier, keyword, or operator
   * Syntactic, such as an arithmetic expression with unbalanced parentheses
   * Semantic, such as an operator applied to an incompatible operand
   * Logical, such as an infinitely recursive call

P a g e | **2** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What are the goals of error handler in a parser?** 
   * It should report the presence of errors clearly and accurately.
   * It should recover from each error quickly enough to be able to detect subsequent errors.
   * It should not significantly slow down the processing of correct programs.
2. **What are error recovery strategies in parser?** 
   * Panic mode
   * Phrase level
   * Error productions
   * Global correction
3. **Define context free grammar? (NOV 2011)**

A Context Free Grammar (CFG) consists of terminals, non-terminals, a start symbol and productions. The Grammar G can be represented as G = (V, T, S, P)

* + V is a set of non-terminals
  + T is a set of terminals
  + S is a start symbol
  + P is a set of production rules Production rules are given in the following form

Non terminal  (V U T)\*

1. **Define derivation.**

Derivation is the top-down construction of parse tree. The production treated as a rewriting rule in which the non-terminal on the left is replaced by the string on the right side of the production.

1. **What is left-most and right-most derivation?** 
   * The left-most non-terminal in each derivation step, this derivation is called as left-most derivation.
   * The right-most non-terminal in each derivation step, this derivation is called as right-most derivation (Canonical derivation).
2. **What is Parsing Tree? (MAY 2012)** 
   * A parse tree can be viewed as a graphical representation for a derivation.
   * The leaves of a parse tree are terminal symbols.
   * Inner nodes of a parse tree are non-terminal symbols.
3. **Define yield of the string?**

The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right; they constitute a sentential form called the yield or frontier of the tree.

P a g e | **3** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Define ambiguous. (MAY 2012)** 
   * A grammar that produces more than one parse tree for a sentence is said to be *ambiguous.*
2. **Define ambiguous grammar.** 
   * An ambiguous grammar is one that produces more than one left most or more than one right most derivation for the same sentence.
3. **What is left recursion?**

+

 A grammar is left recursive if it has a non-terminal A such that there is a derivation **A**  **A****.**

 Top-down parsing techniques *cannot* handle left-recursive grammars, so a transformation that eliminates left recursion is needed.

**Example:** The left recursive pair of productions **A****A** **| β** could be replaced by non- left- recursiveproductions

**A****βA’ A’** **A’ | ε**

1. **Define left factoring?** 
   * Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parser.
2. **What are the problems with top down parsing?**

The following are the problems associated with top down parsing:

* + - Backtracking
    - Left recursion
    - Left factoring
    - Ambiguity

1. **Define top down parsing?** 
   * Top-down parser viewed as an attempt to find the left most derivation for an input string. It can be viewed as attempting to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
   * Top down parsing called recursive descent that may involve backtracking ie. making repeated scanning of the input.
2. **What is meant by recursive-descent parser?** 
   * A parser that uses a set of recursive procedures to recognize its input with no backtracking is called a recursive-descent parser.
   * This recursive-descent parser called predictive parsing.

P a g e | **4** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**19. Briefly describe the LL (k) items.** . **(NOV 2013)**

In LL (k) the first “L “ scanning the input from left to right and

|  |  |
| --- | --- |
| second “L” | producing a leftmost derivation and |
| the “1” | one input symbol of lookahead at each step |

1. **What are the possibilities of non-recursive predictive parsing?** 
   1. If X = a = $, the parser halts and announces successful completion of parsing.
   2. If X = a = $, the parser pops X off the stack and advances the input pointer to the next symbol.
   3. If X is a nonterminal, the program consults entry M[X, a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry.
2. **Write the algorithm for FIRST and FOLLOW.**

**FIRST**

1. If X is terminal, and then FIRST(X) is {X}.
2. If X ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X  Y1,Y2….Yk is a production, then place a in FIRST(X) if for some i , a is in FIRST(Yi) , and ε is in all of FIRST(Y1),…FIRST(Yi-1);

**FOLLOW**

* + - 1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right end marker.
      2. If there is a production A  αBβ, then everything in FIRST (β) except for ε is placed in FOLLOW (B).
      3. If there is a production AαB, or a production AαBβ where FIRST (β) contains ε, then everything in FOLLOW (A) is in FOLLOW (B).

1. **What is bottom up parser?** 
   * + Bottom-up parsing is also known as shift-reduce parsing.
     + Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
2. **Define handle?** 
   * A **handle** of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a right most derivation.

P a g e | **5** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What is meant by handle pruning?** 
   * + A rightmost derivation in reverse can be obtained by *handle pruning*.
     + If w is a sentence of the grammar at hand, then w = γn, where γn is the *n*th right-sentential form of some as yet unknown rightmost derivation

S = γ0 => γ1…=> γn-1 => γn = w

1. **What is meant by viable prefixes?** 
   * The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called *viable prefixes.*
   * An equivalent definition of a viable prefix is that it is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential
2. **Define LR (k) parser?**

LR parsers can be used to parse a large class of context free grammars. The technique is called ***LR (K)*** parsing.

* + “L” is for left-to-right scanning of the input
  + “R” for constructing a right most derivation in reverse
  + “k” for the number of input symbols of lookahead that are used in making parsing decisions.

1. **Mention the types of LR parser?**

The three methods in LR parser

* + Simple LR (SLR) parser
  + Canonical LR (CLR) parser
  + Lookahead LR (LALR) parser

1. **What are the techniques for producing LR parsing Table?** 
   1. Shift s, where s is a state
   2. Reduce by a grammar production A
   3. Accept and
   4. Error
2. **What are the two functions of LR parsing algorithm?**

The two functions in LR parsing algorithm are

 Action function  GOTO function

P a g e | **6** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**30. Define LALR grammar?**

The Lookahead (LALR) parser method is often used in practice because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming language can be expressed conveniently by an LALR grammar. If there are no parsing action conflicts, then the given grammar is said to be an LALR (1) grammar. The collection of sets of items constructed is called LALR

(1) collections.

**31. Define SLR parser?**

The parsing table consisting of the parsing action and goto function determined by constructing an SLR parsing table algorithm is called SLR(1) table. An LR parser using the SLR (1) table is called SLR (1) parser. A grammar having an SLR (1) parsing table is called SLR (1) grammar.

**32. Differentiate phase and pass.** . **(NOV 2012)**

|  |  |
| --- | --- |
| **Phase** | **Pass** |
|  |  |
|  Phase is often used to call such a single |  Number of passes of a compiler is the number of times |
| independent part of a compiler. | it goes over the source. |
|  It is used in complier. |  It also used in compiler. |
|  It has lexical, syntax, semantic analyzer, |  Compilers are indentified as one-pass or multi-pass |
| intermediate code generator, code optimizer, | compilers. |
| and code generator. |  I t is easier to write a one-pass compiler and also they |
|  | perform faster than multi-pass compilers. |
|  |  |

**33. State the function of an intermediate code generator. (MAY 2013)**

A source program can be translated directly into target language, some benefits of using machine-independent intermediate form:

* 1. Retargeting is facilitated; a compiler for different machine can be created by attaching a back end for the new machine to an existing front end.
  2. A machine-independent code optimizer can be applied to the intermediate representation.

1. **What are the syntax-directed methods?**

The syntax-directed methods can be used to translate into intermediate form programming language constructs such as

* Declaration
* Assignment statements
* Boolean Expression
* Flow of control statements

P a g e | **7** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What are the different forms of Intermediate representations? (NOV 2013)**

The three kinds of intermediate representations are

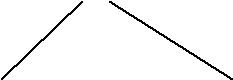
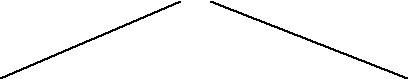
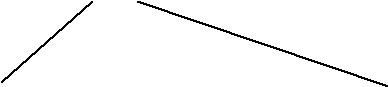
* 1. Syntax trees
  2. Postfix notation
  3. Three - address code

1. **How can you generate three-address code?**

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

1. **What is a syntax tree? Draw the syntax tree for the assignment statement.** 
   * A syntax tree depicts the natural hierarchical structure of a source program.
   * The syntax tree for the assignment statement **a: = b \* -c + b \* -c.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **assign** |  |  |  |  |  |
| **a** | **+** | |  |  |  |
| **\*** |  |  |  | **\*** |  |
| **b** | **uminus** | | **b** | **uminus** | |
|  |  |  |  |  |  |
|  | **c** | |  | **c** | |

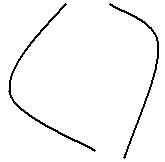


1. **Draw the dag for the assignment statement: a: = b \* -c + b \* -c. (NOV 2011)** 
   * Directed Acyclic Graph (DAG) gives more compact way for common sub-expressions are identified.
   * The DAG for the assignment statement **a: = b \* -c + b \* -c.**

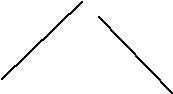
**assign**



**a** **+**



**\***



**b** **uminus**

**c**

P a g e | **8** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

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

**a b c uminus \* b c uminus \* + assign**

1. **What are the functions used to create the nodes of syntax trees?** 
   * + mkunode(op, child)
     + mknode(op, left, right)
     + mkleaf(id, entry)
2. **Define three-address code. (NOV 2012)** 
   * Three-address code is a sequence of statements of the general form
     + 1. ***:= y op z***
   * where x, y and z are names, constants, or compiler-generated temporaries;
   * op stands for any operator, such as fixed or floating-point arithmetic operator, or a logical operator on boolean-valued data.
3. **Construct three address codes for the following a: = b \* -c + b \* -c.**

The three address code as t1 := - c t2 := b \* t1 t3 := - c t4 := b \* t3

t5 := t2 + t4 a := t5

1. **List the types of three address statements.**

The types of three address statements are

* 1. Assignment statements
  2. Assignment Instructions
  3. Copy statements
  4. Unconditional Jumps
  5. Conditional jumps
  6. Procedure calls and return
  7. Indexed assignments
  8. Address and pointer assignments

P a g e | **9** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What are the various implementing three-address statements?**

The three implementation of three address statements are

* + 1. Quadruples
    2. Triples
    3. Indirect triples

1. **What is a quadruple?** 
   * A *quadruple* is a record structure with four fields, such as

***op, argl, arg2,*** 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.

1. **What are triples?**

 The Three-address statements can be represented by records with only three fields:

***op, arg1*** and ***arg2***

* + The fields *arg l* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure. Since three fields are used, this intermediate code format is known as ***triples.***
  + This method is used to avoid temporary names into the symbol table.

1. **Define indirect triples. Give the advantage?** 
   * Listing pointers to triples rather than listing the triples themselves. This implementation is called ***indirect triples.***

**Advantages:**

* + It can save some space compared with quadruples, if the same temporary value is used more than once.

1. **Write a short note on declarations?** 
   * Declarations in a 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.
   * The procedure ***enter (name, type, offset)*** create a symbol table entry for *name,* its *type* and relative address *offset* in its data area.

P a g e | **10** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**49. What are the semantic rules are defined in the declarations operations?**

The semantic rules are defined by the following ways

* + - 1. mktable(previous)
      2. enter(table, name, type, offset)
      3. addwidth(table, width)
      4. enterproc(table, name, newtable)

1. **What are the two primary purposes of Boolean Expressions?**

In Boolean expressions have two primary purposes

* + 1. They are used to compute logical values
    2. They are used as conditional expressions in statements that alter the flow of control, such as if-then, if-then-else, or while-do statements.

1. **Define Boolean Expression.** 
   * Boolean expressions which are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions.
   * Relational expression of the form **E1 relop E2,** where E1 and E2 arithmetic expressions.
   * Consider Boolean Expressions with the following grammar:

E  E ***or*** E | E ***and*** E | ***not*** E | (E) | id ***relop*** id | ***true*** | ***false***

**52. What are the methods of translating Boolean expressions?**

There are two principal methods of representing the value of a Boolean expression.

* 1. The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression.
  2. The second principal method of implementing boolean expression is by flow of control that is representing the value of a Boolean expression by a position reached in a program.

1. **What are the three address code for a *or b and not c* ?**

The three address sequence for a or b and not c t1 := not c

t2 := b and t1 t3 := a or t2

**54. What is meant by Shot-Circuit or jumping code?**

Translate a Boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “short-circuit” or “jumping” code.

P a g e | **11** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**55. Write a three address code for the expression a<b or c<d and e<f?**

The three address code as

* + 1. ***if*** a< b goto ***103***
    2. t1 := 0
    3. goto ***104***
    4. t1 := 1
    5. ***if*** c< d goto ***107***
    6. t2 := 0
    7. goto ***108***
    8. t2 := 1
    9. ***if*** e< f goto ***111***
    10. t3 := 0
    11. goto ***112***
    12. t3 := 1
    13. t4 := t2 ***and*** t3
    14. t5 := t1 ***or*** t4

1. **Define back patching.** 
   * Backpatching can be used to generate code for Boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated.
   * Each such statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. This subsequent filling of addresses for the determined labels is called

***Backpatching.***

1. **What are the three functions of backpatching?**

The three functions in backpatching are

1. makelist(i) – create a new list.
2. merge(*p1,p2*) – concatenates the lists pointed to by p1 and p2.
3. backpatch(p,i) – insert i as the target label for the statements pointed to by p.

P a g e | **12** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**58. Derive the first and follow for the follow for the following grammar. (MAY 2013)**

S0|1|AS0|BS0 A **ε** B **ε**

**Computation for FIRST:**

**FIRST(S)** = {0, 1} U FIRST (A) U FIRST (B) = {0, 1} U {ε} U {ε} = **{0, 1, ε}**

**FIRST (A)** = **{ε}**

**FIRST (B)** = **{ε}**

**Computation for FOLLOW:**

**FOLLOW (S)** = {$} U {0} U {0} = **{$, 0}**

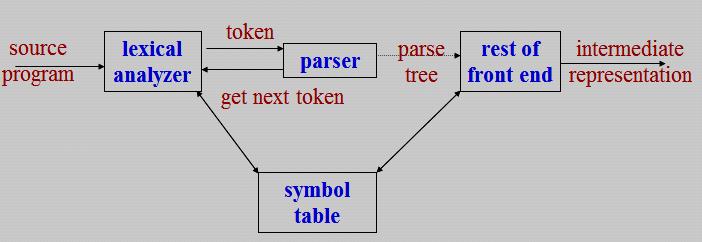
**FOLLOW (A)** =FOLLOW(S) = **{$, 0}**

**FOLLOW (B)** =FOLLOW(S) = **{$, 0}**

P a g e | **13** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* + 1. **MARKS**

1. **Explain the role of the parser? (11 marks) (MAY 2012)** 
   * In compiler model, parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source program.
   * The parser should report any syntax errors in an intelligible fashion.
   * It should also recover from commonly occurring errors so it can continue processing the remainder if it’s input.



**Position of parser in compiler model**

There are three general types of parsers for grammars.

* 1. Universal parsing methods  too inefficient to use in production compilers.
     + Cocke-Younger –Kasami algorithm and
     + Earley’s algorithm
  2. Top down parser  it builds parse trees from the top (root) to the bottom (leaves).
  3. Bottom up parser  it start from the leaves and work up to the root.
* In both cases, the input to the parser is scanned from left to right, one symbol at a time.
* The most efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  + - * LL for top-down parsing
      * LR for bottom-up parsing

P a g e | **14** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Syntax error handling:**

* If a compiler to process only correct programs, its design and implementation would be greatly simplified.

The program can contain errors at many different levels of syntax error handler

* *Lexical,* such as misspelling an identifier, keyword, or operator
* *Syntactic,* such as an arithmetic expression with unbalanced parentheses
* *Semantic,* such as an operator applied to an incompatible operand
* *Logical,* such as an infinitely recursive call

The error handler in a parser has simple to state goals:

* It should report the presence of errors clearly and accurately.
* It should recover from each error quickly enough to be able to detect subsequent errors.
* It should not significantly slow down the processing of correct programs.

Several parsing methods such as LL and LR methods, detect an error as soon as possible.

**Error - Recovery Strategies:**

There are many different strategies that a parser can recover from a syntactic error.

* Panic mode
* Phrase level
* Error productions
* Global correction

**Panic mode recovery:**

* This is the simplest method to implement and can be used by most parsing methods.
* On discovering an error, parser discards input symbols one at a time until one of the designated set of **synchronizing tokens** is found.
* The synchronizing tokens are usually delimiters such as semicolon or ***end.***
* It skips many inputs without checking additional errors, so it has an advantage of simplicity.
* It guaranteed not to go in to an infinite loop.

**Phrase - level recovery**

* On discovering an error, parser perform local correction on the remaining input;
* It may replace a prefix of the remaining input by some string that allows the parser to continue.
* Local correction would be to *replace a comma by a semicolon, delete an extra semicolon, or insert a* *missing semicolon.*

P a g e | **15** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Error productions**

* Augment the grammar with productions that generate the erroneous constructs.
* The grammar augmented by these error productions to construct a parser.
* If an error production is used by the parser, generate error diagnostics to indicate the erroneous construct recognized the input

**Global correction**

* + Algorithms are used for choosing a minimal sequence of changes to obtain a globally least cost correction.
  + Given an incorrect input string *x* and grammar G, these algorithms will find a parse tree for a related string y; such that the number of *insertions, deletions and changes of tokens* required to transform x

into y is as small as possible.

* + This technique is most costly in terms of time and space

1. **Explain the Context Free Grammar (CFG)? (6 marks)**

A ***Context Free Grammar (CFG)*** consists of terminals, non-terminals, a start symbol and productions.

The (CFG) Grammar G can be represented as G = (V, T, S, P)

* A finite set of terminals (The set of tokens)
* A finite set of non-terminals (syntactic-variables)
* A start symbol (one of the non-terminal symbol)
* A finite set of productions rules in the following form
  + A  , where **A** is a non-terminal and  is a string of terminals and non-terminals including the empty string).
  + Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

**Example:**

The grammar with the following productions defines simple arithmetic expressions. expr  expr op expr

expr  ( expr ) expr  - expr expr  id

op  + op  - op  \* op  / op  ↑

P a g e | **16** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* In this grammar, the terminals symbols are

**id + - \* /** ↑ **( )**

* The non-terminal symbols are ***expr*** and ***op***
* ***expr*** is the start symbol.

**Notational Conventions:**

1. These symbols are terminals:
   1. Lower case letters in the alphabet such as a, b, c.
   2. Operator symbols such as +,-, etc.
   3. Punctuation symbols such as parenthesis, comma, etc.
   4. The digits 0, 1,…….., 9.
   5. Boldface strings such as ***id*** or ***if.***
2. These symbols are non-terminals
   1. Upper case letters in the alphabet such as A, B, C.
   2. The letter S, when it appears, is usually the start symbol.
   3. Lower-case italic names such as ***expr*** or ***stmt***.
3. Upper-case letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*, that is, either non-terminals or terminals.
4. Lower-case letters late in the alphabet u, v, ..., z, represent strings of terminals.
5. Lower-case Greek letters  **,β**, γ represent strings of grammar symbols.
6. If A  1, A2, A3, ……, Ak are all productions with, A on the left (A-productions), write as A1|2|3|…|k the alternatives for A.
7. The left side of the first production is the start symbol.

**Derivations:**

* Derivational view gives a precise description of the top-down construction of a parse tree.
* The central idea is that a production is treated as a rewriting rule in which the non-terminals on the left is replaced by the string on the right side of the production.
* For example, consider the following grammar for arithmetic expressions, with the non-terminals E representing an expression.

**E**  **E + E | E – E | E \* E | (E) | - E | id**

* The production E  - E signifies that an expression preceded by a minus sign is also an expression. This production can be used to generate more complex expressions from simpler expressions by allowing us to replace any instance of an E by - E.

**E**  **-E**

P a g e | **17** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* Given a grammar G with starts symbol, use the + relation to define L (G), the language generated by G.
* Strings in L (G) may contain only terminal symbols of G.
* A string of terminals w is in L (G) if and only if S + w. The string w is called a *sentence of G.*
* A language that can be, generated by a grammar is said to be a ***context-free language.***
* If two grammars generate, the same language, the grammars are said to be *equivalent*.

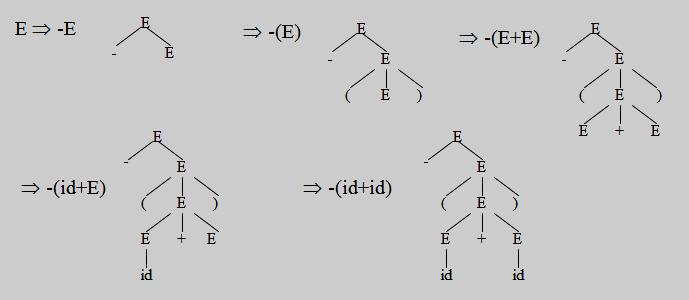
The string – (id+id) is a sentence of the grammar, and then the derivation is

E  -E  - (E)  - (E+E)  - (**id**+E)  - (**id**+**id**)

* At each derivation step, we can choose any of the non-terminals in the sentential form of G for the ***replacement.***
* If we always choose the left-most non-terminal in each derivation step, this derivation is called as ***left-most derivation****.*
* If we always choose the right-most non-terminal in each derivation step, this derivation is called as ***right-most derivation (Canonical derivation).***

**Parse Trees and Derivations:**

* A ***parse tree*** may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order.
* Each interior node of a parse tree is labeled by some non-terminals A, and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this A was replaced in the derivation.
* The leaves of the parse tree are labeled by non-terminals or terminals and read from left to right; they constitute a sentential form, called the ***yield or frontier of the tree.***



P a g e | **18** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Ambiguity**

* A grammar produces more than one parse tree for a sentence is called as an ***ambiguous***.
* An ***ambiguous grammar*** is one that produces more than one left most or more than one right most derivation for the same sentence.
* For the most parsers, the grammar must be unambiguous.
* The ***unambiguous grammar*** unique selection of the parse tree for a sentence.

The sentence **id+id\*id** has the two distinct *leftmost derivations:*

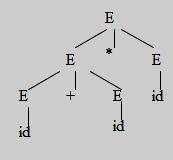
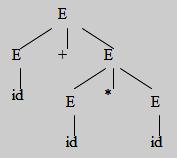
E  E + E

* + id + E
* id + E \* E
* id + id \* E
* id + id \* id

E  E \* E

* E + E \* E
* id + E \* E
* id + id \* E
* id + id \* id

with the two corresponding *parse trees* are



**3. Write the steps in writing a grammar for a programming language. (5 marks)(NOV 2013)**

Grammars are capable of describing the syntax of the programming languages.

***Regular Expressions vs. Context-Free Grammars:***

* Every constructs that can be described by a regular expression can also be described by a grammar.
* For example the regular expression **(a | b)\* abb,** the grammar is:

A0 → aA0 | bA0 | aA1 A1 → bA2

A2 → bA3 A3 → ε

which describe the same language, the set of strings of a’s and b’s ending in abb.

* Mathematically, the NFA is converted into a grammar that generates the same language as recognized by the NFA.

P a g e | **19** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

There are several reasons the regular expressions differ from CFG.

1. The lexical rules of a language are frequently quite simple. No need of any notation as powerful as grammars.
2. Regular expressions generally provide a more concise and easier to understand notation for tokens than grammars.
3. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.
4. Separating the syntactic structure of a language into lexical and non-lexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

* Regular expressions are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords etc…
* Grammars are most useful in describing nested structures such as balanced parenthesis, matching begin - end’s, corresponding if-then-else’s and so on.

**Eliminating Ambiguity:**

* An ambiguous grammar can be rewritten to eliminate the ambiguity.
* Example for eliminate the ambiguity from the following “***dangling-else” grammar:*** stmt  **if** expr **then** stmt

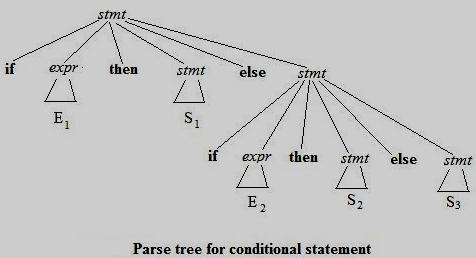
| **if** expr **then** stmt **else** stmt

| **other**

Here **“other”** stands for any other statements. According to this grammar, the compound conditional statement

**i*f*** E1 ***then*** S1 ***else if*** E2 ***then*** S2 ***else*** S3

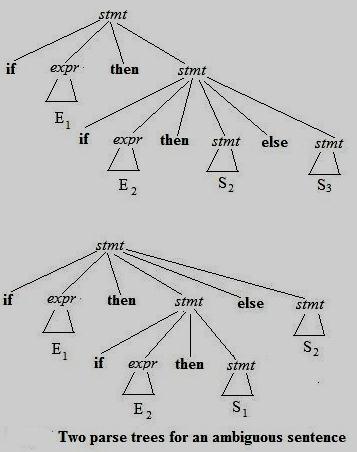
has the parse tree as



P a g e | **20** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

Grammar is ambiguous since the string

***if*** *E1* ***then if*** *E2* ***then*** *S1* ***else*** *S2*has the two parse trees



* In all programming languages with conditional statements of this form, the first parse tree is preferred.
* The general rule is, “Match each ***else*** with the closest previous unmatched ***then***” this disambiguating

rule can be incorporated directly into the grammar. The unambiguous grammar will be:

stmt  matched\_stmt

| unmatched\_stmt

matched\_stmt  ***if*** expr ***then*** matched\_stmt ***else*** matchedstmt | ***other***

unmatched\_stmt  ***if*** expr ***then*** stmt

| ***if*** expr t***hen*** matched\_stmt ***else*** unmatched\_stmt

P a g e | **21** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Elimination of Left Recursion:**

* A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation **A**  **A****.**
* Top-down parsing techniques *cannot* handle left-recursive grammars, so a transformation that eliminates left recursion is needed.
* The left recursive pair of productions **A****A** **| β** could be replaced by non- left- recursive productions

**A****βA’ A’** **A’ | ε**

**Algorithm to eliminating left recursion from a grammar:**

***Input:*** Grammar G with no cycles or ε-productions.

***Output:*** An equivalent grammar with no left recursion.

***Method:*** Note that the resulting non-left-recursive grammar may have ε-productions.

1. Arrange the non-terminals in some order A1, A2, …. An
2. ***for*** i := 1 ***to*** n ***do begin***

***for*** j := 1 ***to*** i-1 ***do begin***

replace each production of the form Ai → Ajγ by the productions Ai → δ1γ | δ2γ | … | δkγ

where Aj → δ1 | δ2 | … | δk are all the current Aj-productions;

**end**

eliminate the immediate left recursion among the Ai-productions

**end**

**Left Factoring:**

* Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.
* The basic idea is that when it is not clear which of two alternative productions to use to expand a non-terminal A, then rewrite the A-productions to defer the decision until the input to make the right

choice.

In general, productions are of the form **A** **→** **αβ1** **| αβ2** then it is left factored as:

**A → αA’**

**A’ → β1 | β2**

P a g e | **22** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Algorithm for Left factoring a grammar**

***Input:*** Grammar G.

***Output:*** An equivalent left-factored grammar.

***Method:*** For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If α ≠ ε,i.e., there is a nontrivial common prefix, replace all the A productions **A** **→** **αβ1** **| αβ2** **| … | αβn** **| γ** where γ represents all alternatives that do not begin with α by,

**A → αA’| γ**

**A’ → β1 | β2 | … | βn**

**4. Briefly write on Parsing techniques. Explain with illustration the designing of a Predictive Parser. (11 marks) (NOV 2013)**

* The top-down parsing and show how to construct an efficient non-backtracking form of top-down parser called ***predictive parser.***
* Define the class ***LL (1) grammars*** from which predictive parsers can be constructed automatically.

**Recursive-Descent Parsing:**

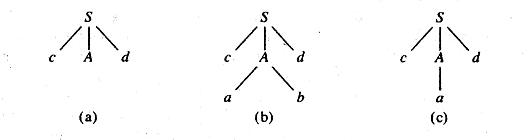
* Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string.
* It is to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
* The special case of recursive-descent parsing called predictive parsing, where no backtracking is required.
* The general form of top-down parsing, called recursive-descent, that may involve backtracking, ie, making repeated scans of input.
* However, backtracking parsers are not seen frequently.
* One reason is that backtracking is rarely needed to parse programming language constructs.
* In natural language parsing, backtracking is still not very efficient and tabular methods such as the dynamic programming algorithm.

Consider the grammar

**S**  **cAd**

**A** **ab | a**

An input string ***w=cad,*** steps in top-down parse are as:



P a g e | **23** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.

**Predictive Parsers:**

* For writing a grammar, eliminating left recursion from it and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a recursive-descent parsing that needs no backtracking

i.e., a ***predictive parser.***

* Flow-of-control constructs in most programming languages, with their distinguishing keywords, are usually detectable in this way.
* For example, if we have the productions *stmt* **if** *expr* **then** *stmt* **else** *stmt*

**| while expr do** *stmt* **| begin** *stmt\_list* **end**

then the keywords ***if, while,*** and ***begin*** that could possibly succeed to find a statement.

**Transition Diagrams for Predictive Parsers:**

Several differences between the transition diagrams for a lexical analyzer and a predictive parser.

* In case of parser, there is one diagram for each non-terminal.
* The labels of edges are tokens and non-terminals.
* A transition on a token means that transition if that token is the next input symbol.
* A transition on a non-terminal A is a call of the procedure for A.

To construct the transition diagram of a predictive parser from a grammar, first eliminate left recursion from the grammar, and then left factor the grammar.

Then for each non-terminal A do the following:

1. Create an initial and final (return) state.
2. For each production A  X1, X2 ... Xn, create a path from the initial to the final state, with edges labeled X1, X2,….,Xn.

**Predictive Parser working:**

* It begins in the start state for the start symbol.
* If after some actions it is in state **s** with an edge labeled by terminal **a** to state **t**, and if the next input symbol is **a**, then the parser moves the input cursor one position right and goes to state **t**.
* If, on the other hand, the edge is labeled by a non-terminal A, the parser instead goes to the start state for A, without moving the input cursor.
* If it ever reaches the final state for **A**, it immediately goes to state **t**, in effect having read **A** from the input during the time it moved from state **s** to **t**.

P a g e | **24** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* Finally, if there is an edge from **s** to **t** labeled **ε**, then from state **s** the parser immediately goes to state **t**, without advancing the input.
* A predictive parsing program based on a transition diagrams attempts to match terminal symbols against the input and makes a potentially recursive procedure call whenever it has to follow an edge

labeled by a non-terminal.

* A non-recursive implementation can be obtained by stacking the states **s** when there is a transition on non-terminal out of **s** and popping the stack when the final state for a non-terminal is reached.
  + Transition diagrams can be simplified by substituting diagrams in one another; these substitutions are similar to the transformations on grammars.

AβA’

A’αA’ | ε

Consider the following grammar for arithmetic expressions,

ET

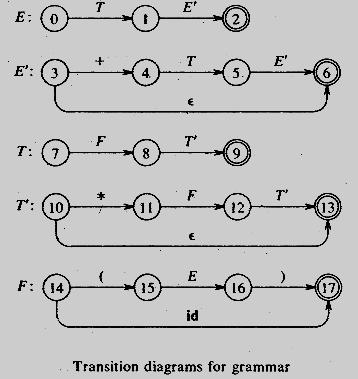
E’+TE’ | ε

TFT ’

T’ \*FT’ | ε

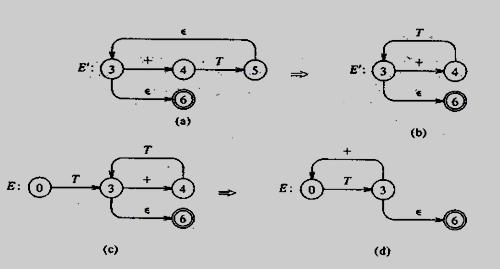
F (E) | id

**Transition diagrams for grammar:**

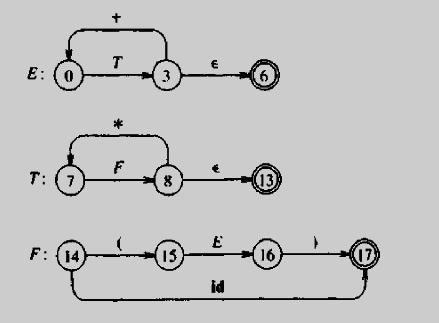


P a g e | **25** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Simplified transition diagram:**



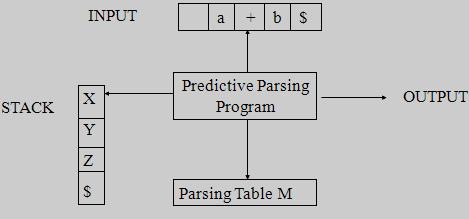
**Simplified transition diagrams for arithmetic expressions:**



P a g e | **26** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**5. Explain the Non-recursive predictive parsing? (11 marks)**

A non recursive predictive parser by maintaining a stack explicitly, rather than implicitly through recursive calls. The key problem during predictive parser is that of determining the production to be applied for a non-terminal.



**Model of a non-recursive predictive parser**

1. table-driven predictive parser has
   * + an input buffer,
     + a stack,
     + a parsing table; and
     + an output stream.
   * The ***input buffer*** contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.
   * The ***stack contains*** a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $.
   * The ***parsing table*** is a two dimensional array M [A, a], where ***A*** is a non-terminal, and ***a*** is a terminal or the symbol $.

The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If ***X = a = $,*** the parser halts and announces successful completion of parsing.
2. If ***X = a*** ***≠*** ***$,*** the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal, the program consults entry M [X, a] of the parsing table M. This entry will be

either an X-production of the grammar or an error entry. For example M [X, a] = {X  UVW}, the parser replaces X on top of the stack by WVU (U on top).

1. If M[X, a] = error, the parser calls an error recovery routine.

P a g e | **27** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Algorithm Non recursive predictive parsing:**

***Input:*** A string w and a parsing table M for grammar G.

***Output:*** If w is in L (G), a leftmost derivation of w; otherwise an error indication.

***Method:*** Initially, the parser is in a configuration in which it has $S on the stack with S, the startsymbol of G on top; and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input.

set ip to point to the first symbol of w$; **repeat**

let X be the top of the stack and a the symbol pointed by ip **if** X is a terminal or **$ then**

i**f** X=a t**hen**

pop X from the stack and advance ip **else** error( )

**else /\*** X is a non-terminal **\*/**

**if** M [X ,a] = XY1Y2…YK **then begin** pop X from the stack;

push YK … …Y2 Y1 on to the stack ,with Y1 on top; output the production X Y1 Y2 …YK

**end**

**else** error( ) **until** X= $ /\* stack is empty \*/

**ALGORITHM FOR FIRST:**

1. If X is terminal, and then FIRST(X) is {X}.
2. If X ε is a production, then add ε to FIRST(X).
3. If X is non-terminal and X  Y1,Y2….Yk is a production, then place a in FIRST(X) if for some i , a is in FIRST(Yi) , and ε is in all of FIRST(Y1),…FIRST(Yi-1);

**ALGORITHM FOR FOLLOW:**

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

P a g e | **28** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**6. Construct the predictive parser for the following grammar E → E + T | T**

**T → T \* F | F F → (E) | id**

**Compute FIRST and FOLLOW and also find the parsing table. The input string is id+id \* id.**

**Solution:**

The given grammar is

E  E + T | T

T  T \* F | F

F  (E) | id

**1. ELIMINATING LEFT RECURSION FROM THE GRAMMAR:**

E  TE’

E’  +TE’ |  T  FT’

T’  \*FT’ |  F  (E) | id

**2. COMPUTATION OF FIRST:**

**FIRST (E)** = FIRST (T) = FIRST (F) = { **(, id** }

**FIRST (E`)** = { **+, ε** }

**FIRST (T)** = FIRST (F) = { **(, id** } **FIRST (T`)** = { **\*, ε** }

**FIRST (F)** = { **(, id** }

**3. COMPUTATION OF FOLLOW:**

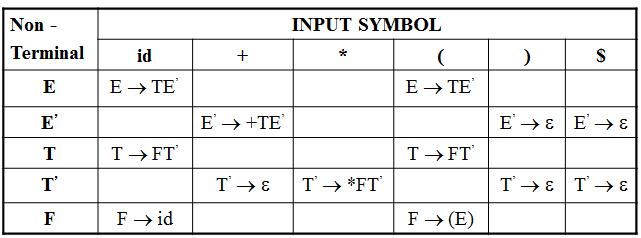
**FOLLOW (E**) = {**$**} **U** FOLLOW (E) = { **), $** } **FOLLOW (E’)** = FOLLOW (E) = { **), $** }

**FOLLOW (T)** = FOLLOW (E’) **U** FIRST (E’)= {), $} **U** {+} = { **+,), $** } **FOLLOW (T’)** = FOLLOW (T) = { **+,), $** }

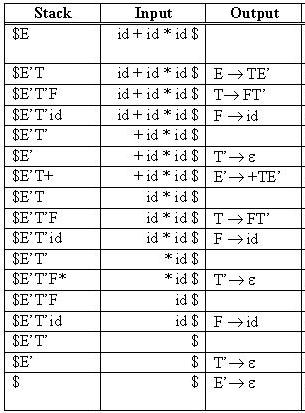
**FOLLOW (F)** =FOLLOW (T’) **U** FIRST (T’) ={+,), $} **U** {\*} = { **+,\*,), $** }

P a g e | **29** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**5. CONSTRUCTION OF PARSING TABLE:**



**6. The Predictive parser on input string is id+id \* id.**



P a g e | **30** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**7. Consider the following LL (1) grammar S → i E t S | i E t S e S | a**

**E → b** **Find the parsing table for the above grammar.**

**Solution:**

The given LL (1) grammar is

S → i E t S | i E t S e S | a

E → b

**1. ELIMINATION OF LEFT FACTORING:**

S  i E t S S’ | a

S’  e S | 

E  b

**2. COMPUTATION OF FIRST: FIRST(S)** = { **i , a** }

**FIRST(S’)** = { **e ,** }

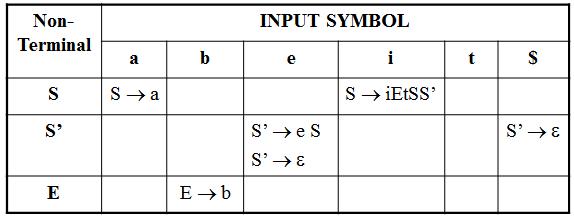
**FIRST (E)** = { **b** }

**3. COMPUTATION OF FOLLOW:**

**FOLLOW(S)** = {**$**} **U** FIRST(S’) = {$} **U** {e} = { **$, e** } **FOLLOW (S’)** = FOLLOW(S) = { **$, e** }

**FOLLOW (E)** = { **t** }

**4. CONSTRUCTION OF PARSING TABLE:**



P a g e | **31** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**8. Explain the LR parsing algorithm in detail. (11 marks)(NOV 2011, 2012)(MAY 2012)**

Bottom-up syntax analysis technique can be used to parse a large class of context-free grammars. The technique is called ***LR (k) parsing.***

* the **"L"** is for left-to-right scanning of the input,
* the **"R"** for constructing a rightmost derivation in reverse, and
* the **k** for the number of input symbols of look ahead that are used in making parsing decisions.
* When (k) is omitted, k is assumed to be 1.

LR parsing is attractive for a variety of reasons.

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general non backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
4. An LR parser can detect a syntactic error as soon as it is possible to do a left-to-right scan of the input.

The principal drawback of the method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. A special tool – an ***LR parser generator***.

Three techniques are used for constructing an LR parsing table for a grammar.

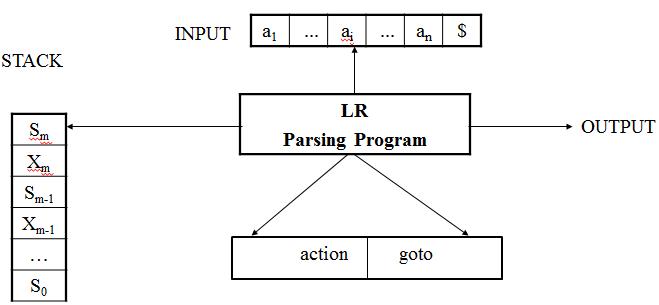
1. The first method, called ***simple LR (SLR),*** is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammars on which the other methods succeed.
2. The second method, called ***canonical LR (CLR),*** is the most powerful, and the most expensive.
3. The third method, called ***look ahead LR (LALR)***, is intermediate in power and cost between the other two. The LALR method will work on most programming language grammars and, with some effort, can be implemented efficiently.

P a g e | **32** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**The LR Parsing Algorithm:**

LR parsing consists of

* an input,
* an output,
* a stack,
* a driver program, and
* a parsing table that has two parts (***action and goto***).



**Model of an LR Parser**

* The driver program is the same for all LR parsers; only the parsing table changes from one parser to another.
* The **parsing program** reads characters from an input buffer one at a time.
* The program uses a **stack** to store a string of the form ***s0X1s1X2s2*** ***... Xmsm***, where, ***sm*** is on top.
* Each ***Xi*** is a grammar symbol and each ***si*** is a symbol called a ***state.***
* Each state symbol summarizes the information contained in, the stack below it, and the combination of the state symbol on top of the' stack and' the current input symbol are used to index the parsing table and determine the shift reduce parsing decision.

The parsing table consists of two parts,

* + - a parsing action function ***action*** and
    - a goto function ***goto***.
* The program driving the LR parser behaves as follows.
* It determines ***sm***, the state currently on top of the stack, and ***ai,*** the current, input symbol.
* It then consults ***action[sm, ai],*** the parsing action table entry for state ***sm*** and input ***ai,*** which can have one of four values:
  1. shift s, where s is a state,
  2. reduce by a grammar production A β
  3. accept, and
  4. error

P a g e | **33** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**LR PARSING ALGORITHM:**

***Input***: An input string ***w*** and an LR parsing table with functions ***action*** and ***goto*** for a grammar G. ***Output***: If w is in L (G), a bottom-up parse for w; otherwise, an error indication.

***Method***: Initially, the parser has ***s0*** on its stack, where ***s0*** is the initial state, and ***w$*** in the input buffer.The parser then executes the program until an accept or error action is encountered.

set *ip* to point to the first symbol of w$; **repeat forever begin**

let s bet the state on the top of the stack and a the symbol pointed to by ip;

**if** action[s,a]=shift s’ **then begin** push a then s’ on top of the stack; advance *ip* to the next input symbol

**end**

**else if** action[s,a]=reduce Aβ **then begin** pop 2\*| β| symbols off the stack;

let s’ be the state now on top of the stack; push A the goto[s’,A] on top of the stack; output the production A β

**end**

**else if** action[s,a]=accept **then return**

**else** *error*()

**end**

P a g e | **34** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**9. Consider the following grammar to construct the SLR parsing table (11 marks) (NOV 2012) E****E+T | T**

**T****T\*F | F F** **(E) | id**

**Construct an LR parsing table for the above grammar. Give the moves of the LR parser on id \* id + id.**

**Solution:**

The given SLR grammar is

1. -> E + T / T
2. -> T \* F / F
3. -> (E) / id

Let the grammar G be

1. -> E + T
2. -> T
3. -> T \* F
4. -> F
5. -> (E)
   1. -> id
6. **The augmented Grammar G’:**

E’ -> E

1. -> E + T
2. -> T
3. -> T \* F
4. -> F
5. -> (E)
   1. -> id
6. **COMPUTATION OF CLOSURE FUNCTION: I0 :**

E’ -> .E

* 1. -> .E + T

E -> .T

T -> . T \* F

T -> .F

F -> .(E)

F -> . id

P a g e | **35** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | |
|  |  | | |  |
|  | **3. COMPUTATION OF GOTO FUNCTION:** | | | |
|  | **goto (I, X)** | | { **where I -> set of states and X -> E, T, F, +, \*, (, ), id** } | |
|  | **goto(I0, E)** |  |  |  |
|  | **I1:** | E’ -> E**.** |  |  |
|  |  | E -> E**.** + T |  |  |
|  | **goto(I0, T)** |  |  |  |
|  | **I2:** | E -> T**.** |  |  |
|  |  | T -> T**.** \* F |  |  |
|  | **goto(I0, F)** |  |  |  |
|  | **I3:** | T -> F**.** |  |  |

**goto(I0, +)** -> NULL **goto(I0, +)** -> NULL

**goto(I0, ( )**

**I4 :** F -> (**.**E)

1. -> **.**E + T
2. -> **.**T
3. -> **.**T \* F
4. -> **.**F
5. -> **.**(E)

F -> **.**id

**goto(I0, ))** -> NULL

**goto(I0, id)**

**I5:** F -> id**.**

Repeat in the new set for the closure function

**goto(I1, +)**

**I6:** E -> E + **.**TT -> **.**T \* F T -> **.**F

F -> **.**(E)

F -> **.**id

P a g e | **36** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**goto(I2, \*)**

**I7:** T -> T \* **.**FF -> **.**(E) F -> **.**id

**goto(I3, X)** -> NULL

**goto(I4, E)**

**I8:** F -> (E **.** )

E -> E **.** + T

**goto(I4, T)**



**I2:** E -> T**.**

T - > T**.**\* E

**goto(I4, F)**



**I3:** T -> F**.**

**goto(I4, ( )**



**I4:** F -> (**.**E)

E -> **.**E + T

E -> **.**T

T -> **.**T \* F

T -> **.**F

F -> **.**(E)

F -> **.**id

**goto(I4, id)**



I5: F -> id**.**

**goto(I6, T)**

**I9:** E -> E + T**.**

T -> T**.** \* F

**goto(I6, F)**



**I3:** T -> F**.**

P a g e | **37** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**goto(I6, ( )**



**I4:** F -> (**.**E)

1. -> **.**E + T
2. -> **.**T
3. -> **.** T \* F
4. -> **.**F
5. -> **.**(E)

F -> **.** id

**goto(I6, id)**



**I5:** F -> id**.**

**goto(I7, F)**

**I10:** T -> T \* F**.**

**goto(I7, ( )**



**I4:** F -> (.E)

1. -> .E + T
2. -> .T
3. -> .T \* F
4. -> .F
5. -> .(E)

F -> .id

**goto(I7, id)**



**I5:** F -> id.

**goto(I8, ) )**

**I11:** F -> (E).

**goto(I8, +)**



**I6:** E -> E + **.**TT -> **.**T \* F T -> **.**F

F -> **.**(E)

F -> **.**id

P a g e | **38** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**goto(I9, \*)**



**I7:** T -> T \***.**FF -> **.**(E)

* + 1. -> **.**id

1. **CONSTRUCTION OF PARSING TABLE:** 
   1. **Shifting Process**

**I0 :** goto(I0, E) = I1goto(I0, T) = I2 goto(I0, F) = I3 goto(I0, ( ) = I4 goto(I0, id) = I5

**I1:** goto(I1, +) = I6

**I2:** goto(I2, \* ) = I7

**I4:** goto(I4, E) = I8goto(I4, T) = I2 goto(I4, F) = I3 goto(I4, ( ) = I4 goto(I4, id) = I5

**I6:** goto(I6, T) = I9goto(I6, F) = I3 goto(I6, ( ) = I4 goto(I6, id) = I5

**I7:** goto(I7, F) = I10goto(I7, ( ) = I4 goto(I7, id) = I5

**I8:** goto(I8, )) = I11goto(I8, +) = I6

**I9:** goto(I9, \* ) = I7

P a g e | **39** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **i) ELIMINATION OF LEFT RECURSION ELIMINATION**

E -> TE’

E’ -> +TE’ / ε

T -> FT’

T’ -> \* FT’ / ε

F -> (E) / id

**ii) COMPUTATION OF FIRST**

**FIRST (E)** = FIRST (T) = FIRST (F) = { **(, id** }

**FIRST (E`)** = { **+, ε** }

**FIRST (T)** = FIRST (F) = { **(, id** } **FIRST (T`)** = { **\*, ε** }

**FIRST (F)** = { **(, id** }

**iii) COMPUTATION OF FOLLOW**

**FOLLOW (E**) = {**$**} **U** FOLLOW (E) = { **), $** } **FOLLOW (E’)** = FOLLOW (E) = { **), $** }

**FOLLOW (T)** = FOLLOW (E’) **U** FIRST (E’)= {), $} **U** {+} = { **+,), $** } **FOLLOW (T’)** = FOLLOW (T) = { **+,), $** }

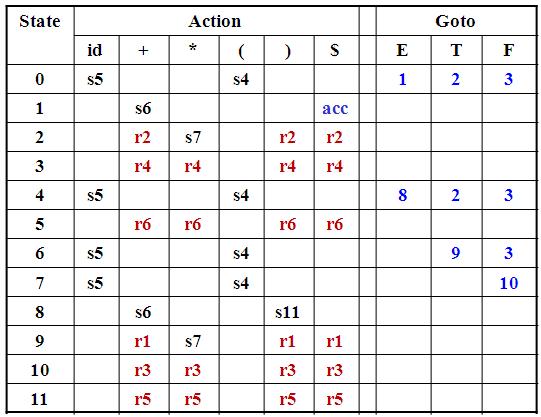
**FOLLOW (F)** =FOLLOW (T’) **U** FIRST (T’) ={+,), $} **U** {\*} = { **+,\*,), $** }

1. **Reducing Process**

|  |  |  |  |
| --- | --- | --- | --- |
| **I2:** | E -> T**.** | **FOLLOW (T)** | = { **+ , ) , $** } |
| **I3:** | T -> F**.** | **FOLLOW (F)** | = { **+ , \* , ) , $** } |
| **I5:** | F -> id**.** | **FOLLOW (F)** | = { **+ , \* , ) , $** } |
| **I9:** | E -> E + T**.** | **FOLLOW (T)** | = { **+ , ) , $** } |
| **I10:** T -> T \* F**.** | | **FOLLOW (F)** | = { **+ , \* , ) , $** } |
| **I11:** | F -> (E)**.** | **FOLLOW (F)** | = {**+ , \* , ) , $** } |

P a g e | **40** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**5. SLR Parsing Tables of Expression Grammar:**



**6. The SLR parser and given input string is id+id.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **STACK** | **I/P STRING** |  | **PARSING ROUTINE** |
|  |  |  |  | |
| (1) | **0** | **id + id $** | action[0, id] = s5 then **shift ‘s5’** | |
|  |  |  | push **id** and **5** into the stack | |
|  |  |  |  | |
| (2) | **0 id 5** | **+ id $** | action [5, +] = r6 then **reduce r6:** | |
|  |  |  | **F -> id** |  |
|  |  |  | 1) | POP **2 symbols** from the stack |
|  |  |  | 2) | goto [0, F] = **3** |
|  |  |  | 3) | Push **‘F3’** into the stack |
|  |  |  |  | |
| (3) | **OF3** | **+id$** | action[3,+] = r4 then **reduce r4:** | |
|  |  |  | **T->F** |  |
|  |  |  | 1) | POP **2 symbols** from the stack |
|  |  |  | 2) | goto [0, T] = **2** |
|  |  |  | **3)** Push **‘T2’** into the stack | |
|  |  |  |  | |
| (4) | **0T2** | **+id$** | action[2,+] = r2 then **reduce r4:** | |
|  |  |  | **E->T** |  |
|  |  |  | 1) | POP **2 symbols** from the stack |
|  |  |  | 2) | goto [0, E] = **1** |
|  |  |  | 3) | Push **‘E1’** into the stack |
|  |  |  |  |  |

P a g e | **41** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | | | | |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  | (5) | **0E1** | **+id$** |  | action[1, +] = s6 then **shift ‘s6’** | |  |  |
|  |  |  |  |  |  | push **+** and **6** into the stack | |  |  |
|  |  |  |  |  |  |  | |  |  |
|  |  | (6) | **0E1+6** | **id$** |  | action[6, id] = s5 then **shift ‘s5’** | |  |  |
|  |  |  |  |  |  | push **id** and **5** into the stack | |  |  |
|  |  |  |  |  |  |  | |  |  |
|  |  | (7) | **0E1+6id5** | **$** |  | action[5,$] = r6 then **reduce r6:** | |  |  |
|  |  |  |  |  |  | **F->id** |  |  |  |
|  |  |  |  |  |  | 1) | POP **2 symbols** from the stack |  |  |
|  |  |  |  |  |  | 2) | goto [6, F] = **3** |  |  |
|  |  |  |  |  |  | 3) | Push **‘F3’** into the stack |  |  |
|  |  |  |  |  |  |  | |  |  |
|  |  | (8) | **0E1+6idF3** | **$** |  | action[3,$] = r4 then **reduce r4:** | |  |  |
|  |  |  |  |  |  | **T->F** |  |  |  |
|  |  |  |  |  |  | 1) | POP **2 symbols** from the stack |  |  |
|  |  |  |  |  |  | 2) | goto [6, T] = **4** |  |  |
|  |  |  |  |  |  | 3) | Push **‘T4’** into the stack |  |  |
|  |  |  |  |  |  |  | |  |  |
|  |  | (9) | **0E1+6T4** | **$** |  | action[4,$] = r1 then **reduce r1:** | |  |  |
|  |  |  |  |  |  | **E->E+T** | |  |  |
|  |  |  |  |  |  | 1) | POP **6 symbols** from the stack |  |  |
|  |  |  |  |  |  | 2) | goto [0, E] = **1** |  |  |
|  |  |  |  |  |  | 3) | Push **‘E1’** into the stack |  |  |
|  |  |  |  |  |  |  | |  |  |
|  |  | (10) | **0E1** | **$** |  | action[1,$] = **acc** | |  |  |
|  |  |  |  |  |  | **The given input string is accepted.** | |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

P a g e | **42** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**10. List out and discuss the different type of intermediate code? (11 marks) (NOV 2012)**

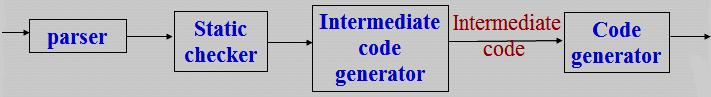
In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code.

A source program can be translated directly into target language, some benefits of using machine-independent intermediate form:

1. Retargeting is facilitated; a compiler for different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

It is used to translates into an intermediate form programming language constructs such as

* Declaration
* Assignment statements
* Boolean Expression
* Flow of control statements



The three kinds of intermediate representations are

1. ***Syntax trees***
2. ***Postfix notation***
3. ***Three - address code***

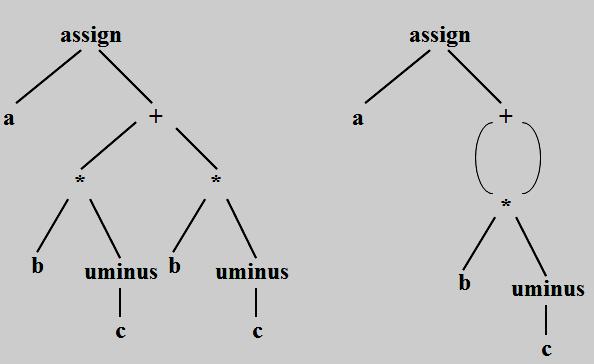
The semantic rules for generating three - address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

**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.

P a g e | **43** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

A syntax tree and DAG for the assignment statement ***a := b \* - c + b \* - c***



**(a) Syntax tree** **(b) DAG**

* ***Postfix notation*** is a linearized representation of a syntax tree; it is a list of the nodes of the in which anode appears immediately after its children.
* The postfix notation for the syntax tree 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 number of operands that the operator at a node expects. The recovery of edges is similar to the evaluation, using a stack, of an expression in postfix notation.

**Syntax tree directed-translation:**

* Syntax trees for assignment statements are produced by the syntax-directed definition.
* Non-terminal S generates an assignment statement.

The syntax-directed definition will produce the dag if the functions

* + ***mkunode(op, child)***
  + ***mknode(op, left, right)***
  + ***mkleaf(id,id.place)***
* 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.
* The symbol table entry can be found from an attribute ***id.name***, representing the lexeme associated with that occurrence of ***id***.

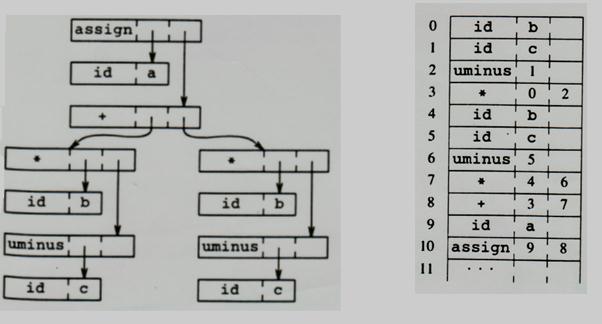
P a g e | **44** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | |
|  |  | |  |  |
|  | **PRODUCTION** | | **SEMANTIC RULES** | |
|  | S  id: = E | | S.nptr := mknode (‘assign’, mkleaf (id, id.entry), 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 := mknode(‘uminus’,E1.nptr) | |
|  | E  (E1) | | E.nptr := E1.nptr | |
|  | E  id |  | E.nptr := mkleaf(id, id.entry) | |

**Syntax-directed definition to produce syntax tress for assignment statements**

**Two way representation of syntax trees:**

* Each node is represented as a record with a field for its operator and additional fields for pointers to its children.
* 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 10.



**Two way representation of syntax trees**

P a g e | **45** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**11. Explain the types of Three-address statements? (6 marks)**

**Three-address code:**

* Three-address code is a sequence of statements of the general form
  1. ***:= y op z***
* where x, y and z are names, constants, or compiler-generated temporaries;

op stands for any operator, such as fixed or floating-point arithmetic operator, or a logical operator on boolean-valued data.

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.

* Three-address code is 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 are represented by the three-address code sequences

The three address codes for the following ***a: = b \* -c + b \* -c***

|  |  |  |  |
| --- | --- | --- | --- |
| t1 | := - c | t1 | := - c |
| t2 := b \* t1 | | t2 := b \* t1 | |
| t3 := - c | | t5 := t2 + t2 | |
| t4 := b \* t3 | | a := t5 | |

t5 := t2 + t4 a := t5

***(a) Code for syntax tree*** ***(b) Code for DAG***

***“Three-address code”*** is that each statement usually contains three addresses, two for the operands and onefor the result.

P a g e | **46** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**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”.

The common three-address statements are

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.
   * The 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.

1. **param x and call p, n** for procedure calls and return y, where y representing a returned value isoptional. 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(x1,, x2,..., xn). The integer n indicating the number of actual parameters in ”call p, n” is not redundant because calls can be nested.

1. **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.
2. **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.

P a g e | **47** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Explain the syntax directed translation into three- address code? (5 marks)** 
   * 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.
   * We create a new name every time a temporary is needed; techniques for reusing temporaries are given.
   * The ***S-attributed definition*** generates three-address code for assignment statements.
   * Given input a: = b\* – c + b\* – c, it produces the code.
   * 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.
   * Use the notation ***gen(x ’: =’ y ’+’ z)*** 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. Three- address statements might be sent to

an output file, rather than built up into the code attributes.

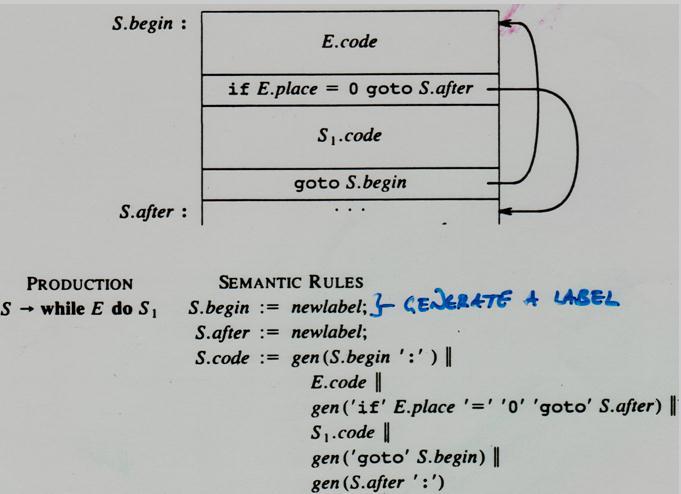
* + Flow-of-control statements can be added to the language of assignments by productions and semantic rules like the ones for while statements.
  + The code for ***S*** ** ***while E do S1***, is generated using new attributes ***S.begin*** and ***S.after*** to mark the first statement in the code for E and the statement following the code for S, respectively.
  + These attributes represent labels created by a function ***newlabel*** that returns a new label every time it is called.
  + ***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.

P a g e | **48** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Syntax directed definition to produce three- address code for assignments:**

|  |  |
| --- | --- |
| **PRODUCTION** | **SEMANTIC RULES** |
|  |  |
| S  id := E | S.code := E.code || **gen**(**id.**place ‘:=’ E.place ) |
|  |  |
| E  E1 + E2 | E.place := newtemp ; |
|  | E.code := E1.code || E2.code || |
|  | **gen**(E.place ‘:=’ E1.place **‘+’** E2.place) |
|  |  |
| E  E1 \* E2 | E.place := newtemp ; |
|  | E.code := E1.code || E2.code || |
|  | **gen**(E.place ‘:=’ E1.place ‘**\*’** E2.place) |
|  |  |
| E  - E1 | E.place:= newtemp ; |
|  | E.code := E1.code || **gen**(E.place ‘:=’ **‘uminus’** E1.place) |
|  |  |
| E  ( E1 ) | E.place:= E1.place ; |
|  | E.code := E1.code |
|  |  |
| E  id | E.place := id.place ; |
|  | E.code := ‘’ |
|  |  |

* Expressions that govern the flow of control may in general be Boolean expressions containing relational and logical operators.
* The semantic rules for while statements to allow for flow of contro1 within Boolean expressions



**Semantic rules generating code for a while statement**

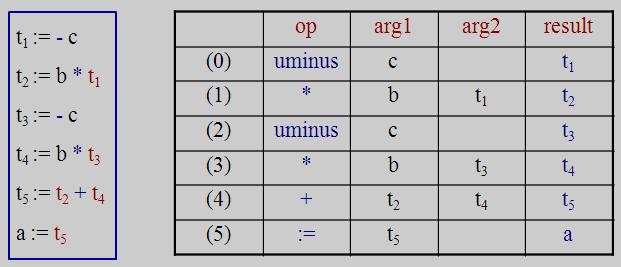
P a g e | **49** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**13. Explain the three implementation of three-address statements? (11 marks)**

**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
    1. quadruples,
    2. triples, and
    3. Indirect triples.

1. **Quadruples:** 
   * A ***quadruple*** is a record structure with four fields, are ***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 for the assignment ***a: = b \* -c + b \* -c***

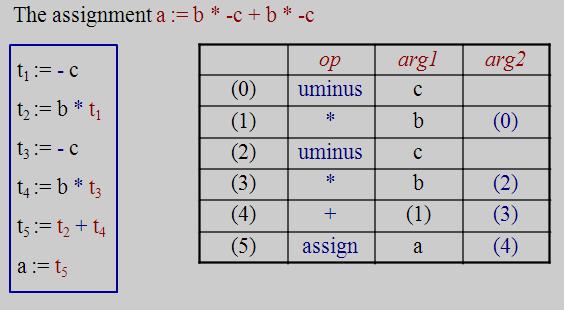


**Quadruples of three address statements**

* The contents of field’s ***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.

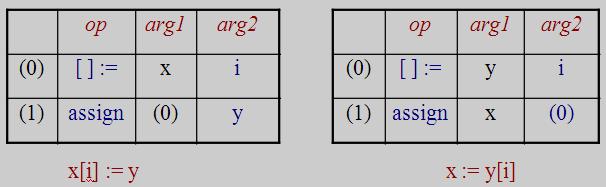
P a g e | **50** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Triples:** 
   * To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
   * Three-address statements can be represented by records with only three fields: ***op, arg 1*** and ***arg2***.
   * The field’s ***arg l*** and ***arg2,*** for the arguments of ***op***, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
   * Since three fields are used, this intermediate code format is known as ***triples.***
   * Triples correspond to the representation of a syntax tree or dag by an array of nodes.



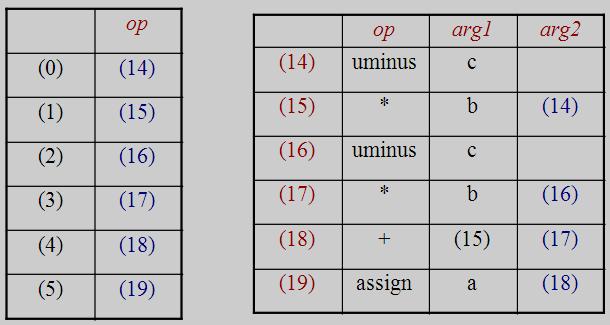
**Triple representation of three – address statements**

* Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves.
* 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 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*** and ***x: = y[i]*** requires two entries in the triple structure



P a g e | **51** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **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.

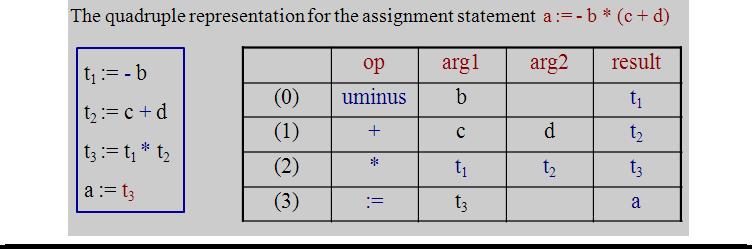


**Indirect triples representation of three-address statements**

**14. Write the quadruple representation for the assignment statement a : = - b \* ( c + d )**

**(5 marks)(MAY 2012)**

* A ***quadruple*** is a record structure with four fields, are ***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***.
* The quadruples representation for the assignment statement ***a : = - b \* ( c + d )***



P a g e | **52** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

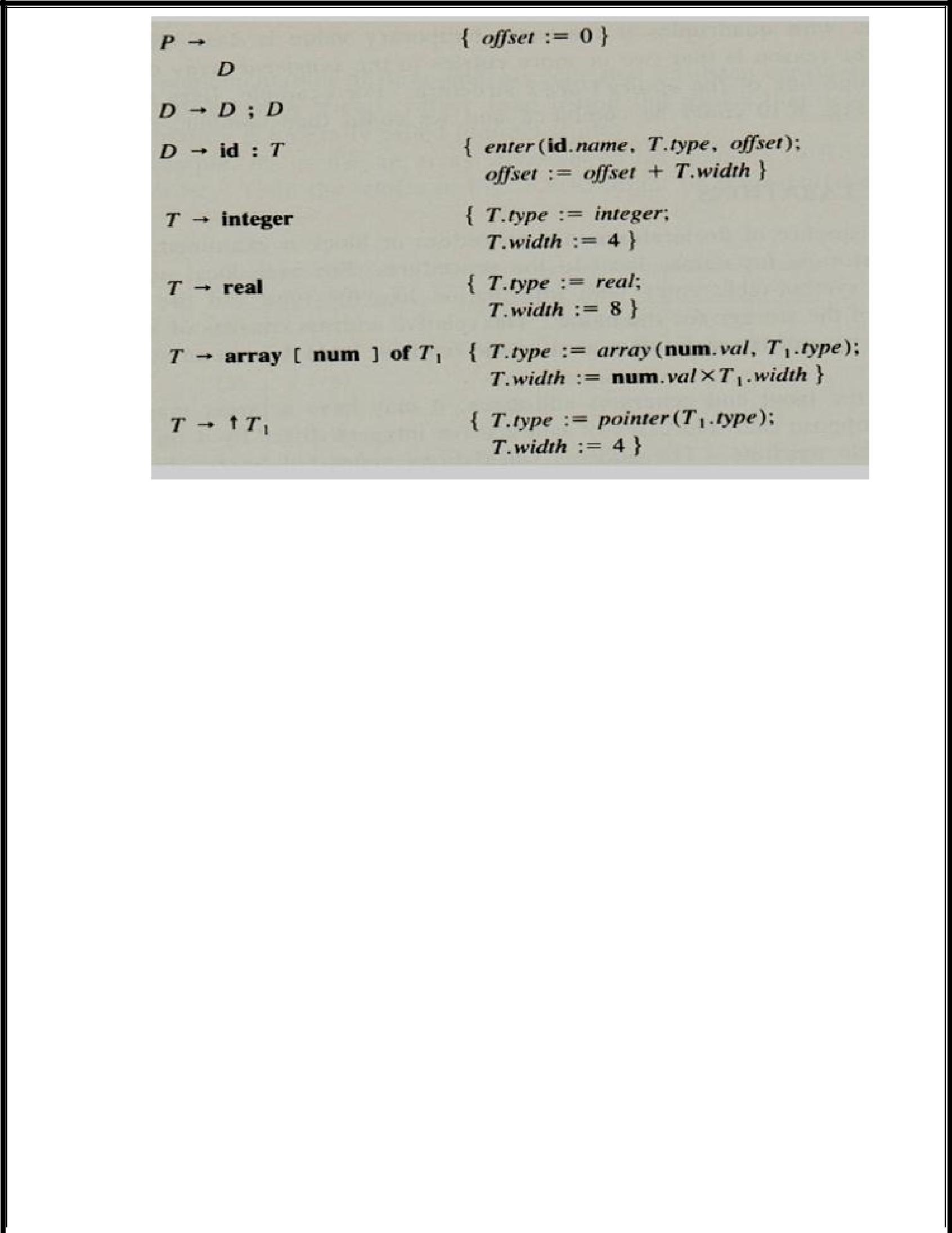
**15. Describe in detail the declarations in a procedure and the methods to keep track of scope information. (11 marks) (NOV 2013)**

* 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.
* 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.

**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 available relative address.
* Non-terminal P generates a sequence of declarations of the form ***id: T***.
* Before the 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 type’s ***integer*** and ***real*** by applying the type constructors’ ***pointer*** and ***array***.
* If type expressions are represented by graphs, then attribute ***type*** might be a pointer to the node representing a type expression.
* ***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 a g e | **53** LANGUAGE TRANSLATORS DEPARTMENT OF CSE



**Computing the types and relative addresses of declared names**

* In Pascal and C, a pointer the type of the object pointed. Storage allocation for such types is simpler if all pointers have the same width.
* The initialization of offset in the translation scheme is the first production appears on one line as:

**P**  **{ offset: = 0 } D**

* Non-terminals generating ε, called marker non-terminals can be used to rewrite productions so that all actions appear at the ends of right sides. Using a marker non-terminal M, can be

**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. When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended.

The 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. The non-terminal T has synthesized attributes type and width. For simplicity, suppose that there is a separate symbol table for each procedure in the language.

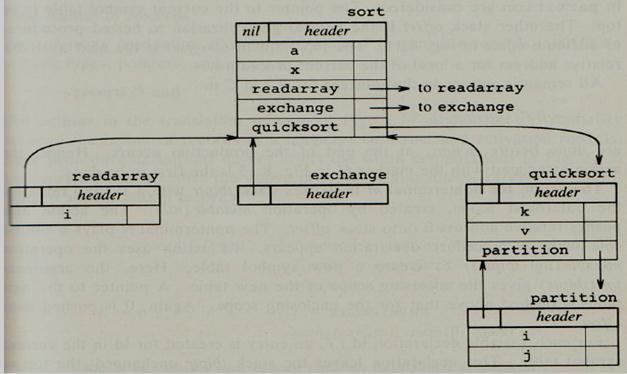
P a g e | **54** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* A new symbol table is created when a procedure declarations **D** **proc id D1**; **S** and entries for the declarations in D1 in a new symbol 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.

For example, the symbol tables for five procedures.

* The symbol tables for procedures ***readarra***y, ***exchange*** and ***quick sort*** point back and containing ***procedure sort***, consisting of the entire program.
* The ***partition*** is declared with ***quick sort, its*** table point to that of ***quick sort.***

**Symbol tables for nested procedures:**



P a g e | **55** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

The Semantic rules for operations:

1. ***mktable (previous)*** 
   * It 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)*** 
   * + It creates a new entry for name ***name*** in the symbol table pointed to by table.
     + Again, ***enter*** places type ***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 headerassociated with this symbol table.
4. ***enterproc (table, name, newtable)*** 
   * + It 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 shows how data can be in one pass, using a stack ***tblptr*** to hold pointers to symbol tables of the enclosing procedures.
* With the symbol tables to ***tblptr*** will contain pointers to the tables for ***sort, quicksort,*** and ***partition*** when the declarations in partition are considered.
* The pointer to the current symbol table is on top.
* The other stack ***offse***t is the natural generalization to nested procedures of attribute ***offset.***
* The top element of ***offset*** is the next available relative address for a local of the current procedure.
* 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 N plays a similar role when a procedure declaration appears.
* Its action uses the operation ***mktable(top(tblptr))*** to create a new symbol table.
* 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***.

P a g e | **56** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Processing declarations in nested procedures**

**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**

**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) }

P a g e | **57** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Explain in detail about assignment statements? (11 marks)** 
   * Expressions can be of type integer, real, array and record fields.
   * The translation of assignment into three- address code that names can be looked up in the symbol table and how elements of array and records can be accessed.

**Names in the Symbol Table:**

* Three address statements using names for pointers to their symbol table entries.
* The lexeme for the name represented by **id**, the attribute as ***id.name***
* The operation ***lookup (id.name)***, check if there is an entry for the occurrence of the name in the symbol table.
* If so, a pointer of the entry is returned.
* Otherwise returns ***nil*** to indicate that no entry is found.

The semantic action use procedure ***emit*** to 3 address statements to an output file, code attributes for non-terminals.

**S****id: = E**

* The non-terminal ***S*** represents the **name** modified **lookup** operation first checks if name appears in the current symbol table, accessible through **table pointer**.
* If not, lookup uses the pointer in the header of a table to find the symbol table.
* If the name cannot be found, then **lookup** returns **nil.**

**Translation scheme to produce three-address code for assignments**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **S → id: = E** | { p := lookup(id.name); | |
|  |  | **if** p != nil **then** | |
|  |  | emit(p ′ :=′ E.place) | |
|  |  | **else** error } | |
|  | **E → E1 + E2** | { E. place := newtemp; | |
|  |  | emit(E.place ′ :=′ E1.place ′ +′ E2.place) } | |
|  | **E → E1** ∗ **E2** | { E. placer:= newtemp; | |
|  |  | emit(E.place ′ :=′ E1.place ′ \*′ E2.place) } | |
|  | **E → −E1** | { E. place := newtemp; | |
|  |  | emit(E.place ′ :=′ ′uminus′ E1.place) } | |
|  |  |  |  |
|  | P a g e | **58** LANGUAGE TRANSLATORS | DEPARTMENT OF CSE | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | |
|  |  |  |  |  |
|  | **E → (E1)** | { | E.place := E1.place } | |
|  | **E → id** | { | p := lookup(**id**.name); | |
|  |  |  | **if** p != nil **then** | |
|  |  |  | E.place := p | |
|  |  |  | **else** error } | |
|  | **Reusing Temporary Names:** |  |  |  |

* The ***newtemp*** generates a new temporary name each time temporary is needed.
* The temporaries used to hold intermediate values in expression calculations for the symbol table and space has to be allocated to hold their values.
* Temporaries can be reused by changing ***newtemp.***
* The temporary data are generated during the syntax directed translation of expression.
* The code generated by the rules is ***E********E1+E2*** of the form
  + evaluate E1 into t1
  + evaluate E2 into t2
  + t := t1 + t2

Consider the assignment statements **x: = a\*b + c\*d** **–** **e\*f**

|  |  |  |  |
| --- | --- | --- | --- |
| **STATEMENTS** | |  | **VALUE OF C** |
|  |  |  | **0** |
| **$0 := a \* b ;** | | 1 | c incremented by 1 |
| **$1** | **:= c \* d ;** | 2 | c incremented by 1 |
| **$0** | **:= $0 + $1 ;** | 1 | c decremented twice, incremented once |
| **$1** | **:= e \* f ;** | 2 | c incremented by 1 |
| **$0** | **:= $0 - $1 ;** | 1 | c decremented twice, incremented once |
| **x := $0 ;** | | 0 | c decremented once |

* A count c, initialized to zero.
* Whenever a temporary name is used as an operand, decrement c by 1.
* Whenever a new temporary name is created, use $c and increment c by 1.

P a g e | **59** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Type Conversion with Assignments:**

* There are different types of variables and constants, so the compiler must either reject certain mixed-type operations or generate the type conversion instructions.
* Consider the grammar for assignment statements, there are two types –***real and integer***, with integers converted to reals when necessary.
* An attribute **E.type** holds the type of an expression which is either ***real or integer.***

The semantic rule for the ***E.type*** production EE+E is:

EE+E { E.type :=

***if*** E1.type = integer ***and***

E2.type = integer ***then*** integer

**else** real }

For example, for the input

**x := y + i \* j**

Assuming ***x and y*** have type ***real*** and ***i and j*** have type ***integer,*** the output as **t1 := i int\* j**

**t2 := inttoreal t1 t3:= y real+ t2 x := t3**

**Semantic action for** EE+EE.place := newtemp;

**if** E1.type = integer and E2.type = integer **then begin emit**(E.place ':=' E1.place 'int+' E2.place);

***E.type := integer***

**end**

e**lse** if E1.type = real and E2.type = real **then begin** **emit**(E.place ':=' E1.place ‘real+' E2.place);

***E.type := real***

**end**

**else if** E1.type = integer and E2.type = real **then begin** u := newtemp;

**emit**(u ':=' 'inttoreal' E1.place); **emi**t(E.place ':=' u 'real+' E2.place);

***E.type := real***

**end**

P a g e | **60** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**else if** E1.type = real and E2.type = integer **then begin** u := newtemp;

**emit**(u ':=' 'inttoreal' E2.place); **emit**(E.place ':=E1.place 'real+' u);

***E.type := real***

**end else**

***E.type := type\_error;***

**Accessing Fields in Records:**

* The compiler must keep track of both the types and relative addresses of the fields of a record.
* The information in the symbol table entries for the field names that looking up names tin the symbol

table can be used as field names.

***pointer (record (t))*** or ***p*** ↑ ***.info + 1***

* + The ***type of p*** is the record (t), from which t can be extracted.
  + The ***name info*** field lookup in the symbol table pointed to by t.

1. **State and write the semantic rules for Boolean expressions. (11 marks)(MAY 2013)**

In programming languages, Boolean expressions have two primary purposes

* 1. They are used to compute logical values.
  2. They are used as conditional expressions in statements that alter the flow of control, such as if-then, if-then-else, or while-do statements.
* Boolean expressions are composed of the boolean operators (**and, or,** and **not**) applied to elements that are boolean variables or relational expressions.
* Relational expression of the form **E1 relop E2,** where E1 and E2 arithmetic expressions.
* Consider boolean expressions with the following grammar:

E  E ***or*** E | E ***and*** E | ***not*** E | (E) | id ***relop*** id | ***true*** | ***false***

* We use the attribute ***op*** to determine which of the comparison operators **<, <=, =, !=, >, or >=** is represented by ***relop.***
* Assume that ‘***or’*** and ***‘and’*** are ***left-associative***, and that ***or*** has ***lowest precedence***, then ***and,*** then ***not.***

P a g e | **61** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Methods of Translating Boolean Expression:**

There are two principal methods of representing the value of a Boolean expression.

* + 1. The first method is to encode true and false numerically and to evaluate a boolean expression analogously to an arithmetic expression.
    2. The second principal method of implementing boolean expression is by flow of control that is representing the value of a Boolean expression by a position reached in a program. This method is implementing the Boolean expressions in flow of control statements, such as the if-then and while-do statements.

1. **Numerical Representation:** 
   * Consider the implementation of boolean expression using ***1 to denote true*** and ***0 to denote false***.

 For example, expressions as **a or b and not c**

The translation for 3 address sequence is

**t1 := not c t2 := b and t1 t3 := a or t2**

A relational expression such as a<b is equivalent to the conditional statement if a<b then 1 else 0, which can be translated into the three - address code sequence.

100 : if a < b goto 103

101 : t := 0

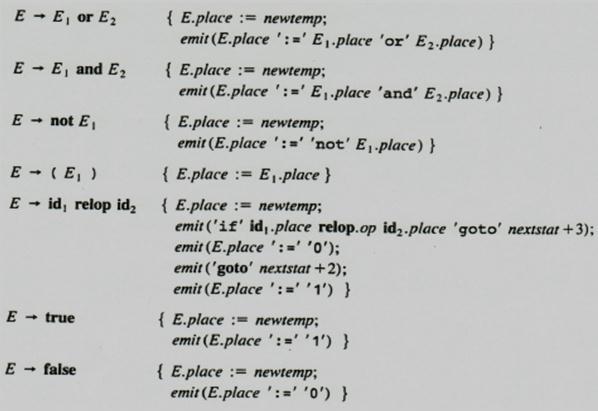
102 : goto 104

103 : t := 1

104 :

P a g e | **62** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**A translation scheme for producing three-address code for boolean expression**



* + We assume that ***emit*** places three -address statements into output file in the right format.
  + The ***nextstat*** that gives the index of the next three-address statement in the output sequence and ***emit*** increments ***nextstat*** after producing each three-address statement.
  + We use the attribute **op** to determine which of the comparison operators is represented by ***relop.***

1. **Shot-Circuit or jumping code:** 
   * Translate a Boolean expression into three-address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called ***“short-circuit” or “jumping” code.***
   * It is possible to evaluate boolean expressions without generating code for the boolean operators ***and,*** ***or*** and ***not***, the value of an expression by a position in the code sequence.

P a g e | **63** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Translation of a<b or c<d and e<f**

The three address code as

* 1. ***if*** a< b goto ***103***
  2. t1 := 0
  3. goto ***104***
  4. t1 := 1
  5. ***if*** c< d goto ***107***
  6. t2 := 0
  7. goto ***108***
  8. t2 := 1
  9. ***if*** e< f goto ***111***
  10. t3 := 0
  11. goto ***112***
  12. t3 := 1
  13. t4 := t2 ***and*** t3
  14. t5 := t1 ***or*** t4

1. **Flow-of-Control Statements:**

Consider the translation of boolean expressions into three-address code as if-then, if-then-else, and while –do statements such those generated by the following grammar

**S → if E then S1**

**| if E then S1 else S2 | while E do S1**

In the translation, we assume that a three-address code statement can have a symbolic label, and that the function newlabel generates such labels.

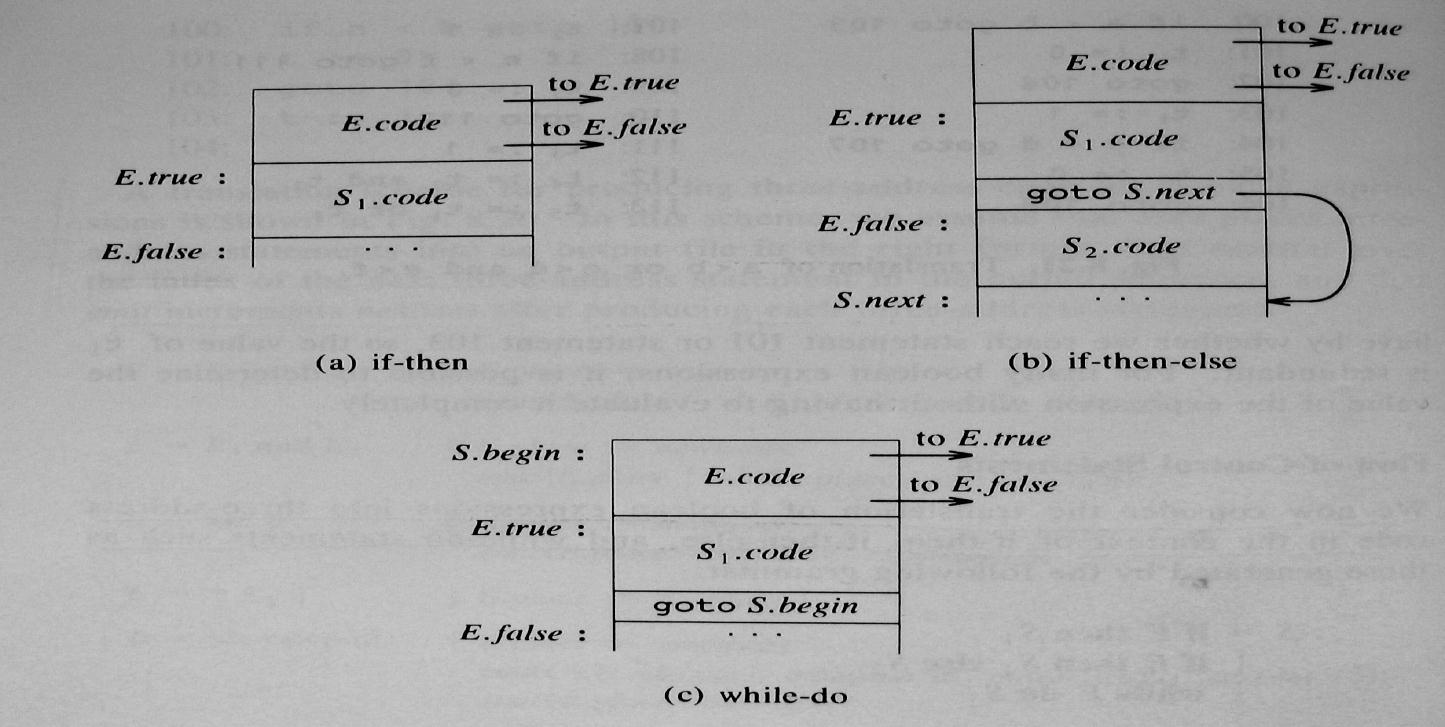
With a boolean expression E, we associate two labels:

* ***E.true,*** the label to which control flows if E is true.
* ***E.false,*** the label to which control flows if E is false.

We associate to S the inherited attribute ***S.next*** that represents the label attached to the first statement after the code for S.

P a g e | **64** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Code for if-then, if-then-else, and while-do statements:**



**Syntax-directed definition for flow-of control statements:**

|  |  |  |
| --- | --- | --- |
| **PRODUCTION** | **SEMANTIC RULES** |  |
|  |  |  |
|  | E.true := newlabel; |  |
| **S → if E then S1** | E.false := S.next; |  |
| S1.next := S.next; |  |
|  |  |
|  | S.code := E.code || gen(E.true ‘ :’) || S1.code |  |
|  |  |  |
|  | E.true := newlabel; |  |
|  | E.false := newlabel; |  |
|  | S1.next := S.next; |  |
| **S → if E then S1 else S2** | S2.next := S.next; |  |
|  | S.code := E.code || gen(E.true ‘:’) || S1.code || |  |
|  | gen(‘goto’ S.next) || |  |
|  | gen(E.false ‘ :’) || S2.code |  |
|  |  |  |
|  | S.begin := newlabel; |  |
|  | E.true := newlabel; |  |
|  | E.false := S.next; |  |
| **S → while E do S1** | S1.next := S.begin; |  |
|  | S.code := gen(S.begin ′ :′) || E.code || |  |
|  | gen(E.true ′ :′) || S1.code || |  |
|  | gen(′goto′ S.begin) |  |
|  |  |  |

P a g e | **65** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**(iv) Control Flow Translation of Boolean Expression:**

Boolean Expressions are translated in a sequence of conditional and unconditional jumps to either ***E.true*** or ***E.false.***

* ***E.true***, the place control is to reach if ***E is true*** and
* ***E.false,*** the place control is to reach if ***E is false.***

The expression E is of the form a < b. The generated code is of the form ***if a < b then goto E.true***

***goto E.false***

Suppose E is of the form **E** **E1** **or E2.**

* If E1 is true, then E is true, so E1.true = E.true.
* If E1 is false, then E2 must be evaluated, so E1.false is set to the label of the first statement in the code for E2.

**Syntax-directed definition to produce three-address code for booleans:**

|  |  |  |
| --- | --- | --- |
| **PRODUCTION** | **SEMANTIC RULES** |  |
|  |  |  |
|  | E1.true := E.true; |  |
|  | E1.false := newlabel; |  |
| **E → E1 or E2** | E2.true := E.true; |  |
|  | E2.false := E.false; |  |
|  | E.code := E1.code || gen(E1.false ′ :′) || E2.code |  |
|  |  |  |
|  | E1.true := newlabel; |  |
|  | E1.false := E.false; |  |
| **E → E1 and E2** | E2.true := E.true; |  |
|  | E2.false := E.false; |  |
|  | E.code := E1.code || gen(E1.true ′ :′) || E2.code |  |
|  |  |  |
|  | E1.true := E.false; |  |
| **E → not E1** | E1.false := E.true; |  |
|  | E.code := E1.code |  |
|  |  |  |
|  | E1.true := E.true; |  |
| **E → (E1)** | E1.false := E.false; |  |
|  | E.code := E1.code |  |
|  |  |  |
| **E → id1 relop id2** | E.code := gen(′if′ **id1.place** relop.op **id2.place** ′goto′ E.true) |  |
| || gen(′goto′ E.false) |  |
|  |  |
|  |  |  |
| **E → true** | E.code := gen(′goto′ E.true) |  |
|  |  |  |
| **E → false** | E.code := gen(′goto′ E.false) |  |
|  |  |  |

P a g e | **66** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Mixed mode Boolean expression:** 
   * Boolean Expressions often contain arithmetic sub-expressions as in (a + b) <c.
   * In languages where false has the numerical value 0 and true the value 1, then (a<b) + (b<a) can be an arithmetic expression with value 0 if a and b have the same value and 1 otherwise.

Consider the following Grammar:

**E → E+E | E and E | E relop E | id**

* ***E + E,*** produces an arithmetic result, and the arguments can be mixed; while expressions ***E and*** E, and ***E relop E,*** produces boolean values represented by flow of control.
* Expression ***E and E*** requires both arguments to be boolean, but the operations ***+*** and ***relop*** take either type of argument, including mixed ones.
* **E→id** is assumed of type arithmetic.
* To generate code we use a synthesized attribute ***E.type*** that will be either ***arith or bool.***
* E will have inherited attributes ***E.true*** and ***E.false*** for boolean expressions and synthesized attribute E.place for arithmetic Expressions. useful for the jumping code.

The semantic rule for **E** **→** **E1+E2** E.type := arith;

i**f** E1.type := arith and E2.type := arith **then begin** E.place := newtemp;

E.code := E1.code || E2.code ||

gen(E.place′ :=′ E1.place ′ +′ E2.place)

**end**

**esle if** E1.type := arith and E2.type := bool **then begin** E.place:= newtemp;

E2.true := newlabel; E2.false := newlabel;

E.code := E1.code || E2.code ||

gen(E2.true′ :′ E.place ′ :=′ E1.place + 1) || gen(′goto′ nextstat + 1) ||

gen(E2.false′ :′ E.place ′ :=′ E1.place)

P a g e | **67** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**18. Explain the case statements? (5 marks)**

The ***“switch” or “case”*** statement is available in various languages; the FORTRAN computed and assigns goto’s for switch statements.

The switch-statement syntax is **switch** expression

**begin**

**case** value: statement **case** value: statement

...

**case** value: statement **default:** statement

**end**

There is a selector expression, which is to be evaluated, followed by n constant values that the expression, including a ***default “value”,*** which always the expression if no other values does.

The translation of a switch code is

1. Evaluate the expression.
2. Find which value in the list of cases is same as the value of expression. The default value matches the expression if none of the values explicitly.
3. Execute the statement associated with the value found.

* To implement a sequence of conditional **goto’s** is to create a table of pair, each pair consisting of a value and a label for the code of the corresponding statement.
* A compiler to compare the value of expression with each value in the table.
* If no other match is found, the last (default) entry is sure to match.

**Syntax directed translation of case statements:**

Consider the following switch statement **switch** E

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **begin** |  |  |  |  |
|  | **case** V1 | : | S1 |  |  |
|  | **case** V2 | : | S2 |  |  |
| ... | |  |  |  |  |
|  | **case** Vn-1: | | Sn-1 |  |  |
|  | **default** | : | Sn |  |  |
|  | **end** |  |  |  |  |
|  |  | | |  |  |
|  | P a g e | **68** LANGUAGE TRANSLATORS | | | DEPARTMENT OF CSE | |

To translate in the form the keyword **switch** generate 2 labels

* ***Test*** and ***next***
* A new temporary variable ***t.***
* Then parse the expression E, generate code to evaluate E into t.
* After processing E, generate the jump ***goto test.***

To translate in the form the keyword **case**

* Create a new label **Li** and enter into the symbol table.
* We place on a stack, used only to store cases, a pointer to symbol table entry and value **Vi** of the case constant.
* Each statement ***case Vi*** ***: Si,*** creates label **Li,** followed by code for **Si,** followed by jump ***goto next.***
* When the keyword ***end*** terminate the body of switch statement.

**Translation of case statement:**

code to evaluate ***E into t***

**goto test**

**L1:** code for S1

**goto next**

**L2:** code for S2

**goto next**

...........

**L n-1:** code for Sn-1

**goto next**

**Ln:** code for Sn

**goto next**

**test:** if t = V1 goto L1

if t = V2 goto L2

……….

if t = Vn-1 goto Ln-1

**goto Ln**

**next:**

P a g e | **69** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Another Translation of case statement**

Code to evaluate ***E into t***

if t ≠ V1 goto L1

code for S1

**goto next**

**L1:** if t ≠ V2 goto L2

code for S2

**goto next**

**L2:** ...

**Ln-2:** if t ≠ Vn-1 goto Ln-1

code for Sn-1

**goto next**

**Ln:** code for Sn

**next:**

**19. How the code is generated for procedure calls? (5 marks) (NOV 2011)**

* The procedure is such an important and frequently used programming construct that is imperative for a compiler to generate good code for procedure calls and returns.
* The run-time routines that handle procedure argument passing, calls, and returns are part of the run-time support package.

Consider a grammar for a simple procedure call statement

* 1. S **call id** (Elist)
  2. Elist Elist, E
  3. Elist  E

**Calling sequences:**

* When a procedure call occurs, space must be allocated for the activation record of the called procedure.
* The arguments of the called procedure must be evaluated and available to the called procedure in known place.
* Environment pointers must be established to enable the called procedure to access data in enclosing blocks.

P a g e | **70** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* The state of the calling procedure must be saved so it can resume execution after the call.
* Also saved in a known place is the return address, location to which the called routine must transfer after it is finished.
* The return address is usually the location of the instruction that follows that call for the calling procedure.
* Finally, a jump to the beginning of the code for the called procedure must be generated.

**Return Statements:**

* When a procedure returns, several actions also must take place.
* If the called procedure is a function, the result must be stored in a known place.
* The activation record of the calling procedure must be restored.
* A jump to the calling procedure's return address must be generated.
* No exact division of run-time tasks between the calling and called procedure.

**Translation includes**

* Calling sequence  actions taken on entry to and exit from each procedure.
* Arguments are evaluated and put in a known places(return address) location to which the called routine must transfer after it is finished.
* Static allocation return address is placed after code sequence itself.
* Parameters passed by reference.
* 3 address code generates statements needed to evaluate those arguments that are simple names then the list.

**For separate evaluation:**

* Save E.place for each expression E in id (E, E, E, E,..)
* Data structure used is queue.

**Semantics:**

1. S call id (Elist)

{ **for** each item p on queue **do**

emit (param p);

emit (‘call’ **id.**place); }

2) Elist Elist, E

{ append E.place to end of queue }

3) Elist  E

{ initialize queue to contain only E.place }

***Queue*** is emptied & single pointer is given to symbol table denoting value of E.

P a g e | **71** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **How back patching can be used to generate code for Boolean expressions? (11 marks) (NOV 2011)** 
   * To implementing syntax-directed definitions, to use two passes.
   * The main problem with generating codes for Boolean expressions and flow of control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time jump statements are generated.
   * By generating a series of branching statement with the targets of the jumps left unspecified.
   * Each such statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. This subsequent filling of addresses for the determined labels is called

***Back patching.***

* + Back patching can be used to generate code for Boolean expression and flow of control statements in one pass.

To generate quadruples into a quadruple array and labels are indices to this array. To manipulate list if labels, we use three functions:

* 1. **makelist(i)** -- creates a new list containing only i, an index into the array of quadruples; ***makelist*** returns a pointer to the list it has made.
  2. **merge(p1,p2) –** concatenates the lists pointed to by p1and p2,and returns a pointer to theconcatenated list.
  3. **backpatch(p,i)** –inserts i as the target label for each of the statements on the list pointed to by p.

1. **Boolean Expressions:**

Construct a translation scheme for producing quadruples for Boolean expressions during bottom-up parsing. We insert a marker non-terminal M into the grammar, the index of the next quadruple to be generated.

The grammar is:

E → E1 **or** M E2

E → E1 a**nd** M E2

E → **not** E1

E → (E1)

E → id1 **relop** id2

E → **false**

E → t**rue**

M → **ε**

P a g e | **72** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* Synthesized attributes ***truelist*** and ***falselist*** of non-terminal E are used to generate jumping code for Boolean expressions.
* ***E.truelist :*** Contains the list of all the jump statements left incomplete to be filled by the label for thestart of the code for ***E:=true.***
* ***E.falselist*** : Contains the list of all the jump statements left incomplete to be filled by the label for thestart of the code for ***E:=false.***
* The variable ***nextquad*** holds the index of the next quadruple to follow.
* ***M.quad*** represents records the number of first statement (index).Consider the production **E** **→** **E1** **and M E2 .**

The semantic actions as

|  |  |  |  |
| --- | --- | --- | --- |
| **PRODUCTION** | **SEMANTIC RULES** | |  |
|  |  | |  |
|  | { backpatch(E1.falselist, M.quad) | |  |
| **E → E1 or M E2** | E.truelist = merge(E1.truelist, E2.truelist) | |  |
|  | E.falselist = E2.falselist | } |  |
|  |  | |  |
|  | { backpatch(E1.truelist, M.quad) | |  |
| **E → E1 and M E2** | E.truelist = E2.truelist |  |  |
|  | E.falselist = merge(E1.falselist, E2.falselist) } | |  |
|  |  |  |  |
| **E → not E1** | E.truelist = E1.falselist |  |  |
| E.falselist = E1.truelist |  |  |
|  |  |  |
|  |  |  |  |
| **E → (E1)** | E.truelist = E1.truelist |  |  |
| E.falselist = E1.falselist |  |  |
|  |  |  |
|  |  | |  |
|  | E.truelist = makelist(nextquad)E.falselist = | |  |
| **E → id1 relop id2** | makelist(nextquad +1 ) |  |  |
| emit(if **id1.**place **relop.op id2.**place goto \_\_ ) | |  |
|  |  |
|  | emit(goto \_\_\_) |  |  |
|  |  | |  |
| **E → true** | E.truelist = makelist(nextquad) | |  |
| emit(goto \_\_\_) |  |  |
|  |  |  |
|  |  | |  |
| **E → false** | E.falselist = makelist(nextquad) | |  |
| emit(goto \_\_\_) |  |  |
|  |  |  |
|  |  |  |  |
| **M → ε** | M.Quad = nextquad |  |  |
|  |  |  |  |

P a g e | **73** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Flow-of –Control Statements:** 
   * Backpatching can be used to translate flow-of-control statements in one pass. Translation scheme for statements generated by the following grammar:

**S→ if E then S**

**| if E then S else S | while E do S**

**| begin L end | A**

**L → L;S**

**| S**

Here ***S*** denotes a statement, ***L*** a statement list, ***A*** an assignment statement, and ***E*** a boolean expression.

**Scheme to implement the Translation:**

**(1) S**  **if E then M S1**

{ backpatch (E.truelist , M.quad);

S.nextlist := mergelist (E.falselist , S1.nextlist) }

1. **S**  **if E then M1 S1 N else M2 S2**

{ backpatch (E.truelist , M1.quad);

backpatch (E.falselist , M2.quad);

S.nextlist := mergelist (S1..nextlist, mergelist (N.nextlist , S2.nextlist)) }

We backpatch the jumps when E is true to the quadruple M1.quad, which is the beginning of the code for S1. Similarly, we backpatch when E is false to go to the beginning of the code for S2.The list S.nextlist includes all jumps out of S1 and S2, as well as the jump generated by N.

1. **S** **while M1 E do M2 S1**

{ backpatch (S1.nextlist , M1.quad); backpatch (E.truelist , M2.quad);

S.nextlist := E.falselist

emit('goto' M1.quad) }

**(4) S****begin L end**

{ S.nextlist := L.nextlist }

P a g e | **74** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**(5) S** **A**

{ S.nextlist := nil }

**(6) L****L1 M S**

{ backpatch(L1.nextlist , M.quad);

L.nextlist := S.nextlist }

**(7) N****ε**

{ N.nextlist := makelist (nextquad); emit(‘goto\_’) }

**(8) M****ε**

{ M.quad := nextquad }

P a g e | **75** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**UNIVERSITY QUESTIONS**

* 1. **MARKS**

1. What is the use of context free grammar? **(NOV 2011) (Ref.Qn.No.7, Pg.no.3)**
2. Draw the dag for the assignment statement: a: = b \* -c + b \* -c **(NOV 2011) (Ref.Qn.No.38, Pg.no.8)**
3. Define Ambiguous.**(MAY 2012) (Ref.Qn.No.12, Pg.no.4)**
4. What is Parsing Tree?**(MAY 2012) (Ref.Qn.No.10, Pg.no.3)**
5. Define Three-Address Code.**(NOV 2012) (Ref.Qn.No.41, Pg.no.9)**
6. Differentiate phase and pass.**(NOV 2012) (Ref.Qn.No.32, Pg.no.7)**
7. Derive the first and follow for the follow for the following grammar.

S0|1|AS0|BS0 Aɛ Bɛ **(MAY 2013) (Ref.Qn.No.58, Pg.no.13)**

1. State the function of an intermediate code generator.**(MAY 2013) (Ref.Qn.No.33, Pg.no.7)**
2. Briefly describe the LL (k) items.**(NOV 2013) (Ref.Qn.No.19, Pg.no.5)**
3. What are the different forms of Intermediate representations?**(NOV 2013) (Ref.Qn.No.35, Pg.no.8)**

**11 MARKS**

**NOV 2011(REGULAR)**

**1.** Explain the LR parsing algorithm in detail. **(Ref.Qn.No.8, Pg.no.32)**

**(OR)**

1. a) How back patching can be used to generate code for Boolean expressions? (6)

**(Ref.Qn.No.21, Pg.no.72**

* 1. How the code is generated for procedure calls? (5) **(Ref.Qn.No.19, Pg.no.70)**

**MAY 2012(ARREAR)**

1. a) Write an algorithm for constructing LR parser table. **(Ref.Qn.No.8, Pg.no.32)**
   1. Write the quadruple representation for the assignment statement a: =-b\*(c+d) **(Ref.Qn.No.14, Pg.no.52)**

**(OR)**

**2.** Discuss the Role of the parser. **(Ref.Qn.No.1, Pg.no.14)**

**NOV 2012(REGULAR)**

1. a) Write an algorithm for constructing LR parser table. **(Ref.Qn.No.8, Pg.no.32)**
   1. Consider the following grammar to construct the LR parsing table **(Ref.Qn.No.9, Pg.no.35)**

EE+T | T TT\*F | F F (E) | id

P a g e | **76** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**(OR)**

**2.** List out and discuss the different type of intermediate code. **(Ref.Qn.No.10, Pg.no.43)**

**MAY 2013(ARREAR)**

**1.** Give the following CFG grammar G=({S,A,B},S,(a,b,x),P) with P:SA

Sxb

AaAb

AB Bx

For this grammar answer the following questions:

Compete the set of LR (1) items for this grammar. Augment the grammar with the default initial production S’S$ as the production (0) and Construct the corresponding LR parsing table.

**(OR)**

**2.** State and write the semantic rules for Boolean expressions. **(Ref.Qn.No.17, Pg.no.61)**

**NOV 2013 (REGULAR)**

1. (a) Write the steps in writing a grammar for a programming language. (5) **(Ref.Qn.No.3, Pg.no.19)**
   1. Briefly write on Parsing techniques. Explain with illustration the designing of a Predictive Parser. (6)

**(Ref.Qn.No.4, Pg.no.23)**

**(OR)**

**2.** Describe in detail the declarations in a procedure and the methods to keep track of scope information.

**(Ref.Qn.No.15, Pg.no.53)**

P a g e | **77** LANGUAGE TRANSLATORS DEPARTMENT OF CSE