COMPILER DESIGN

UNIT 1: Introduction to Compiler

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.

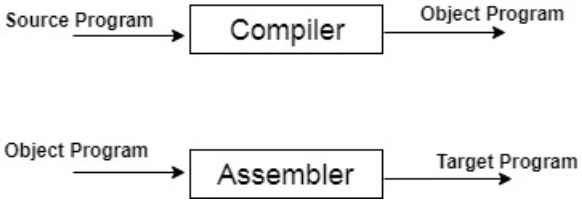


Fig: Execution process of source program in Compiler

Compiler Phases

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:

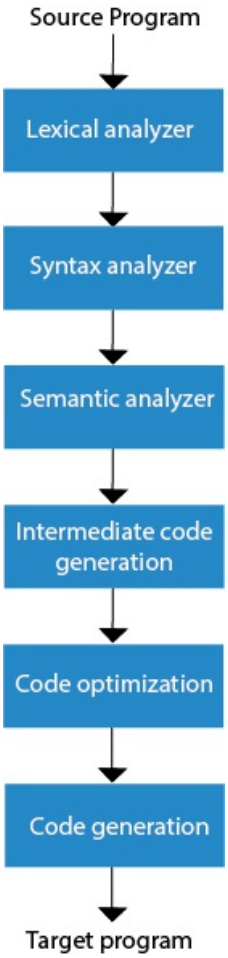


Fig: phases of com piler

Lexical Analysis:

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

Syntax Analysis

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

Semantic Analysis

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

Intermediate Code Generation

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

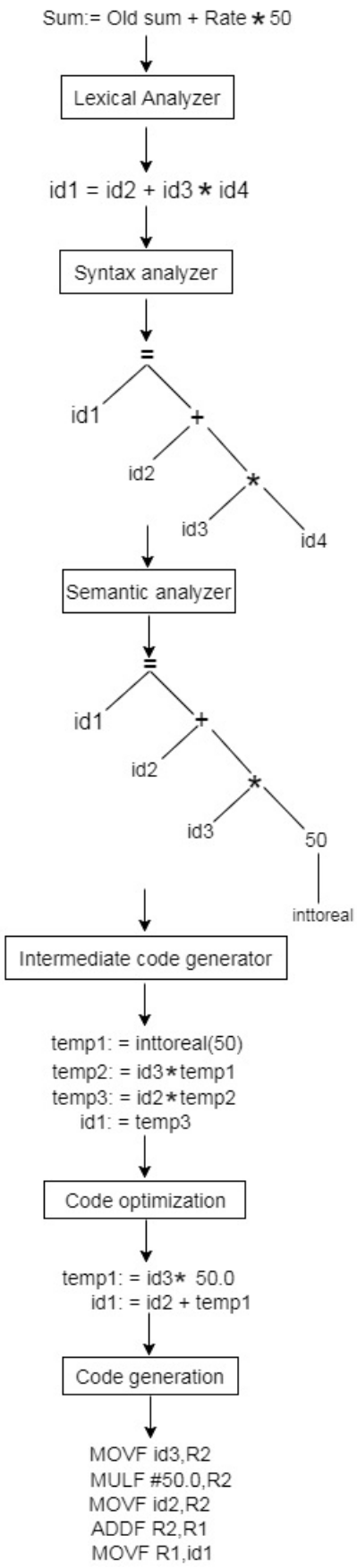
Code Optimization

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

Code Generation

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

Example:



Compiler Passes

Pass is a complete traversal of the source program. Compiler has two passes to traverse the source program.

Multi-pass Compiler

- Multi pass compiler is used to process the source code of a program several times.
- In the first pass, compiler can read the source program, scan it, extract the tokens and store the result in an output file.
- In the second pass, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.
- In the third pass, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax.
- This pass is going on, until the target output is produced.

One-pass Compiler

- One-pass compiler is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
- In the one pass compiler, when the line source is processed, it is scanned and the token is extracted.
- Then the syntax of each line is analyzed and the tree structure is build. After the semantic part, the code is generated.
- The same process is repeated for each line of code until the entire program is compiled.

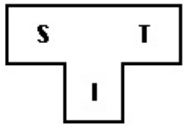
Bootstrapping

- Bootstrapping is widely used in the compilation development.
- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.
- Bootstrap compiler is used to compile the compiler and then you can use this compiled compiler to compile everything else as well as future versions of itself.

A compiler can be characterized by three languages:

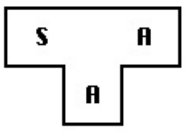
1. Source Language
2. Target Language
3. Implementation Language

The T- diagram shows a compiler ${}^{S}C_{I}{}^{T}$ for Source S, Target T, implemented in I.

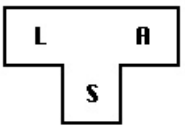


Follow some steps to produce a new language **L** for machine **A**:

1. Create a compiler ${}^{S}C_{A}{}^{A}$ for subset, S of the desired language, L using language "A" and that compiler runs on machine A.

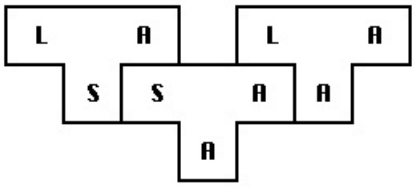


2. Create a compiler ${}^{L}C_{S}{}^{A}$ for language L written in a subset of L.



3. Compile ${}^{L}C_{S}{}^{A}$ using the compiler ${}^{S}C_{A}{}^{A}$ to obtain ${}^{L}C_{A}{}^{A}$. ${}^{L}C_{A}{}^{A}$ is a compiler for language L, which runs on machine A and produces code for machine A.

$$ {}^{L}C_{S}^{A} \rightarrow {}^{S}C_{A}^{A} \rightarrow {}^{L}C_{A}^{A} $$



The process described by the T-diagrams is called bootstrapping.

Finite state machine

- Finite state machine is used to recognize patterns.
- Finite automata machine takes the string of symbol as input and changes its state accordingly. In the input, when a desired symbol is found then the transition occurs.
- While transition, the automata can either move to the next state or stay in the same state.
- FA has two states: accept state or reject state. When the input string is successfully processed and the automata reached its final state then it will accept.

A finite automata consists of following:

Q: finite set of states
∑: finite set of input symbol
q0: initial state
F: final state

δ: Transition function

Transition function can be define as

1. $\delta: Q \times \sum \rightarrow Q$

FA is characterized into two ways:

1. DFA (finite automata)
2. NDFA (non deterministic finite automata)

DFA

DFA stands for Deterministic Finite Automata. Deterministic refers to the uniqueness of the computation. In DFA, the input character goes to one state only. DFA doesn't accept the null move that means the DFA cannot change state without any input character.

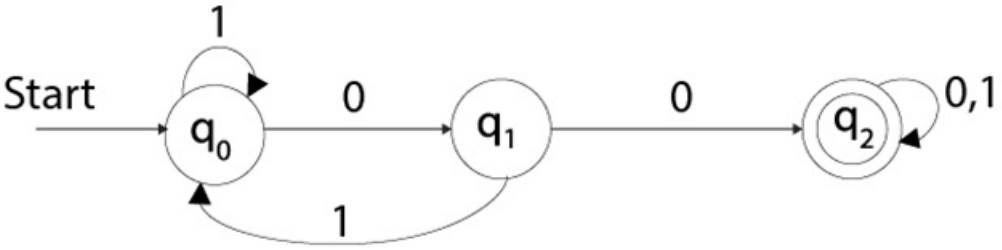
DFA has five tuples $\{Q, \sum, q0, F, \delta\}$

Q: set of all states
$\sum$: finite set of input symbol where $\delta: Q \times \sum \rightarrow Q$
q0: initial state
F: final state
δ: Transition function

Example

See an example of deterministic finite automata:

1. Q = {q0, q1, q2}
2. $\sum$ = {0, 1}
3. q0 = {q0}
4. F = {q3}



NDFA

NDFA refer to the Non Deterministic Finite Automata. It is used to transit the any number of states for a particular input. NDFA accepts the NULL move that means it can change state without reading the symbols.

NDFA also has five states same as DFA. But NDFA has different transition function.

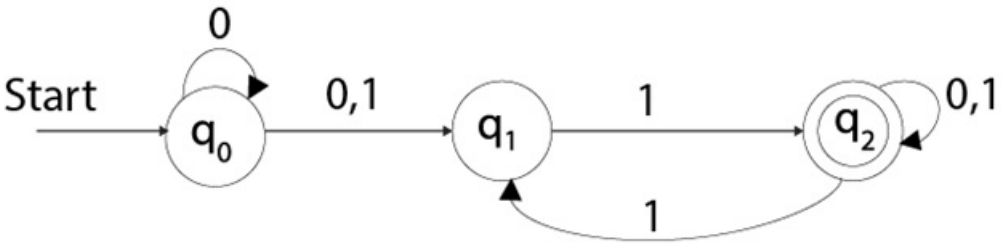
Transition function of NDFA can be defined as:

$\delta: Q \times \sum \rightarrow 2^Q$

Example

See an example of non deterministic finite automata:

1. Q = {q0, q1, q2}
2. $\sum$ = {0, 1}
3. q0 = {q0}
4. F = {q3}



Regular expression

- Regular expression is a sequence of pattern that defines a string. It is used to denote regular languages.
- It is also used to match character combinations in strings. String searching algorithm used this pattern to find the operations on string.

- In regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx,.....}
- In regular expression, x+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx,.....}

Operations on Regular Language

The various operations on regular language are:

Union: If L and M are two regular languages then their union L U M is also a union.

1. L U M = {s | s is in L or s is in M}

Intersection: If L and M are two regular languages then their intersection is also an intersection.

1. L ∩ M = {st | s is in L and t is in M}

Kleene closure: If L is a regular language then its kleene closure L1* will also be a regular language.

1. L* = Zero or more occurrence of language L.

Example

Write the regular expression for the language:

$L = \{ab^n\ w{:}n \geq 3, w \in (a,b)^+\}$

Solution:

The string of language L starts with "a" followed by atleast three b's. Itcontains atleast one "a" or one "b" that is string are like abbba, abbbbbba, abbbbbbbb, abbbb.....a

So regular expression is:

$r = ab^3b^* (a+b)^+$

Here + is a positive closure i.e. $(a+b)^+ = (a+b)^* - \in$

Optimization of DFA

To optimize the DFA you have to follow the various steps. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states and T2 contains non-final states.

Step 4: Find the similar rows from T1 such that:

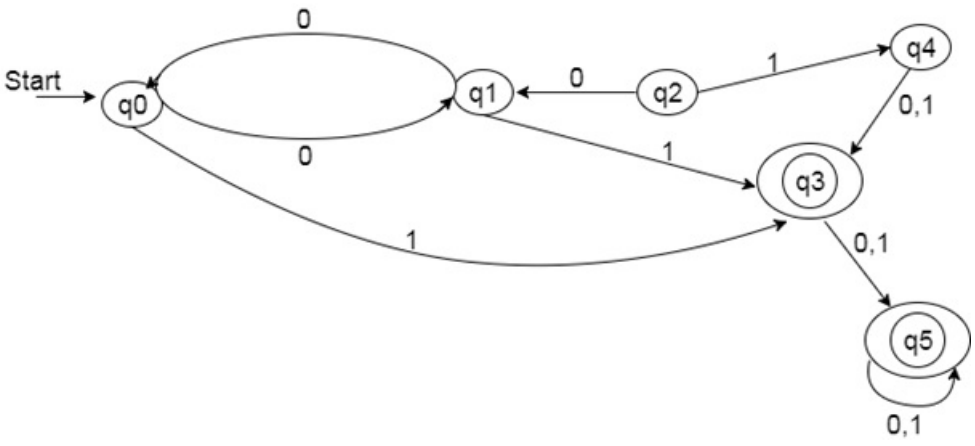
That means, find the two states which have same value of a and b and remove one of them.

Step 5: Repeat step 3 until there is no similar rows are available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example



Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for rest of the states.

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q5	q5
*q5	q5	q5

Step 3:

Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final sates:

State	0	1
q0	q1	q3
q1	q0	q3

2. Other set contains those rows, which starts from final states.

State	0	1
q3	q5	q5
q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
*q3	q3	q3

Now it is the transition table of minimized DFA.

Transition diagram of minimized DFA:

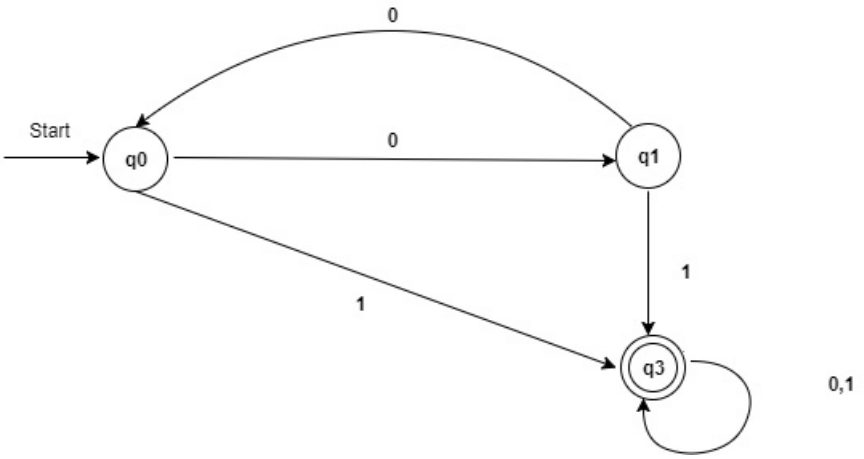


Fig: Minimized DFA

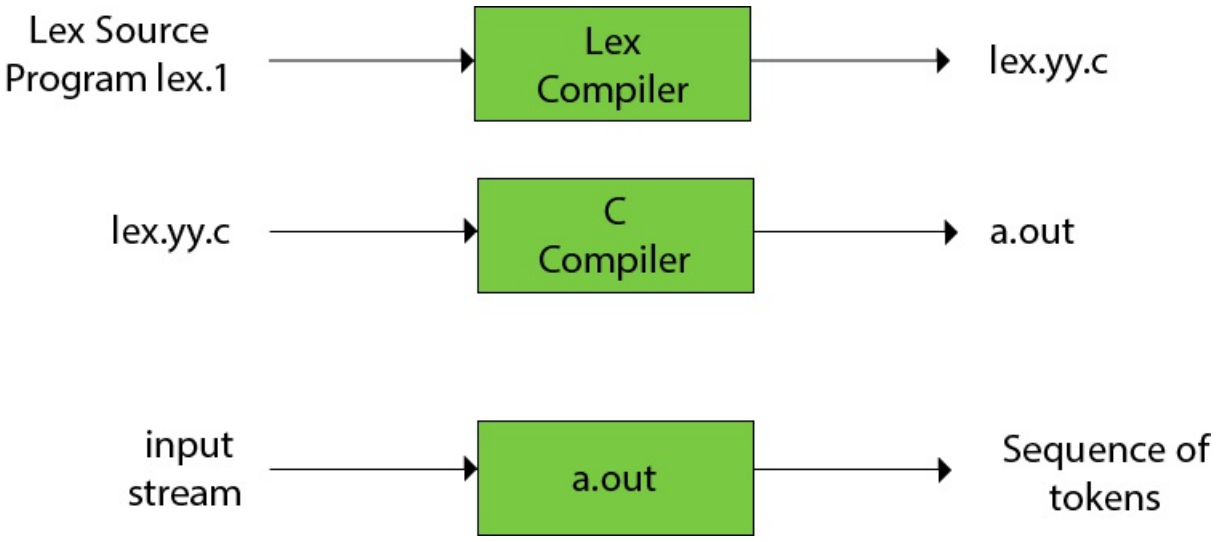
LEX

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.

- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

Definitions include declarations of constant, variable and regular definitions.

Rules define the statement of form p1 {action1} p2 {action2}....pn {action}.

Where **pi** describes the regular expression and **action1** describes the actions what action the lexical analyzer should take when pattern pi matches a lexeme.

User subroutines are auxiliary procedures needed by the actions. The subroutine can be loaded with the lexical analyzer and compiled separately.

Formal grammar

- Formal grammar is a set of rules. It is used to identify correct or incorrect strings of tokens in a language. The formal grammar is represented as G.
- Formal grammar is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- Formal grammar is used mostly in the syntactic analysis phase (parsing) particularly during the compilation.

Formal grammar G is written as follows:

1. G = <V, N, P, S>

Where:

N describes a finite set of non-terminal symbols.
V describes a finite set of terminal symbols.
P describes a set of production rules
S is the start symbol.

Example:

1. L = {a, b}, N = {S, R, B}

Production rules:

1. S = bR
2. R = aR
3. R = aB
4. B = b

Through this production we can produce some strings like: bab, baab, baaab etc.

This production describes the string of shape ba^nab.

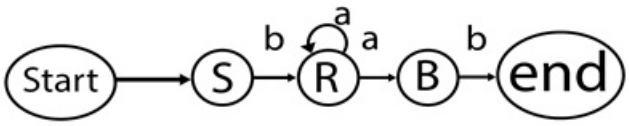


Fig: Formal grammar

BNF Notation

BNF stands for **Backus-Naur Form**. It is used to write a formal representation of a context-free grammar. It is also used to describe the syntax of a programming language.

BNF notation is basically just a variant of a context-free grammar.

In BNF, productions have the form:

Where leftside $\in (V_n \cup V_t)+$ and definition $\in (V_n \cup V_t)*$. In BNF, the leftside contains one non-terminal.

We can define the several productions with the same leftside. All the productions are separated by a vertical bar symbol "|".

There is the production for any grammar as follows:

In BNF, we can represent above grammar as follows:

YACC

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.

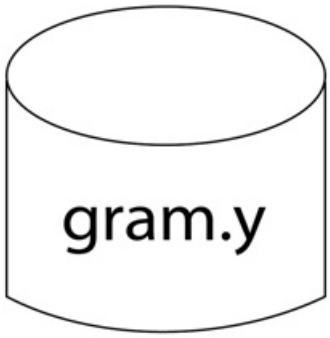
These are some points about YACC:

Input: A CFG- file.y

Output: A parser y.tab.c (yacc)

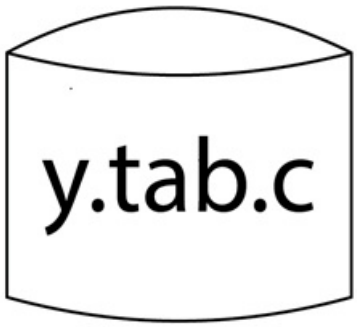
- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.

The basic operational sequence is as follows:



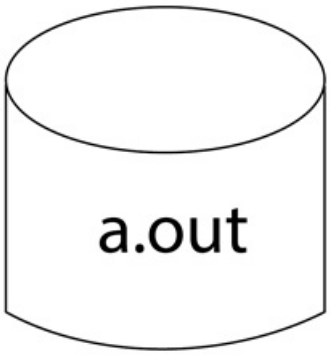
This file contains the desired grammar in YACC format.

It shows the YACC program.



It is the c source program created by YACC.

C Compiler



Executable file that will parse grammar given in gram.Y

Context free grammar

Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.

Context free grammar G can be defined by four tuples as:

Where,

G describes the grammar

T describes a finite set of terminal symbols.

V describes a finite set of non-terminal symbols

P describes a set of production rules

S is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Example:

L= $\{wcw^R \mid w \in (a, b)^*\}$

Production rules:

Now check that abbcbba string can be derived from the given CFG.

By applying the production S → aSa, S → bSb recursively and finally applying the production S → c, we get the string abbcbba.

Capabilities of CFG

There are the various capabilities of CFG:

- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficientparser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

Derivation

Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be replaced with production rule.

Left-most Derivation

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Example:

Production rules:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

The left-most derivation is:

1. S = S + S
2. S = S - S + S
3. S = a - S + S
4. S = a - b + S
5. S = a - b + c

Right-most Derivation

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

Example:

1. S = S + S
2. S = S - S
3. S = a | b |c

Input:

a - b + c

The right-most derivation is:

1. S = S - S
2. S = S - S + S
3. S = S - S + c
4. S = S - b + c
5. S = a - b + c

Parse tree

- Parse tree is the graphical representation of symbol. The symbol can be terminal or non-terminal.
- In parsing, the string is derived using the start symbol. The root of the parse tree is that start symbol.
- It is the graphical representation of symbol that can be terminals or non-terminals.
- Parse tree follows the precedence of operators. The deepest sub-tree traversed first. So, the operator in the parent node has less precedence over the operator in the sub-tree.

The parse tree follows these points:

- All leaf nodes have to be terminals.
- All interior nodes have to be non-terminals.
- In-order traversal gives original input string.

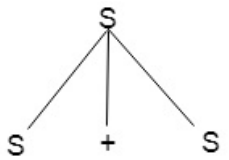
Example:

Production rules:

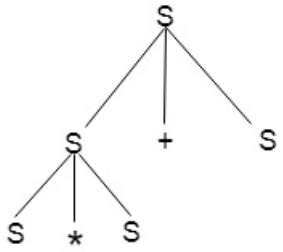
Input:

a * *b* + *c*

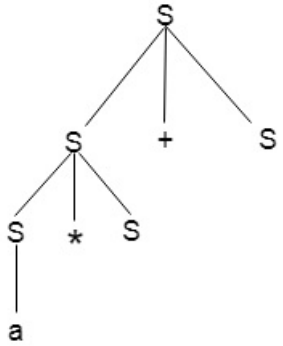
Step 1:



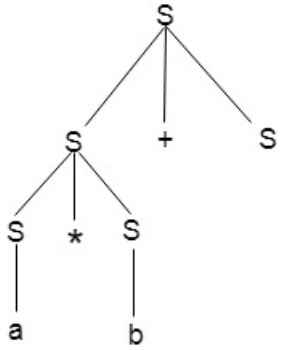
Step 2:



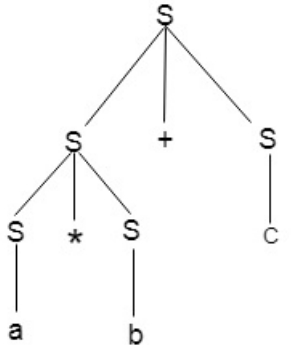
Step 3:



Step 4:



Step 5:



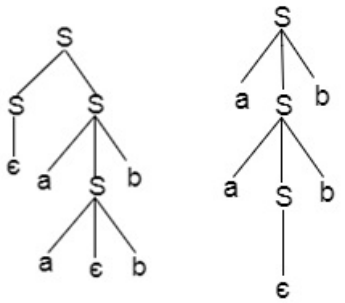
Ambiguity

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivative or more than one parse tree for the given input string. If the grammar is not ambiguous then it is called unambiguous.

Example:

1. S = aSb | SS
2. S = ∈

For the string aabb, the above grammar generates two parse trees:



If the grammar has ambiguity then it is not good for a compiler construction. No method can automatically detect and remove the ambiguity but you can remove ambiguity by re-writing the whole grammar without ambiguity.

Array references in arithmetic expressions

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive location. Array can be one dimensional or two dimensional.

For one dimensional array:

1. A: array[low..high] of the ith elements is at:
2. base + (i-low)*width → i*width + (base - low*width)

Multi-dimensional arrays:

Row major or column major forms

- Row major: a[1,1], a[1,2], a[1,3], a[2,1], a[2,2], a[2,3]
- Column major: a[1,1], a[2,1], a[1, 2], a[2, 2],a[1, 3],a[2,3]
- In raw major form, the address of a[i1, i2] is
- Base+((i1-low1)*(high2-low2+1)+i2-low2)*width

Translation scheme for array elements

Limit(array, j) returns nj=highj-lowj+1

place: the temporaryor variables

offset: offset from the base, null if not an array reference

The production:

1. S → L := E
2. E → E+E
3. E → (E)
4. E → L
5. L → Elist]
6. L → id
7. Elist → Elist, E
8. Elist → id[E

A suitable transition scheme for array elements would be:

Production Rule	**Semantic Action**

S → L := E	{if L.offset = null then emit(L.place ':=' E.place)
	else EMIT (L.place'['L.offset ']' ':=' E.place);
	}

E → E+E	{E.place := newtemp; EMIT (E.place ':=' E1.place '+' E2.place); }

E → (E)	{E.place := E1.place;}

E → L	{if L.offset = null then E.place = L.place
	else {E.place = newtemp;
	EMIT (E.place ':=' L.place '[' L.offset ']');
	}
	}

L → Elist]	{L.place = newtemp; L.offset = newtemp; EMIT (L.place ':=' c(Elist.array)); EMIT (L.offset ':=' Elist.place '*' width(Elist.array); }
L → id	{L.place = lookup(id.name); L.offset = null; }

Elist → Elist, E	{t := newtemp;
	m := Elist1.ndim + 1;
	EMIT (t ':=' Elist1.place '*' limit(Elist1.array, m));
	EMIT (t, ':=' t '+' E.place);
	Elist.array = Elist1.array;
	Elist.place := t;
	Elist.ndim := m;
	}

Elist → id[E	{Elist.array := lookup(id.name); Elist.place := E.place Elist.ndim := 1; }

Where:

ndim denotes the number of dimensions.

limit(array, i) function returns the upper limit along with the dimension of array

width(array) returns the number of byte for one element of array.

Statements that alter the flow of control

The goto statement alters the flow of control. If we implement goto statements then we need to define a LABEL for a statement. A production can be added for this purpose:

1. S → LABEL : S
2. LABEL → id

In this production system, semantic action is attached to record the LABEL and its value in the symbol table.

Following grammar used to incorporate structure flow-of-control constructs:

1. S → **if** E then S
2. S → **if** E then S **else** S
3. S → **while** E **do** S
4. S → begin L end
5. S→ A
6. L→ L ; S
7. L → S

Here, S is a statement, L is a statement-list, A is an assignment statement and E is a Boolean-valued expression.

PauseNext

Mute

Current Time 0:00

/

Duration 0:00

Loaded: 0%

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

Translation scheme for statement that alters flow of control

- We introduce the marker non-terminal M as in case of grammar for Boolean expression.
- This M is put before statement in both if then else. In case of while-do, we need to put M before E as we need to come back to it after executing S.
- In case of if-then-else, if we evaluate E to be true, first S will be executed.
- After this we should ensure that instead of second S, the code after the if-then else will be executed. Then we place another non-terminal marker N after first S.

The grammar is as follows:

1. S → **if** E then M S
2. S → **if** E then M S **else** M S
3. S → **while** M E **do** M S
4. S → begin L end
5. S → A
6. L→ L ; M S
7. L → S
8. M → ∈
9. N → ∈

The translation scheme for this grammar is as follows:

Production rule	**Semantic actions**
S → if E then M S1	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
S → if E then M1 S1 else M2 S2	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
S → while M1 E do M2 S1	BACKPATCH (S1,NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
S → begin L end	S.NEXT = L.NEXT
S → A	S.NEXT = MAKELIST ()
L → L ; M S	BACKPATHCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
L → S	L.NEXT = S.NEXT
M → ∈	M.QUAD = NEXTQUAD
N→ ∈	N.NEXT = MAKELIST (NEXTQUAD) GEN (goto_)

Triples

The triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.

In triples, the results of respective sub-expressions are denoted by the position of expression. Triple is equivalent to DAG while representing expressions.

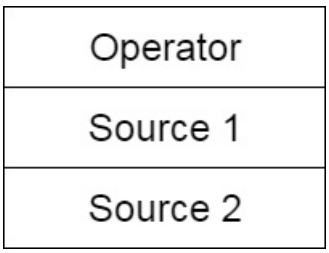


Fig: Triples field

Example:

1. a := -b * c + d

Three address code is as follows:

$t_1 := -b$ $t_2 := c + d$M $t_3 := t_1 * t_2$ $a := t_3$

These statements are represented by triples as follows:

Operator Source 1 Source 2

(0) uminus	b	-
(1) +	c	d
(2) *	(0)	(1)
(3) :=	(2)	-

Three address code

- Three-address code is an intermediate code. It is used by the optimizing compilers.
- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Example

GivenExpression:

1. a := (-c * b) + (-c * d)

Three-address code is as follows:

$t_1 := -c$

$t_2 := b * t_1$

$t_3 := -c$

$t_4 := d * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

t is used as registers in the target program.

The three address code can be represented in two forms: **quadruples** and **triples**.

Boolean expressions

Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

Consider the grammar

1. E → E OR E
2. E → E AND E
3. E → NOT E
4. E → (E)
5. E → id relop id
6. E → TRUE
7. E → FALSE

The relop is denoted by <, >, <, >.

The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Backward Skip 10sPlay VideoForward Skip 10s

Production rule	**Semantic actions**
E → E1 OR E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'OR' E2.place) }
E → E1 + E2	{E.place = newtemp(); Emit (E.place ':=' E1.place 'AND' E2.place) }
E → NOT E1	{E.place = newtemp(); Emit (E.place ':=' 'NOT' E1.place) }
E → (E1)	{E.place = E1.place}
E → id relop id2	{E.place = newtemp(); Emit ('if' id1.place relop.op id2.place 'goto' nextstar + 3); EMIT (E.place ':=' '0')

```
                    EMIT ('goto' nextstat + 2)
                    EMIT (E.place ':=' '1')
                    }
                    {E.place := newtemp();
E → TRUE            Emit (E.place ':=' '1')
                    }
                    {E.place := newtemp();
E → FALSE           Emit (E.place ':=' '0')
                    }
```

The EMIT function is used to generate the three address code and the newtemp( ) function is used to generate the temporary variables.

The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Here is the example which generates the three address code using the above translation scheme:

```
1.  p>q AND r<s OR u>r
2.      100: if p>q goto 103
3.      101: t1:=0
4.      102: goto 104
5.      103: t1:=1
6.      104: if r>s goto 107
7.      105: t2:=0
8.      106: goto 108
9.      107: t2:=1
10.     108: if u>v goto 111
11.     109: t3:=0
12.     110: goto 112
13.     111: t3:= 1
14.     112: t4:= t1 AND t2
15.     113: t5:= t4 OR t3
```

Case Statements

Switch and case statement is available in a variety of languages. The syntax of case statement is as follows:

```
1.    switch E
2.            begin
3.                    case V1: S1
4.                    case V2: S2
5.                    .
6.                    .
7.                    .
8.  case Vn-1: Sn-1
9.  default: Sn
10.           end
```

The translation scheme for this shown below:

Code to evaluate E into T

```
1.  goto TEST
2.            L1:     code for S1
3.                    goto NEXT
4.            L2:     code for S2
5.                    goto NEXT
6.                    .
7.                    .
8.                    .
9.            Ln-1:   code for Sn-1
10.                   goto NEXT
11.           Ln:     code for Sn
12. goto NEXT
13.           TEST:    if T = V1 goto L1
14.                    if T = V2goto L2
15.                    .
16.                    .
17.                    .
18.                    if T = Vn-1 goto Ln-1
19.                    goto
20. NEXT:
```

- When switch keyword is seen then a new temporary T and two new labels test and next are generated.
- When the case keyword occurs then for each case keyword, a new label Li is created and entered into the symbol table. The value of Vi of each case constant and a pointer to this symbol-table entry are placed on a stack.