

Describe input buffering scheme in lexical analyzer

An input buffering scheme in a lexical analyzer is a technique used to improve the efficiency of reading input from the source program. In this scheme, the input is read in blocks or batches instead of individual characters. The size of the block is typically determined by the number of characters that can be read efficiently as a group. When a block of input is read, it is stored in a buffer or array. The lexical analyzer then processes the input in the buffer, one character at a time. As each character is processed, it is removed from the buffer until the entire buffer has been processed. At that point, the next block of input is read and stored in the buffer, and the process is repeated. The advantage of this scheme is that it reduces the overhead associated with reading individual characters from the source program. By reading input in larger blocks, the lexical analyzer can process more characters at once, which can be faster and more efficient. Additionally, the input buffering scheme can reduce the number of system calls required to read input from the source program. This can be beneficial in systems where system calls are expensive or time-consuming. Overall, the input buffering scheme is a useful technique for improving the performance of lexical analyzers and other programs that read input from external sources.

To develop a lexical analyzer to identify identifiers, constants, comments, operators etc using C program

ALGORITHM:

- Step1: Start the program.
- Step2: Declare all the variables and file pointers.
- Step3: Display the input program.
- Step4: Separate the keyword in the program and display it.
- Step5: Display the header files of the input program
- Step6: Separate the operators of the input program and display it.
- Step7: Print the punctuation marks.
- Step8: Print the constant that are present in input program.
- Step9: Print the identifiers of the input program

Compiler writing tools

1. Parser Generators: these produce syntax analysis, normally from input that is based on a context free grammar.
Scanner Generators: these automatically generate lexical analyzer.
Syntax-Directed Translation Engines: In this tool, the parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree
Automatic Code Generator: These generators take an intermediate code as input and convert each rule of intermediate language into equivalent machine language. The template matching technique is used.
The intermediate code statements are replaced templates that represent the corresponding sequence of machine instructions.
5) Data-Flow Analysis Engines: Data-flow analysis engines that facilitate the path of information about how values are transmitted one part of a program to leach other part. Data-flow analysis is a key part of code optimization.
Compiler-construction toolkits : Compiler construction toolkits that provide an integrated set of routines for constructing various phases of a Compiler.

Bootstrapping

The process by which a simple language is used to translate a more complicated program, which in turn may handle an even more complicated program and so on, is known as bootstrapping. In other words, one wants to write a compiler for a language A, targeting language B (the machine language) and written in language B. The most obvious approach is to write the compiler in language B. But if B is machine language, it is a horrible job to write any non-trivial compiler in this language. Instead, it is customary to use a process called "bootstrapping", referring to the seemingly impossible task of pulling oneself up by the bootstraps. Bootstrapping is an important concept in building new compilers. A compiler is characterized by three languages:..

Its source language,
Its object language and
The language in which it is written.

Advantages

It is a non-trivial test of the language being compiled, and as such is a form of dogfooding.
Compiler developers and bug reporting part of the community only need to know the language being compiled.
Compiler development can be done in the higher level language being compiled. Improvements to the compiler's back-end improve not only general purpose programs but also the compiler itself. It is a comprehensive consistency check as it should be able to reproduce its own object code.

PARSING CONFLICTS IN SLR PARSING TABLE:

2 TYPES:

- 1.A shift -reduce conflict occurs in a state that requests both a shift action and a reduce action.
- 2.A reduce -reduce conflict occurs in a state that request 2 or more different reduce actions.

Operator-Precedence Parsing Algorithm

Algorithm: Operator-precedence parsing algorithm

Input: The precedence relations from some operator precedence grammar and an input string of terminals from that grammar
Output: Strictly speaking, there is no output. We could construct a skeletal parse tree as we parse, with one non-terminal labeling all interior nodes and the use of single productions not shown. Alternatively, the sequence shift-reduce steps could be considered the output.

Method: Let the input string be $a_1a_2...a_n$. Initially, The stack contains \$.

MOD 1

PHASES OF COMPILER:

-Lexical Analysis

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

- <token-name, attribute-value>

-Syntax Analysis

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

-Semantic Analysis

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

-Intermediate Code Generation

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

-Code Optimization

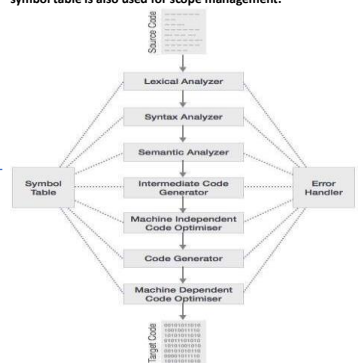
The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

-Code Generation

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

-Symbol Table

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



Token	Lexeme	Pattern
Const	Const	Const
If	If	If
Relation	<, <=, =, >, >=, >	< or <= or = or > or >= or >
Id	Pl, count, n, l	Letter followed by letters and digits.
Number	3.14159, 0, 6.02e23	Any numeric constant
Literal	"Darshan Institute"	Any character between "and" "except"

TOKENS:

A token is a fundamental unit of syntax. In programming, a token is a sequence of characters that represent a specific type of element or operator. programming languages like C language keywords (int, char, float, const, goto, continue, etc.) identifiers (user-defined names), operators (+, -, *, /, %, etc.), delimiters/punctuators like comma (,), semicolon(;), braces ({}, etc.), strings can be considered as tokens. This phase recognizes three types of tokens: Terminal Symbols (TRM)- Keywords and Operators, Literals (LIT), and Identifiers (IDN)

Patterns :

Pattern describes a rule that must be matched by sequence of character lexemes to form a token. It can be defined by regular expressions or grammar rules.

Lexeme :

Lexeme is a sequence of character that matches the pattern for a token i.e instance of a token .

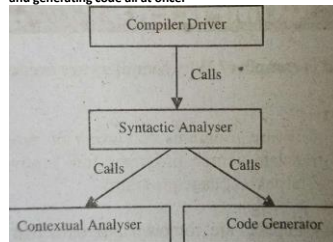
MOD 3:

MOD

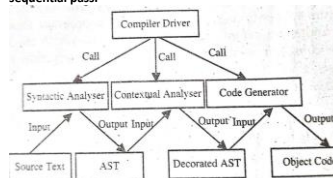
TYPES OF COMPILER

-Single pass compiler:

It makes a single pass over the source text, parsing, analysing and generating code all at once.



Multi pass Compiler: A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases. A multipass compiler makes the source code go through parsing, analyzing, generating, etc. multiple times while generating intermediate code after each stage. It converts the program into one or more intermediate representations in steps between source code and machine code. It reprocesses the entire compilation unit in each sequential pass.



SYMBOL TABLE:

A Symbol Table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The symbol table is accessed by most phases of a compiler, beginning with the lexical analysis to optimization. The symbol table carries the collected information about each named object in the program to other phases of the compiler. As soon as a named token is found, depending on the token type, a call to the symbol table routines must be made. The symbol table- will contain the following types of information for the input strings in a source program:

Use of Symbol Table by Analysis and Synthesis Phases

Symbol table information is used by the analysis and synthesis phases.

- To verify that used identifiers have been defined (declared),
- To verify that expressions and assignments are semantically correct - type checking.

Regular Expressions

A regular expression is a description of a set of strings. We define regular expression as a means of representing certain subsets of strings over the alphabet and prove that regular sets are precisely those accepted by finite automata or transition diagrams. Regular expressions are useful for representing certain sets of strings in an algebraic fashion. Actually these describe the language accepted by finite state automata. Compiler uses this program of lexical analyzer in the process of compilation. The task of lexical analyzer is to scan the input program and separate out the 'tokens'.

Lex :

Lex, tool is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The action is a piece of code, which is to be executed whenever a token specified by the corresponding regular expression is recognized. LEX helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Recursive + Descent Parsing

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right it uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as predictive parsing.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature. The main limitation of recursive descent parsing (and all top-down parsing algorithms in general) is that they only work on grammars with certain properties. For example, if a grammar contains any left recursion, recursive descent parsing does not work.

Left Recursive Grammar "r":

A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS. A grammar containing a production having left recursion is called as Left Recursive Grammar.

A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

$A \rightarrow \alpha A \beta$

The above Grammar is left recursive because the left of production is occurring at a first position on the right side

MOD 2

Bottom-Up Evaluation of S-Attributed Definitions Synthesised attributes can be evaluated by a bottom-up parser as the input is being parsed. The parser can keep the values of the synthesised attributes associated with the grammar symbols on its stack. Whenever a reduction is made, the values of the new synthesised attributes are computed from the attributes appearing on the stack for the grammar symbols on the right side of the reducing production.

Synthesised Attributes on the Parser Stack

1) A translator for S-attributed definition is implemented using LR parser generator. A bottom up method is used to parse the input string. A parser stack is used to hold the values of synthesised attribute.

The stack is implemented as a pair of state and value. Each state entry is the pointer to the LR (1) parsing table. There is no need to store the grammar symbol implicitly in the parser stack at the state entry. But for ease of understanding we will refer the state by unique grammar symbol that is been placed in the parser stack. Hence parser stack can be denoted as stack[i]. And stack[i] is a combination of state[i] and value.

Bottom-Up Evaluation of Inherited Attributes Using a bottom-up translation scheme, one can implement any L-attributed definition based on LL (1) grammar. He/fehe can also implement some of L-attributed definitions based on LR (1) using bottom-up translations scheme:

The semantic actions are evaluated during the reductions. During the bottom-up evaluation of S-attributed definitions, one has a parallel stack to hold synthesised attributes.

One will convert grammar to an equivalent grammar to guarantee the following:

All embedding semantic actions in our translation scheme will be moved to the end of the production rules.

All inherited attributes will be copied into the synthesised attributes (may be new non-terminals).

Error strategies:

Even though developers design programming languages with error handling capability. But the errors are inevitable in the programs despite the programmer's efforts. Thus, compilers are designed to track down and locate the errors in the program. It is the parser that efficiently detects a syntactic error from the program. After detecting the error, the parser must correct or recover that error. With the first approach, the parser can quit after detecting the first error. Thereby it leaves an informative message describing the error location.

But if the errors keep on rising, the compiler must give up after exceeding a particular limit. As it is not useful to create an avalanche of errors.

Following are the different error recovery strategies: 1) Panic-Mode Recovery: Here, the parser discards the input symbol one at a time until it discovers a synchronizing token. Usually, the synchronizing tokens are the delimiters. Though this method is simple. Yet it skips a large amount of input without scanning it for additional errors.

2) Phrase-Level Recovery: In phrase-level recovery, the parser performs a local correction on the remaining input. Once the parser encounters an error it replaces the prefix of the remaining input with a string. And this is done in such a way that the parser should not stop parsing and must continue.

Well, the choice of replacement is left to the compiler designer. But it must be taken care that the replacement should not lead to an infinite loop. This method can correct almost any input string. Still, this method is not able to recover from an error that has occurred before the point of detection.

Error Production: There are some erroneous productions that commonly occurs. These error productions are augmented with the grammar for a language. This facilitates the parser to detect the anticipated error.

4. Global Correction: Here, the parser must make least changes, while correcting an invalid string to a valid string. These methods are costly to implement and is only in theories.

Grammar :

It is a finite set of formal rules for generating syntactically correct sentences or meaningful correct sentences.

Grammar is basically composed of two basic elements - Terminal Symbols -

Terminal symbols are those which are the components of the sentences generated using a grammar and are represented using small case letter like a, b, c etc.

Non-Terminal Symbols - Non-Terminal Symbols are those symbols which take part in the generation of the sentence but are not the component of the sentence. Non-Terminal Symbols are also called Auxiliary Symbols and Variables. These symbols are represented using a capital letter like A, B, C, etc.

A context free grammar (CFG) is a formal grammar which is used to generate all the possible patterns of strings in a given formal language. It is defined as four tuples $G = (V, T, P, S)$ G is a grammar, which consists of a set of production rules. It is used to generate the strings of a language. T is the final set of terminal symbols. It is denoted by lower case letters. V is the final set of non-terminal symbols. It is denoted by capital letters P is a set of production rules, which is used for replacing non-terminal symbols (on the left side of production) in a string with other terminals (on the right side of production).

S is the start symbol used to derive the string

Ambiguity/ Ambiguous Grammars

A grammar that produces more than one parse tree for same string is said to be ambiguous. In other words an ambiguous grammar is one that produces more than one left most or more than one right most derivation tree for same string. A grammar G is said to be ambiguous if there is some word in L(G) generated by more than one leftmost derivation or rightmost derivation. In other words a grammar G is said to be ambiguous if there is some word in L(G) has atleast two derivation trees.

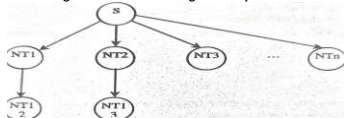
MOD2:

Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder. Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

The operations of construction of parse tree starting from the root node and proceeding toward leaves is called top down parsing, that is the top down parser attempts to derive a tree. Give an input string xy , successive applications of grammar to the grammar's distinguished symbol. If a is an input string then $S \rightarrow x \cdot y \rightarrow \dots \rightarrow a$

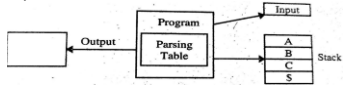
This is a top down loop approach it is easy and convenient to choose a non terminal from a sentential form and derive for left most of the sentential form. Hence Top Down parsing is called as LL (left to left parsing). The input string is scanned by the parser (left to right one symbol token at a time) and leftmost derivation generates the leaves of the parse tree in left to right order, which matches against order of scanning of the input.



Predictive Parsing

It is a top-down parsing method, which consists of a set of mutually procedures to process the input and handles a stack of activation records explicitly. Predictive parsing does not require backtracking in order to derive the input string. Predictive parsing is possible only for the class of LL(1) grammar (context-free grammar). Predictive parser is like a table representation of recursive descent parser. A predictive parser predicts the next construction in the input string by using one or more look-ahead token, i.e., these are table driven top down parsers used to eliminate backtracking.

Given a non-terminal the production should be made on the terminal that the non-terminal produces, along with the building a parse tree (which, indicates that right hand side choices to be made for production). Stacks are used in predictive parsing to avoid recursive procedures to store non-terminals in the present sequential form.



Method:

- 1) Arrange table M as two-dimensional array such that the terminals are on the columns and Non-Terminals are put in rows.
- 2) For each production of the form $A \rightarrow \alpha$, do steps 3 and 4.
- 3) For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].
- 4) If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and $\$$ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, \$].
- 5) Make each undefined entry of M as "ERROR".

Backtracking/ Backtracking Parsers

The process of repeated scans of the input string is called backtracking. A backtracking parser will go for different possibilities for a parse of the input. When the parser generates the parse tree and the leaves, it may repeatedly scan the input several times to obtain the leaves which are leftmost derivation. If a non-terminal A is to derive a next left most derivation then there may be multiple productions as,

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

Now this is a situation to decide the production says

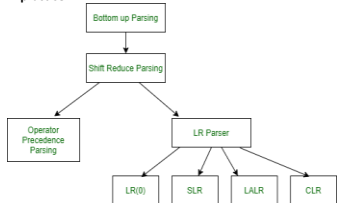
$$A \rightarrow \alpha_1 \mid \text{or } A \rightarrow \alpha_2 \dots \text{Like this and so on.}$$

Like this and so on.

Using the above productions and leftmost derivations the parser should finally lead to the derivation of a given string a , then it announces successful completion of parsing. Else the parser resets the pointer and tries for other productions and continues leftmost derivations until successful completion or failure comes. Hence a Top down parser may require to follow backtracking.

LR parser:

LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compiler and other associated tools. LR parser reads their input from left to right and produces a right-most derivation. It is called a Bottom-up parser because it attempts to reduce the top-level grammar productions by building up from the leaves. LR parsers are the most powerful parser of all deterministic parsers in practice.



Top-Down Parsing

For the construction of parsing tree, it follows top down approach.

The parse tree starts from root to leaves in a Preorder manner.

It top-down parsing non-terminal is expanded to derive the given input string.

It follows only the hierarchy of forward reference.

In top-down parsing, we have only the input and corresponding derive output.

Handle Pruning

Bottom-up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse. Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation. Right most derivation is achieved using handle pruning in reverse order. *

For example, consider the grammar:

$$E \rightarrow E + E \mid E * E \mid \text{id and input string id} + \text{id} * \text{id}$$

Shift-Reduce Parsing

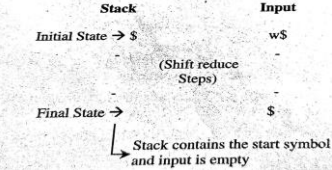
Shift reducing is a form of bottom up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. It is used to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

At each reduction step a string matching right side of a production is replaced by the symbol on left of that production.

Stack Implementation of Shift-Reduce Parsing

A convenient way to implement a shift reduce parser is to use a stack and an input buffer. We use $\$$ to mark the bottom of the stack and also the right end of the input. For example, for the following grammar...

$$E \rightarrow E + E \mid E * E \mid \text{id}$$



An **operator precedence grammar** is a kind of grammar for formal languages. Technically, an operator precedence grammar is a context-free grammar that has the property (among others) that no production has either an empty right-hand side or two adjacent nonterminals in its right-hand side. These properties allow precedence relations to be defined between the terminals of the grammar. A parser that exploits these relations is considerably simpler than more general-purpose parsers such as LALR parsers. Operator-precedence parsers can be constructed for a large class of context-free grammars. It is an operator grammar for which the precedence relations constructed using method 2 are disjoint. That means for any pair of terminals a and b , at any time, only one of the relations $a < b$, $a \text{ } \S \text{ } \text{overset{<}}{\text{>}} \text{ } b$, or $a \text{ } \S \text{ } \text{overset{<}}{\text{>}} \text{ } b$ is true. Consider the operator grammar: $E \rightarrow E + E \mid E * E \mid \text{id}$. This is not an operator grammar, because more than one precedence relation holds between certain pair of terminals. For example $+ \text{ and } +$, $+ \text{ and } *$, and $+ \text{ and } <$.

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars. A grammar is said to be operator precedence grammar if it has two properties:

No R.H.S. of any production has $a \epsilon$.

No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

$a \triangleright b$ means that terminal "a" has the higher precedence than terminal "b".

$a \triangleleft b$ means that terminal "a" has the lower precedence than terminal "b".

$a \approx b$ means that the terminal "a" and "b" both have same precedence.

Constructing the Canonical LR Parsing Tables For removing such contradictory results as we have seen in above example, it is possible to carry more information in the state that will allow us to rule out some of these invalid reductions. The extra information is incorporated into the state by redefining items to include a terminal symbols as a second component. The general form of item becomes $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha \cdot \beta$ is a production and a is a terminal or a right endmarker $\$$. We call such an object as LR (1) item. The 1 refers to the length of second component, called the lookahead of the item. The lookahead has no effect in an item of the form $[A \rightarrow \alpha \cdot \beta, a]$, where β is not ϵ , but an item of the form $[A \rightarrow \alpha \cdot \beta, a]$ calls for a reduction by $A \rightarrow \alpha$ only if the next input symbol is a .

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty stack.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side.	Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parse tree.

Syntax-directed translation (SDT)

refers to a method or compiler implementation where the source language translation is completely driven by the parser, "Le" based on the syntax of the language. The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed. SDT can be a separate phase of a compiler or one can augment conventional grammar with information to control the semantic analysis and translation. Such grammars are called attribute grammars. We augment a grammar by associating attributes with each grammar symbol that describes its properties. With each production in a grammar, we give semantic rules/ actions, which describe how to compute the attribute values associated with each grammar symbol in a production. The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In n, any cases, translation can be done during parsing without building an explicit tree. A class of syntax-directed translations called "L-attributed translations" (L for left-to-right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up.

A syntax-directed definition (SDD)

associates a semantic rule with each grammar production; the rule states how attributes are calculated. Conceptually, each node may have multiple attributes. Perhaps a struct/record/dictionary is used to group many attributes. Attributes may be concerned e.g. with data type, numeric value, symbol identification, code fragment, memory address, machine register choice. In a syntax-directed definition, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values. A SDD is a context-free grammar, plus attributes, and rules attached to the productions of the grammar stating how attributes are computed. For example, consider the Desk calculator's Arithmetic Expression example:

All attributes in this example are Synthesized, val and lexical attribute names are purposely different, Subscripts distinguish occurrences of the same grammar symbol, and iv) practical rules may also have side effects.

Syntax Directed Translation Scheme (SDTS)

In order to perform semantic analysis we make use of a formalism called SDTS. It is a free grammar in which there are attributes associated with the grammar symbols and semantic actions enclosed within braces are inserted with the right hand sides of productions. They are useful for specifying translations during parsing. Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

SDT implement two important classes of SDD's: The underlying grammar is LR-parseable and the SDD is L-attributed. The underlying grammar is LL-parseable and the SDD is L-attributed.

Attribute Grammar

An attribute grammar is a Syntax-Directed Definition (SDD) in which the functions in the semantic rules cannot have side-effects (they can only evaluate values of attributes).

An attribute grammar is a device used to describe more of the structure of a programming language than can be described with a context-free grammar. An attribute grammar is an extension to a context-free grammar. The extension allows certain language rules to be conveniently described, such as type compatibility.

Types of Attribute Grammar

Based on the way the attributes get their values, they can be broadly divided into two categories:

1) S-Attributes: S-attributes stand for synthesized attributes whose values are derived from children (leaves) of grammar symbols. An S-attributed grammar is one that uses only synthesized attributes.

2) L-Attributes Grammar: An L-attributed grammar is one whose value at a node in a parse tree is defined in terms of attributes at parent or sibling of that node, i.e. the value of an inherited attribute is computed from the values of attributes at the siblings and parents of that node.

Mod4:

Annotated Parse Tree

The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree. Features of Annotated parse Tree: High level specification. Hides implementation details. Explicit order of evaluation is not specified.

Synthesised Attribute	Inherited Attribute
The value of synthesised attribute at a node is computed from the values of attribute of children of the node in the parse tree.	The value of inherited attribute is computed from the values of attributes of siblings.
S-attribute can be evaluated during bottom-up parsing (traversal) of a parse tree.	Inherited attributes can be evaluated during top-down traversal of a parse tree.
Synthesised attributes pass information up a parse tree.	Inherited attributes pass on information down the parse tree.
S-attributes are also called reference attributes (call by reference).	Inherited attributes are called value attributes (call by value).

L-Attributed Definitions

A syntax-directed definition is L-attributed if for every production $A \rightarrow X_1 X_2 \dots X_n$, and each inherited attribute of X_j for $1 \leq j \leq n$, the attributes (both inherited as well as synthesised) of the symbols X_1, X_2, \dots, X_{j-1} , (i.e. the symbols to the left of X_j in the production, and the inherited attributes of A). The syntax-directed definition above is an example of the L-attributed definition, because the inherited attribute, L-type depends on T-type, and T is to the left of L in the production D TL. Similarly, the inherited attribute L-type depends on the inherited attribute L-type, and L is parent of L4 in the production L \rightarrow L4 id. When translation carried-out during parsing, the order in which the semantic rules are evaluated by the parser must be explicitly specified. Hence, instead of using the syntax-directed definitions, we use syntax-directed translation schemes to specify the translations. Syntax-directed definitions are more abstract specifications for translations; therefore, they hide many implementation details, freeing the user from having to explicitly specify the order in which translation takes place.

Type Checking

Type checking is one of the most important semantic aspects of compilation. Essentially, type checking:

Allows the programmer to limit what types may be used in certain circumstances, Aligns types to values, and Determines whether these values are used in an appropriate manner. Type checking also helps in deciding which code to be generated as in case of arithmetic expressions. Type checking is involved in large parts of the annotated syntax tree.

Static type checking: It is carried out at compile time it must be possible to compute all the information required at compile time. Eg: type checks, flow of control checks, uniqueness checks 2. Dynamic type checking:

Type checking is carried out while the program is running this is obviously less efficient but a language permits the type of variable to be determined at run time then one must use dynamic type checking.

Storage Organisation

In run-time environment, the executing target program runs in its own logical address space in which each program value has a location. The management and organisation of this logical address is shared between the compiler, operating system, and target machine. The OS maps the logical address into physical addresses, which are usually spread throughout memory. Compiler should consequently perform the following steps: Allocate memory for a variable. Initialize allocated raw memory with a default value. Allow a programmer accessing this memory. Clean (if necessary) and free memory once the corresponding resource is no longer used. Finally, the freed memory should be utilized and marked as ready for further reuse.

The run-time storage might be divided into following types: Code Segment, Data Segment (Holds Global Data), Stack where the local variables and other temporary information is stored.

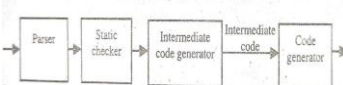
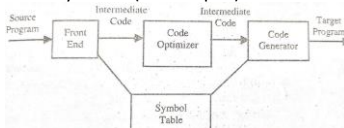
Three address code: QUADRUPLES, TRIPLES, INDIRECT TRIPLES

Quadruple - It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage - Easy to rearrange code for global optimization. One can quickly access value of temporary variables using symbol table.

Triples: The contents of the operand 1, operand2, and result fields are therefore normally the pointers to the symbol records for the names represented by these fields. Hence, it becomes necessary to enter temporary names into the symbol table as they are created.

Indirect Triples - This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

<p>Static Storage Allocation: If the size of every data item can be determined by the compiler and if recursion procedure calls are not permitted, then the space for all programs and data can be allocated at compile time, i.e., statically. Moreover, the association of names to location can also be done 'statically' in this case. In static allocation, names are bound to storage as the program is compiled, so there is no need for a runtime support package. Since the bindings do not change at run time, every time a procedure is activated, its names are bounded to the same storage. This property allows the values of the local names to be retained across activations of a procedure. That is, when control returns to a procedure, the values of the locals are the same as they are when control left the last time.</p> <p>Dynamic Storage Allocation: If the programming language permits either recursive procedures or data structure whose size is adjustable, then some sort of dynamic storage management is necessary. Types of Dynamic Storage Allocation There are two kinds of dynamic storage allocation: Stack Storage Allocation: Storage is organised as a stack and activation records are pushed and popped as activation begin and end respectively. Locals are contained in activation records so they are bound to fresh storage in each activation. Recursion is supported in stack allocation.</p> <p>ii) Heap Storage Allocation: Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required. Recursion is supported. There are various properties of heap allocation which are as follows – Space Efficiency– A memory manager should minimize the total heap space needed by a program. Program Efficiency– A memory manager should make good use of the memory subsystem to allow programs to run faster. As the time taken to execute an instruction can vary widely depending on where objects are placed in memory. Low Overhead– Memory allocation and deallocation are frequent operations in many programs. These operations must be as efficient as possible. That is, it is required to minimize the overhead. The fraction of execution time spent performing allocation and deallocation</p> <p>Intermediate Code Generation f The use of syntax-directed translation is not restrictive compiling. In many compilers the source code is translated to a language; which is intermediate in complexity between a (high-level) programming language and a machine code. Such a language is therefore called intermediate code or intermediate text. It is possible to translate directly from source to machine or assembly language in syntax – directed way but, as we have mentioned, doing so makes generation of optimal, or even relatively good, code a difficult task. The reason efficient machine or assembly language is hard to generate is that one is immediately forced to choose particular register to hold the result of each computation, making the efficient use of registers difficult.</p>  <p>Activation Record The segment of the stack that contains the variables for a function is the "activation record" or "stack frame" of that function. In addition to storing local variables, the activation record for a function also stores some saved register values. A special register, called the Frame Pointer (FP), points to the beginning of the current frame. Another register, the "Stack Pointer" (SP), contains the address of the first unused location on the stack. Simple Java stacks will "grow" from large addresses to small addresses. Thus, when we push items onto the stack, we will subtract from the stack pointer, and when we pop items off the stack, we will add to the 'stack pointer'.</p> <p>Assignment statement: Expression can be type of integer, real, array and record as part of translation of assignment into 3 address code, we show how names can be looked up in the symbol table and how elements of arrays and records can 1. Name of the Symbol Table The translation scheme The lexeme for the name represented by id is, given by attribute id.name. Operation lookup (id.name) checks if there is an entry for this occurrence of the name in the symbol table. 2. Reusing Temporary Names We have been going along assuming that newtemp m generates a new temporary name each time a temporary is needed. It is useful, especially in optimising compilers, to actually create a distance name each time newtemp is called. However, the temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their values.</p>	<p>MODS:</p> <p>PRINCIPAL SOURCES OF OPTIMISATION A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first. Function-Preserving Transformations There are a number of ways in which a compiler can improve a program without changing the function it computes. Function preserving transformations examples: Common sub expression elimination, Copy propagation, Dead-code elimination, Constant folding, Common Sub expressions elimination: An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value. 2. Copy Propagation: Assignments of the form f = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x. 3. Dead-Code Eliminations: A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. 4. Constant folding: Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code. 5. Loop Optimizations: In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization: Ø Code motion, which moves code outside a loop; Ø Induction-variable elimination, which we apply to replace variables from inner loop. Ø Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition. 6. Code Motion: An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. 7. Induction Variables : Loops are usually processed inside out. For example consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables. 8. Reduction In Strength: Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x² is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.</p> <p>DAG Representation of Basic Blocks A useful data structure for automatically analysing basic blocks is a directed acyclic graph (hereafter called a DAG). A DAG is a directed graph with no cycles, which gives a picture of how the value computed by each statement in a basic block is used in subsequent statements in the block. Constructing a DAG from three-address statements is a good way of determining common sub expressions within a block, determining which names are used inside the block but evaluate outside the block, and determining which statements of the block could have their value used outside the block. A DAG for basic block is a directed acyclic graph with the following labels on nodes: The leaves of graph are labeled by unique identifier and that identifier can be variable names or constants. Interior nodes of the graph is labeled by an operator symbol. Nodes are also given a sequence of identifiers for labels to store the computed value. DAGs are a type of data structure. It is used to implement transformations on basic blocks. DAG provides a good way to determine the common sub-expression. It gives a picture representation of how the value computed by the statement is used in subsequent statements.</p>	<p>Optimization of basic blocks There are 2 types of basic block optimization they are 1. structure preserving transformation: The primary structure preserving transformation on basic block are. 1. Common sub expression Elimination : -Dead code Elimination -Renaming of temporary variables -Interchange of two independent adjacent statement 2. Algebraic transformations : Algebraic Transformations: Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones like reduction in strength.</p> <p>Machine Independent Code Optimization High-level language constructs can introduce substantial run-time overhead if one naively translates each construct independently into machine code. Elimination of unnecessary instructions in object code, or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimisation". In local code optimization one can work on code improvement within a basic block. Most global optimisations are based on data-flow analyses, which are algorithms to gather information about a program. The results of data-flow analyses all have the same form - for each instruction in the program, they specify some property that must hold every time that instruction is executed. The analyses differ in the properties they compute.</p> <p>Code Generation The final phase in our compiler -model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently. Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine (often a computer).</p>  <p>Issues in the Design of a Code Generator Tasks which are typically part of a sophisticated COMPILER "code generation" phase include: I) Input to Code Generator: The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation. There are several choices for the intermediate language, including: i) Linear representations such as postfix notation, ii) three address representation such as quadruples, iii) Virtual machine representations such as syntax trees and dags. 2) Target Programs: The output of the code generator is the target program. The output may take on a variety of forms: Absolute machine language, Relocatable machine language or assembly language. Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A program can be compiled and executed as: Producing a relocatable machine language program as output allows subprograms to be compiled separately. 3) Memory Management: Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and code generator. A name in a three-address statement refers to a symbol table entry for the name. If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the "back patching". 4. Instruction Selection: This task tells which instructions to use. The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each 5. Register allocation : The allocation of variables to processor registers. • Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore efficient utilization of : register is • particularly important in generating good code. The use of registers is often subdivided into two subproblems: During register allocation select the set of variables that will reside in registers at a point in the program. During a subsequent register assignment phase, pick the specific register that a variable will reside in.</p>	<p>Algorithm for Code Generation An algorithm for code generation is as below. For every three-address statement of the form x = y op z in the basic block do: Call getreg() to obtain the location L in which the computation y op z should be performed. /* This requires passing the three-address statement x = y op z as a parameter to getreg(), which can be done by passing the index of this statement in the quadruple array. Obtain the current location of the operand y by consulting its address descriptor, and if the value of y is currently both in the memory location as well as in the register, then prefer the register. If the value of y is currently not available in L, then generate an instruction MOV y, L (where y is assumed to represent the current location of y). Generate the instruction OP z, L, and update the address descriptor of x to indicate that x is now available in L, and if L is in a register, then update its descriptor to indicate that it will contain the run-time value of x. If the current values of y and/or z are in the register, and we have no further uses for them, and they are not live at the end of the block, then alter the register descriptor to indicate that after the execution of the statement x = y op z, those registers will no longer contain y and/or z.</p> <p>Simple Code Generator A simple code generator generates the target code for the three-address statements. The main issue during code generation is the utilisation of registers since the number of registers available is limited. The code generation algorithm takes the sequence of three-address statements as input, and assumes that for each operator, there exists a corresponding operator in target language. The machine code instruction takes, the required operands in registers, performs the operation and stores the result in a register. Register and address descriptors are used to keep track of register contents and addresses.</p> <p>Construct the recursive descent parser for handling the arithmetic expression : The recursive descent parser to evaluate syntactically valid arithmetic expressions has five methods corresponding to each of the five non-terminal symbols of this grammar. A tokenizer is not used for this parser. Instead, each method gets input characters from a StringBuilder parameter called source. You can assume that the source argument does not contain any white space or any other characters not part of the expression except for at least one "sentinel" character after the end of the expression to mark the end. In other words, any proper prefix of the argument can contain only characters from the set {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '(', ')', '+', '-', '*', '/', '}'}. You can further assume that the input is syntactically valid, so that no error checking is necessary. Here are few more hints: A string s1 being a proper prefix of a string s2 means both that the length of s1 is strictly less than (less than but not equal to) the length of s2 and that s1 is a prefix of s2. The instance methods charAt(int) and deleteCharAt(int) defined in StringBuilder will be useful to manipulate the input. The static methods isDigit(char) and digit(char, int) in class Character can be used to check if a character is a digit and to convert a digit character into the corresponding integer value, respectively.</p>
---	---	--	--