TITLE: Int code generation for sample language using LEX and YACC.

THEORY:

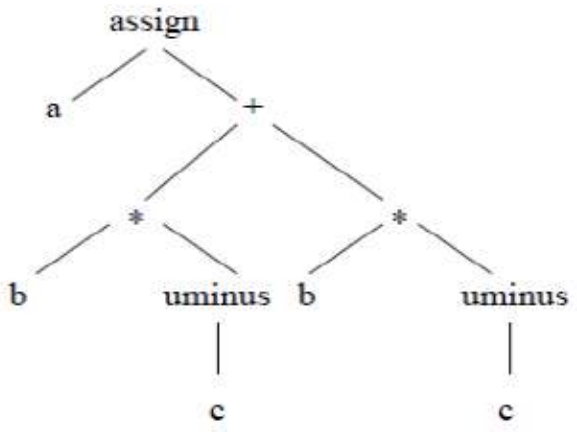
In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. The front end translates a source program into an intermediate representation from which the back end generates target code. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: m x n compilers can be built by writing just m front ends and n back ends.

Intermediate Languages: Three ways of intermediate representation:

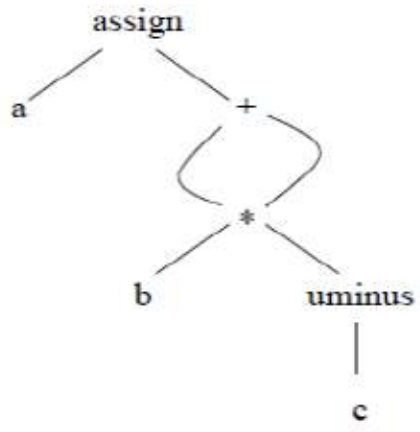
- Syntax tree
- Postfix notation
- Three address code

1. Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub expressions are identified. A syntax tree and DAG for the assignment statement a: = b * - c + b * - c are as follows:



(a) Syntax tree



(b) Dag

2. 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 given above is a b c uminus * b c uminus * + assign.

3. Three-Address Code:

Three-address code is a sequence of statements of the general form x: = y op z

Where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data.

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. The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Implementation of Three-Address Statements:

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

A. Quadruples:

A quadruple is a record structure with four fields, which are, op, arg1, arg2 and result. The op field contains an internal code for the operator. The 3-address statement x = y op z is represented by placing y in arg1, z in arg2 and x in result. The contents of fields arg1, arg2 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.

Fig a shows quadruples for the assignment a: b * c + b * c

B. 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. If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.

The fields arg1 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. Fig b) shows the triples for the assignment statement a: = b * c + b * c.

C. Indirect triples:

Indirect triple representation is the listing pointers to triples rather-than listing the triples themselves. Let us use an array statement to list pointers to triples in the desired order. Fig c) shows the indirect triple representation

Quadruples

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Indirect Triples

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

ALGORITHM:

1) Start
2) Input file name and valid sample C file
1) 3)Open and Read file
3) Match the tokens
4) If token is variable/identifier, make entry in symbol table with corresponding datatype and default value
5) Match against Grammar and generate appropriate error message
6) Map the Grammar to appropriate Quadruple representation
7) Repeat step 4) 5) 6) 7) till sample program ends
8) Display symbol table and Quadruple representation
9) Stop

/**********OUTPUT**********

$ yacc -d parser.y

$ lex lexicalanalyzer.l

$ gcc y.tab.c lex.yy.c -ll -ly

$ ./a.out input.c

The Program is Syntactically Correct!!!

The Symbol Table

Name	Type
a	int
b	int
x	int
c	int
d	int
e	int

The INTERMEDIATE CODE Is:

The Quadruple Table

Operator	Operand1	Operand2	Result
0	>	a	b	t0
1	==	t0	FALSE	5
2	+	a	b	t1
3	=	t1		x
4	GOTO			7
5	-	a	b	t2
6	=	t2		x

**********/