UNIVERSITY OF AGRICULTURE, ABEOKUTA

DEPARTMENT OF COMPUTER SCIENCE

CSC 415 - COMPILING TECHNIQUES

Semester:-	Second
***Course Unit*:-**	3
Course Lecturer:-	DR. A. S. SODIYA
Contact:-	3hrs per week

Course Objectives:-

At the end of this course, students are expected to have understood:-

- ✓ Meaning and concepts of Compilers
- ✓ History and types of compilers
- ✓ Compiling stages
- ✓ Regular expressions
- ✓ Context-free grammar
- ✓ Syntax analysis
- ✓ Finite automata
- ✓ Top-down parsing
- ✓ Bottom-up parsing
- ✓ Code generation
- ✓ Code optimisation

Compiled notes on Compiling Techniques

Section 1:- INTRODUCTION TO COMPILER

DEFINITION OF A COMPILER

A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a language translator, source to source translator, or language converter. A language rewriter is usually a program that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest a lot of time ensuring the correctness of their software.

The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

HISTORY

Software for early computers was primarily written in assembly language for many years. Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler. The very limited memory capacity of early computers also created many technical problems when implementing a compiler.

Towards the end of the 1950s, machine-independent programming languages were first proposed. Subsequently, several experimental compilers were developed. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957. COBOL was an early language to be compiled on multiple architectures, in 1960.

In many application domains the idea of using a higher level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers have become more and more complex.

Early compilers were written in assembly language. The first self-hosting compiler capable of compiling its own source code in a high-level language was created for Lisp by Tim Hart and Mike Levin at MIT in 1962. Since the 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been popular choices for implementation language. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must be compiled either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) compiled by running the compiler in an interpreter.

Section 2: REGULAR EXPRESSION

In computing, a regular expression, also referred to as regex or regexp, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

Utilities provided by Unix distributions including the editor ed and the filter grep were the first to popularize the concept of regular expressions. Regular expressions are used by many text editors, utilities, and programming languages to search and manipulate text based on patterns. Some of these languages, including Perl, Ruby, Awk, and Tcl, have fully integrated regular expressions into the syntax of the core language itself. Others like .NET languages, Java, and Python instead provide regular expressions through standard libraries. For other languages, such as C and C++, non-core libraries are available (However, the upcoming version C++11 introduces regular expressions as part of its Standard Libraries).

Many modern computing systems provide wildcard characters in matching filenames from a file system. This is a core capability of many command-line shells and is also known as globbing. Wildcards differ from regular expressions in generally expressing only limited forms of patterns.

Basic concepts

A regular expression, often called a pattern, is an expression that describes a set of strings. They are usually used to give a concise description of a set, without having to list all elements. For example, the set containing the three strings "Handel", "Händel", and "Haendel" can be described by the pattern H(ä|ae?)ndel (or alternatively, it is said that the pattern matches each of the three strings). In most formalisms, if there is any regex that matches a particular set then there is an infinite number of such expressions.

SYNTAX

A number of special characters or meta characters are used to denote actions or delimit groups; but it is possible to force these special characters to be interpreted as normal characters by preceding them with a defined escape character, usually the backslash "\". For example, a dot is normally used as a "wild card" metacharacter to denote any character, but if preceded by a backslash it represents the dot character itself. The pattern c.t matches "cat", "cot", "cut", and non-words such as "czt" and "c.t"; but c\.t matches only "c.t". The backslash also escapes itself, i.e., two backslashes are interpreted as a literal backslash character.

POSIX

POSIX Basic Regular Expressions

Traditional Unix regular expression syntax followed common conventions but often differed from tool to tool. The IEEE POSIX Basic Regular Expressions (BRE) standard (released alongside an alternative flavor called Extended Regular Expressions or ERE) was designed mostly for backward compatibility with the traditional (Simple Regular Expression) syntax but provided a common standard which has since been adopted as the default syntax of many Unix regular expression tools, though there is often some variation or additional features. Many such tools also provide support for ERE syntax with command line arguments.

In the BRE syntax, most characters are treated as literals — they match only themselves (e.g., a matches "a").

Note that what the POSIX regular expression standards call character classes are commonly referred to as POSIX character classes in other regular expression flavors which support them. With most other regular expression flavors, the term character class is used to describe what POSIX calls bracket expressions.

SIMPLE REGULAR EXPRESSIONS

Simple Regular Expressions is a syntax that may be used by historical versions of application programs, and may be supported within some applications for the purpose of providing backward compatibility. It is deprecated.

LAZY QUANTIFICATION

The standard quantifiers in regular expressions are greedy, meaning they match as much as they can, only giving back as necessary to match the remainder of the regex.

USES

Regular expressions are useful in the production of syntax highlighting systems, data validation, and many other tasks.

While regular expressions would be useful on search engines such as Google, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex. Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public. Notable exceptions: Google Code Search, Exalead.

Section 3:- CONTEXT FREE GRAMMAR

A Context Free Grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

In formal language, a **context-free grammar** (**CFG**), sometimes also called a **phrase structure grammar**, is a grammar that naturally generates a formal language in which clauses can be nested inside clauses arbitrarily deeply, but where grammatical structures are not allowed to overlap. *CFGs* can be expressed by Backus–Naur Form, or *BNF*.

Like regular expressions, context-free grammars are used to describe sets of strings, i.e., languages.

Additionally, a context-free grammar also defines structure on the strings in the language it defines.

A language is defined over some alphabet (Denoted by ε), for example the set of tokens produced by a lexer or the set of alphanumeric characters. The symbols in the alphabet are called terminals.

A context-free grammar recursively defines several sets of strings. Each set is denoted by a name, which is called a non-terminal. The set of non-terminals is disjoint from the set of terminals. One of the non-terminals is chosen to denote the language described by the grammar. This is called the start symbol of the grammar. The sets are described by a number of productions. Each production describes some of the possible strings that are contained in the set denoted by a non-terminal.

A production has the form

$$V \rightarrow w$$

Where *V* is a single non-terminal symbol, and *w* is a string of terminals and/or non-terminals (*w* can be empty). These rewriting rules applied successively produce a parse tree, where the non-terminal symbols are nodes, the leaves are the terminal symbols, and each node expands by the production into the next level of the tree. The tree describes the nesting structure of the expression. *V* can therefore change while *w* is fixed (can't change).

In a context free grammar the left hand side of a production rule is always a single non-terminal symbol. In a general grammar, it could be a string of terminal and/or non-terminal symbols. The grammars are called *context free* because – since all rules only have a non-terminal on the left hand side – one can always replace that non-terminal symbol with what is on the right hand side of the rule. The *context* in which the symbol occurs is therefore not important.

$$\text{N} \rightarrow \text{X1 . . .Xn}$$

Where N is a non-terminal and X1 . . .Xn are zero or more symbols, each of which is either a terminal or a non-terminal. The intended meaning of this notation is to say that the set denoted by N contains strings that are obtained by concatenating strings from the sets denoted by X1....Xn. In this setting, a terminal denotes a singleton set, just like alphabet characters in regular expressions. We will, when no confusion is likely, equate a non-terminal with the set of strings it denotes
Some examples:

$$\text{A} \rightarrow \text{a}$$

Says that the set denoted by the non-terminal A contains the one-character string a.

$$\text{A} \rightarrow \text{aA}$$

says that the set denoted by A contains all strings formed by putting an *a* in front of a string taken from the set denoted by A. Together, these two productions indicate that A contains all non-empty sequences of *a*s and is hence (in the absence of other productions) equivalent to the regular expression a^+.

We can define a grammar equivalent to the regular expression a* by the two productions

$$B \rightarrow$$
$$B \rightarrow aB$$

where the first production indicates that the empty string is part of the set B. Compare this grammar with the definition of s* in figure below

Productions with empty right-hand sides are called empty productions. These are sometimes written with an e on the right hand side instead of leaving it empty.

So far, we have not described any set that could not just as well have been described using regular expressions. Context-free grammars are, however, capable of expressing much more complex languages. It is noted that the language $\{a_n b_n \mid n > 0\}$ is not regular. It is, however, easily described by the grammar

$$S \rightarrow$$
$$S \rightarrow aSb$$

The second production ensures that the *a*s and *b*s are paired symmetrically around the middle of the string, ensuring that they occur in equal number.

The examples above have used only one non-terminal per grammar. When several non-terminals are used, we must make it clear which of these is the start symbol. By convention (if nothing else is stated), the non-terminal on the left-hand side of the first production is the start symbol. As an example, the grammar

$$T \rightarrow R$$
$$T \rightarrow aTa$$
$$R \rightarrow b$$
$$R \rightarrow bR$$

has T as start symbol and denotes the set of strings that start with any number of as followed by a non-zero number of bs and then the same number of as with which it started.

Sometimes, a shorthand notation is used where all the productions of the same non-terminal are combined to a single rule, using the alternative symbol (|) from regular expressions to separate the right-hand sides. In this notation, the above grammar would read

$$T \rightarrow R \mid aTa$$
$$R \rightarrow b \mid bR$$

There are still four productions in the grammar, even though the arrow symbol → is only used twice.

Each sub-expression of the regular expression is numbered and sub-expression s_i is assigned a non-terminal N_i. The productions for N_i depend on the shape of s_i as shown in the table above.

HOW TO WRITE CONTEXT FREE GRAMMARS

As hinted above, a regular expression can systematically be rewritten as a context free grammar by using a non-terminal for every sub-expression in the regular expression and using one or two productions for each non-terminal. The construction is shown in figure 3.1. So, if we can think of a way of expressing a language as a regular expression, it is easy to make a grammar for it. However, we will also want to use grammars to describe non-regular languages. An example is the kind of arithmetic expressions that are part of most programming languages (and also found on electronic calculators). Such expressions can be described by grammar 3.2. Note that, as mentioned in section 2.10, the matching parenthesis can’t be described by regular expressions, as these can’t “count” the number of unmatched opening parenthesis at a particular point in the string. If we didn’t have parenthesis in the language, it could be described by the regular expression

num ((+|-|*|/) num)*

Most constructions from programming languages are easily expressed by context free grammars. In fact, most modern languages are designed this way.

To generate a string of terminal symbols from a CFG, we:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left hand side, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all non-terminals have been replaced by terminal symbols.

A) Context Free Grammar for if……..else………if statement

<start> ---à if (<expr>) <stmt> else <stmt> if (<expr>) <stmt>

<stmt> = Statement

<expr> = Expression

Non-terminal symbols are **<stmt>** and **<expr>**

Terminal symbols are if and else

<start> is the Start symbol

B) Context Free Grammar for Switch statement

<start> ---à switch (<expr>) <stmt>

< stmt> ---à case <expr> : <stmt>

<stmt> = Statement

<expr> = Expression

Non-terminal symbols are **<stmt>** and **<expr>**

Terminal symbols are **switch** and **case**

The symbol **<stmt>** at the end of the first line is the production

<start> is the Start symbol

C) Context Free Grammar for Do………while statement

<start> ---à do <stmt> while (<expr>) ;

<stmt> = Statement

<expr> = Expression

Non-terminal symbols are **<stmt>** and **<expr>**

Terminal symbols are **do** and **while**

<start> is the Start symbol

Section 4:- SYNTAX ANALYSIS

The purpose of syntax analysis (also known as parsing) is to recombine these tokens. Not back into a list of characters, but into something that reflects the structure of the text. This "something" is typically a data structure called the syntax tree of the text. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis, and if the leaves are read from left to right, the sequence is the same as in the input text. Hence, what are important in the syntax tree is how these leaves are combined to form the structure of the tree and how the interior nodes of the tree are labeled.

In addition to finding the structure of the input text, the syntax analysis must also reject invalid texts by reporting syntax errors.

As syntax analysis is less local in nature than lexical analysis, more advanced methods are required. We, however, use the same basic strategy: A notation suitable for human understanding is transformed into a machine-like low-level notation suitable for efficient execution. This process is called parser generation.

The notation we use for human manipulation is context-free grammars1, which is a recursive notation for describing sets of strings and imposing a structure on each such string. This notation can in some cases be translated almost directly into recursive programs, but it is often more convenient to generate stack automata.

These are similar to the finite automata used for lexical analysis but they can additionally use a stack, which allows counting and non-local matching of symbols.

We shall see two ways of generating such automata. The first of these, LL (1), is relatively simple, but works only for a somewhat restricted class of grammars.
The SLR construction, which we present later, is more complex but accepts a wider class of grammars. Sadly, neither of these works for all context-free grammars.
Tools that handle all context-free grammars exist, but they can incur a severe speed penalty, which is why most parser generators restrict the class of input grammars.

Section 5:- FINITE AUTOMATA

A finite automaton is, in the abstract sense, a machine that has a finite number of states and a finite number of transitions between these. A transition between states is usually labelled by a character from the input alphabet, but we will also use transitions marked with e, the so-called epsilon transitions.
A finite automaton can be used to decide if an input string is a member in some particular set of strings. To do this, we select one of the states of the automaton as the starting state. We start in this state and in each step; we can do one of the following:

- Follow an epsilon transition to another state, or
- Read a character from the input and follow a transition labeled by that character.

When all characters from the input are read, we see if the current state is marked as being accepting. If so, the string we have read from the input is in the language defined by the automaton.
We may have a choice of several actions at each step: We can choose between either an epsilon transition or a transition on an alphabet character, and if there are several transitions with the same symbol, we can choose between these. This makes the automaton nondeterministic, as the choice of action is not determined solely by looking at the current state and input. It may be that some choices lead to an accepting state while others do not. This does, however, not mean that the string is sometimes in the language and sometimes not: We will include a string in the language if it is possible to make a sequence of choices that makes the string lead to an accepting state.
You can think of it as solving a maze with symbols written in the corridors.

If you can find the exit while walking over the letters of the string in the correct order, the string is recognized by the maze.

Non Deterministic Finite State Automata

A nondeterministic finite automaton consists of a set S of states. One of these states, $s_0 \epsilon$ S, is called the starting state of the automaton and a subset F _ S of the states are accepting states. Additionally, we have a set T of transitions. Each transition t connects a pair of states s_1 and s_2 and is labeled with a symbol, which is either a character *c* from the alphabet S, or the symbol *e*, which indicates an epsilon-transition. A transition from state *s* to state *t* on the symbol *c* is written as $s^c t$.

Starting states are sometimes called initial states and accepting states can also be called final states (which is why we use the letter F to denote the set of accepting states). We use the abbreviations FA for finite automaton, NFA for nondeterministic finite automaton and (later in this chapter) DFA for deterministic finite automaton.

We will mostly use a graphical notation to describe finite automata. States are denoted by circles, possibly containing a number or name that identifies the state. This name or number has, however, no operational significance; it is solely used for identification purposes. Accepting states are denoted by using a double circle instead of a single circle. The initial state is marked by an arrow pointing to it from outside the automaton.

A transition is denoted by an arrow connecting two states. Near its midpoint, the arrow is labeled by the symbol (possibly e) that triggers the transition. Note that the arrow that marks the initial state is not a transition and is, hence, not marked by a symbol.

Repeating the maze analogue, the circles (states) are rooms and the arrows (transitions) are one-way corridors. The double circles (accepting states) are exits, while the unmarked arrow to the starting state is the entrance to the maze.

A program that decides if a string is accepted by a given NFA will have to check all possible paths to see if any of these accepts the string. This requires either backtracking until a successful path found or simultaneous following all possible paths, both of which are too time-consuming to make NFAs suitable for efficient recognizers. We will, hence, use NFAs only as a stepping stone between regular expressions and the more efficient DFAs. We use this stepping stone because it makes the construction simpler than direct construction of a DFA from a regular expression.

Section 6:- **TOP-DOWN PARSING**

The syntax analysis phase of a compiler will take a string of tokens produced by the lexer, and from this construct a syntax tree for the string by finding a derivation of the string from the start symbol of the grammar.

This can be done by guessing derivations until the right one is found, but random guessing is hardly an effective method. Even so, some parsing techniques are based on "guessing" derivations. However, these make sure, by looking at the string, that they will always guess right. These are called predictive parsing methods. Predictive parsers always build the syntax tree from the root down to the leaves and are hence also called (deterministic) top-down parsers.

Other parsers go the other way: They search for parts of the input string that matches right-hand sides of productions and rewrite these to the left-hand non-terminals, at the same time building pieces of the syntax tree. The syntax tree is eventually completed when the string has been rewritten (by inverse derivation) to the start symbol. Also here, we wish to make sure that we always pick the "right" rewrites, so we get deterministic parsing. Such methods are called bottom-up parsing methods.

We will in the next sections first look at predictive parsing and later at a bottom-up parsing method called SLR parsing.

Predictive parsing

We see that, to the left of the rewritten non-terminals, there are only terminals. These terminals correspond to a prefix of the string that is being parsed. In a parsing situation, this prefix will be the part of the input that has already been read. The job of the parser is now to choose the production by which the leftmost unexpanded non-terminal should be rewritten.

Our aim is to be able to make this choice deterministically based on the next unmatched input symbol.

If we look at the third line in figure 3.6, we have already read two as and (if the input string is the one shown in the bottom line) the next symbol is a b. Since the right-hand side of the production

$$T \rightarrow aTc$$

starts with an *a*, we obviously can not use this. Hence, we can only rewrite T using the production

$$T \rightarrow R$$

Recursive descent

As the name indicates, recursive descent uses recursive functions to implement predictive parsing. The central idea is that each non-terminal in the grammar is implemented by a function in the program. Each such function looks at the next input symbol in order to choose a production. The right-hand side of the production is then parsed in the following way: A terminal is matched against the next input symbol. If they match, we move on to the following input symbol, otherwise an error is reported. A non-terminal is parsed by calling the corresponding function.

As an example, figure 3.16 shows pseudo-code for a recursive descent parser for grammar 3.9. We have constructed this program by the following process:

We have first added a production T0→T$ and calculated FIRST and FOLLOW for all productions.

T0 has only one production, so the choice is trivial. However, we have added a check on the next input symbol anyway, so we can report an error if it isn’t in FIRST(T 0). This is shown in the function parseT’.

For the parseT function, we look at the productions for T. FIRST(R) = {b}, so the production T → R is chosen on the symbol b. Since R is also Nullable, we must choose this production also on symbols in FOLLOW(T), i.e., c or $.

FIRST(aTc) = {a}, so we select T ! aTc on an a. On all other symbols we report an error.

```
function parseT’ =
if next = ’a’ or next = ’b’ or next = ’$’ then
parseT ; match(’$’)
else reportError
```

```
function parseT =
if next = ’b’ or next = ’c’ or next = ’$’ then
parseR
else if next = ’a’ then
match(’a’) ; parseT ; match(’c’)
else reportError
function parseR =
if next = ’c’ or next = ’$’ then
doNothing
else if next = ’b’ then
match(’b’) ; parseR
else reportError
```

For parseR, we must choose the empty production on symbols in FOLLOW(R) (c or $). The production R→bR is chosen on input b. Again, all other symbols produce an error.
The function match takes as argument a symbol, which it tests for equality with the next input symbol. If they are equal, the following symbol is read into the variable next. We assume next is initialized to the first input symbol before parseT’ is called.
The program in figure 3.16 only checks if the input is valid. It can easily be extended to construct a syntax tree by letting the parse functions return the subtrees for the parts of input that they parse.

Section 7:- BOTTOM-UP PARSING

Bottom-up parsing is a strategy for analyzing unknown information that attempts to identify the most fundamental units first, and then to infer higher-order structures from them. As the name suggests, bottom-up parsing works in the opposite direction from top-down. A top-down parser begins with the start symbol at the top of the parse tree and works downward, driving productions in forward order until it gets to the terminal leaves. A bottom-up parse starts with the string of terminals itself and builds from the leaves upward, working backwards to the start symbol by applying the productions in reverse. Along the way, a bottom-up parser searches for substrings of the working string that match the right side of some production. When it finds such a substring, it *reduces* it, i.e., substitutes the left side non-terminal for the matching right side. The goal is to reduce all the way up to the start symbol and report a successful parse.

Types of Bottom-up parsers

The common classes of bottom-up parsers are:

- LR parser
    - LR(0) - No lookahead symbol
    - SLR(1) - Simple with one lookahead symbol
    - LALR(1) - Lookahead bottom up, not as powerful as full LR(1) but simpler to implement. YACC deals with this kind of grammar.
    - LR(1) - Most general grammar, but most complex to implement.
    - LR(*n*) - (where *n* is a positive integer) indicates an LR parser with *n* lookahead symbols; while grammars can be designed that require more than 1 lookahead, practical grammars try to avoid this because increasing *n* can theoretically require exponentially more code and data space (in practice, this may not be as bad). Also, the class of LR(*n*) languages is the same as that of LR(1) languages.
- Precedence parsers
    - Simple precedence parser
    - Operator-precedence parser
    - Extended precedence parser

Bottom-up parsing can also be done by backtracking.

Shift-reduce parsing is the most commonly used and the most powerful of the bottom-up techniques. It takes as input a stream of tokens and develops the list of productions used to build the parse tree, but the productions are discovered in reverse order of a top-down parser. Like a table-driven predictive parser, a bottom-up parser makes use of a stack to keep track of the position in the parse and a parsing table to determine what to do next.

Shift and reduce

A shift-reduce parser uses a stack to hold the grammar symbols while awaiting reduction. During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the non-terminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input.

The parser is a stack automaton which is in one of several discrete states. In reality, in the case of LR parsing, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

Algorithm: Shift-reduce parsing

1. Start with the sentence to be parsed as the initial sentential form
2. Until the sentential form is the start symbol do:
    1. Scan through the input until we recognise something that corresponds to the RHS of one of the production rules (this is called a handle)
    2. Apply a production rule in reverse; i.e., replace the RHS of the rule which appears in the sentential form with the LHS of the rule (an action known as a reduction)

In step 2.1 above we are "shifting" the input symbols to one side as we move through them; hence a parser which operates by repeatedly applying steps 2.1 and 2.2 above is known as a shift-reduce parser.

A shift-reduce parser is most commonly implemented using a stack, where we proceed as follows:

- start with an empty stack
- a "shift" action corresponds to pushing the current input symbol onto the stack
- a "reduce" action occurs when we have a handle on top of the stack. To perform the reduction, we pop the handle off the stack and replace it with the non-terminal on the LHS of the corresponding rule.

Example

Let’s trace the operation of a shift-reduce parser in terms of its actions (shift or reduce) and its data structure (a stack). The chart below traces a parse of (id+id) using the following grammar:

S –> E

E –> T | E + T

T –> id | (E)

Solution

PARSE STACK	**REMAINING INPUT**	**PARSER ACTION**
	(id + id)$	Shift (push next token from input on stack, advance input)
(	id + id)$	Shift
(id	+ id)$	Reduce: T –> id (pop right-hand side of production off stack, push left-hand side, no change in input)
(T	+ id)$	Reduce: E –> T
(E	+ id)$	Shift
(E+	id)$	Shift
(E+ id	)$	Reduce: T –> id
(E + T	)$	Reduce: E –> E + T (Ignore: E –> T)
(E	)$	Shift

(E)	$	Reduce: T –> (E)
T	$	Reduce: E –> T
E	$	Reduce: S –> E
S	$	

In the above parse on step 7, we ignored the possibility of reducing E –> T because that would have created the sequence (E + E on the stack which is not a *viable prefix* of a right sentential form. Formally, viable prefixes are the set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser, i.e. prefixes of right sentential forms that do not extend past the end of the rightmost handle. Basically, a shift-reduce parser will only create sequences on the stack that can lead to an eventual reduction to the start symbol. Because there is no right-hand side that matches the sequence (E + E and no possible reduction that transforms it to such, this is a dead end and is not considered.

Section 8:- **CODE GENERATION**

In computer science, 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).

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the backend) needs to change from target to target.

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three address code. Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program. (For example, a peephole optimization pass would not likely be called "code generation", although a code generator might incorporate a peephole optimization pass.)

MAJOR TASKS IN CODE GENERATION

In addition to the basic conversion from an intermediate representation into a linear sequence of machine instructions, a typical code generator tries to optimize the generated code in some way. The generator may try to use faster instructions, use fewer instructions, exploit available registers, and avoid redundant computations.

Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

Instruction selection: which instructions to use.

Instruction scheduling: in which order to put those instructions. Scheduling is a speed optimization that can have a critical effect on pipelined machines.

Register allocation: the allocation of variables to processor registers.

Instruction selection is typically carried out by doing a recursive post order traversal on the abstract syntax tree, matching particular tree configurations against templates; for example, the tree W := ADD(X,MUL(Y,Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t1 := X and t2 := MUL(Y,Z), and then emitting the instruction ADD W, t1, t2.

In a compiler that uses an intermediate language, there may be two instruction selection stages: one to convert the parse tree into intermediate code, and a second phase much later to convert the intermediate code into instructions from the instruction set of the target machine. This second phase does not require a tree traversal; it can be done linearly, and typically involves a simple replacement of intermediate-language operations with their corresponding opcodes. However, if the compiler is actually a language translator (for example, one that converts Eiffel to C), then the second code-generation phase may involve building a tree from the linear intermediate code.

RUNTIME CODE GENERATION

When code generation occurs at runtime, as in just-in-time compilation (JIT), it is important that the entire process be efficient with respect to space and time. For example, when regular expressions are interpreted and used to generate code at runtime, a non-determistic finite state machine is often generated instead of a deterministic one, because usually the former can be created more quickly and occupies less memory space than the latter. Despite its generally generating less efficient code, JIT code generation can take advantage of profiling information that is available only at runtime.

RELATED CONCEPTS

The fundamental task of taking input in one language and producing output in a non-trivially different language can be understood in terms of the core transformational operations of formal language theory. Consequently, some techniques that were originally developed for use in compilers have come to be employed in other ways as well. For example, YACC (Yet Another Compiler Compiler) takes input in Backus-Naur form and converts it to a parser in C. Though it was originally created for automatic generation of a parser for a compiler, YACC is also often used to automate writing code that needs to be modified each time specifications are changed.
Many integrated development environments (IDEs) support some form of automatic source code generation, often using algorithms in common with compiler code generators, although commonly less complicated.

REFLECTION

In general, a syntax and semantic analyzer tries to retrieve the structure of the program from the source code, while a code generator uses this structural information (e.g., data types) to produce code. In other words, the former adds information while the latter loses some of the information. One consequence of this information loss is that reflection becomes difficult or even impossible. To counter this problem, code generators often embed syntactic and semantic information in addition to the code necessary for execution.

Section 10:- CODE OPTIMIZATION

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side-effects. The only difference visible to the code's user should be that it runs faster and/or consumes less memory. It is really a misnomer that the name implies you are finding an "optimal" solution— in truth, optimization aims to improve, not perfect, the result. It concerns with machine-independent code optimization
Many optimization problems are NP-complete and thus most optimization algorithms rely on heuristics and approximations. It may be possible to come up with a case where a particular algorithm fails to produce better code or perhaps even makes it worse. However, the algorithms tend to do rather well overall.

When and Where To Optimize

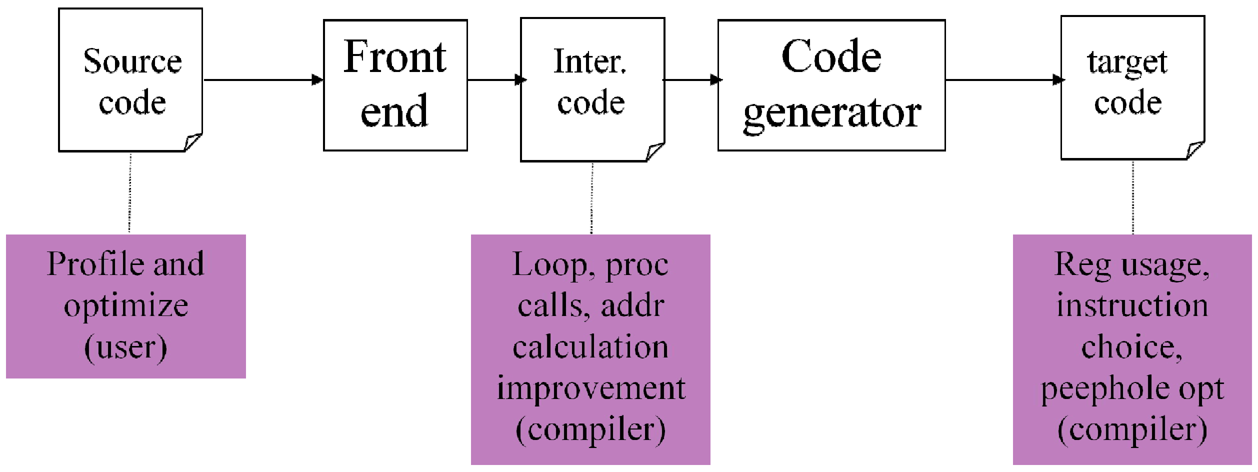
There are a variety of tactics for attacking optimization. Some techniques are applied to the intermediate code, to streamline, rearrange, compress, etc. in an effort to reduce the size of the abstract syntax tree or shrink the number of TAC instructions. Others are applied as part of final code generation—choosing which instructions to emit, how to allocate registers and when/what to spill, and the like. And still other optimizations may occur after final code generation, attempting to re-work the assembly code itself into something more efficient.

Optimization can be very complex and time-consuming; it often involves multiple sub-phases, some of which are applied more than once. Most compilers allow optimization to be turned off to speed up compilation.

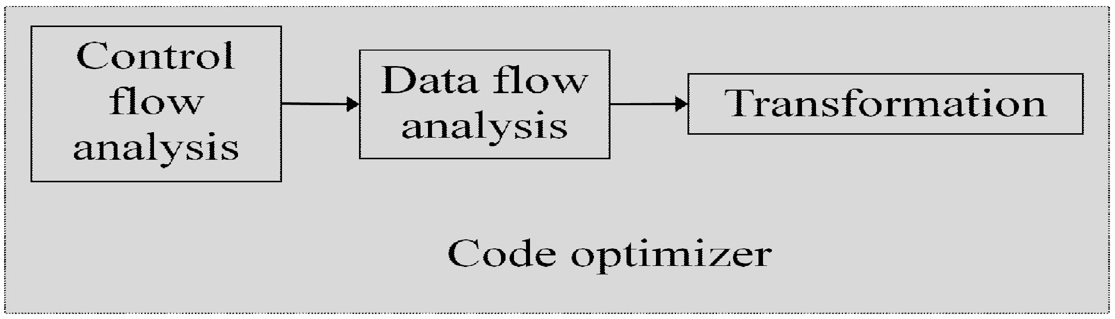
Criterion of code optimization

- Must preserve the semantic equivalence of the programs
- The algorithm should not be modified
- Transformation, on average should speed up the execution of the program
- Worth the effort: Intellectual and compilation effort spend on insignificant improvement.
- Transformations are simple enough to have a good effect

Optimization can be done in almost all phases of compilation.


Source code
Front end
Inter. code
Code generator
target code
Profile and optimize (user)
Loop, proc calls, addr calculation improvement (compiler)
Reg usage, instruction choice, peephole opt (compiler)

Organization of an optimizing compiler



Classifications of Optimization techniques

- Peephole optimization
- Local optimizations
- Global Optimizations
    - Inter-procedural
    - Intra-procedural
- Loop optimization

Factors influencing Optimization

- The target machine: machine dependent factors can be parameterized to compiler for fine tuning
- Architecture of Target CPU:
    - Number of CPU registers
    - RISC vs CISC
    - Pipeline Architecture
    - Number of functional units
- Machine Architecture
    - Cache Size and type
    - Cache/Memory transfer rate

References and Further readings

➢ *Aho, A. V., Sethi, R. and Ullman, J. D.: Compilers:Principles, Techniques and Tools. 2nd edition*

- *N. Wirth: Compiler Construction. Addison-Wesley 1996,* Available under http://www.oberon.ethz.ch/WirthPubl/CBEAll.pdf
- *H.Bal, D.Grune, C.Jacobs: Modern Compiler Design. John Wiley, 2000*
- *W.M.Waite, G.Goos: Compiler Construction. Springer-Verlag 1984*
- Keth Cooper and Linda Torczon. Engineering a compiler. Elsevier, 2nd edition.

Exercises

(1) Differentiate between the following terms:

i. Activation of procedure and procedure definition
ii. Recursive procedure and non-recursive procedure
iii. Static scope rules and Dynamic scope rule

b. Consider the following program segment:

```
AddTo (y)
{
Int f = 1;
for (2 = 2; 2<=y; 2++)
for = * 2;
return (f);
}
```

i. Represent the program using three-address code
ii. Identify the leader statements within the program.
iii. Identify the blocks within the program and sketch the flow graph.

c. What are the limitations of using static allocation?

(2) a. Describe Loop Optimization

b. Using appropriate example, explain

i. Loop unrolling (ii) Loop jamming

c. Mention two properties of a loop in the context of compiler construction

d. Construct a LR parsing table for the following grammar:

$E \longrightarrow TE_1$
$E_1 \longrightarrow +TE_1/E$
$T \quad FT_1$
$T_1 \longrightarrow *FT_1/E$
$F \longrightarrow id$

(3) a. Write the algorithms for functions Getreg ( )

b. Consider the following expression:

x = (a + b) - ((c + d) - e)

Describe the process of generating the Assembly code for the expression.

c. Consider the following three-address code

t1 = a + b

```
t2    =    c + d
t3    =    e - t2
t4    =    t - t3
```

i. show the DAG representation

d. Identify three main difficulties to efficient code generation.

(4) a. Explain the term intermediate code generation.

b. With the aid of example briefly explain the term Three-Address code.

c. Write the expressions used to represent the three-address statements for the following programming language constructs:

i. Boolean expression
ii. Procedure cale
iii. Arithmetic expressions
iv. Array references and derefrencing operations.

d. Represent G = (a+b)* - c/d
Using Quadruple, Triple and Indirect Triple representations.

(5) a. i. What is bottom-up parsing?
ii. State the concept that makes it to trace out the right-most derivations of an input string w in reverse.

b. With the aid of example, explain the term a handle of a right sentential form.

c. Consider the following grammar, and show the handle of each right sentential form for the string (a, (a,a))

(6) a. State and define the two types of conflicts that an LR parser may encounter.

b. Why does LR parsing is attractive.

c. Construct an LALR(1) parsing table for the following grammar

$S \longrightarrow Aa/bAc/dc/bda$

$A \longrightarrow d$

d. Generate the three-address code for the following C program:

```
main ( )
int i = 1;
int b(20);
```

```
White (I < = 20)
A (i) = ;
```