The Curse of

Compiler Construction

The Curse of

Compiler Construction

The Curse of Compiler Construction

The author will not like to take (dis)credit for writing this book.
The authenticity and the originality of the contents of this book are debatable.
This book is written in English, a type of, with minimum use of Greek.
Read about regular expressions, deterministic finite automata, nondeterministic finite automata and context-free grammars before reading this book.
This book is updated regularly, make sure that you are reading the latest version.
Dragons are purely imaginary creatures.

• This book has been printed on environment-friendly paper •

Contents

UNIT 1
INTRODUCTION TO COMPILER CONSTRUCTION

Language Processors

A compiler is a program that reads a program written in a high-level language, called the source language, and translates it into an equivalent program in a low-level language, called the target language.

Programming languages are notations for describing computations to people and machines.
Software running on all computers is written in some programming language.
Before a program can run, it needs to be converted to a suitable form.
A compiler does this conversion.
Compilers detect errors in the source program, if any.

Source program → [Compiler] → Target program

If the target program is an executable machine language program, then it can be called by the user directly.
This is called running of the program.

Input → [Target program] → Output

An interpreter is another common type of language processor.
An interpreter directly executes the source program on the input.

Source program + Input → [Interpreter] → Output

Target programs produced by compilers are faster than interpreters.
Interpreters are useful in debugging.

Compiler or interpreter?

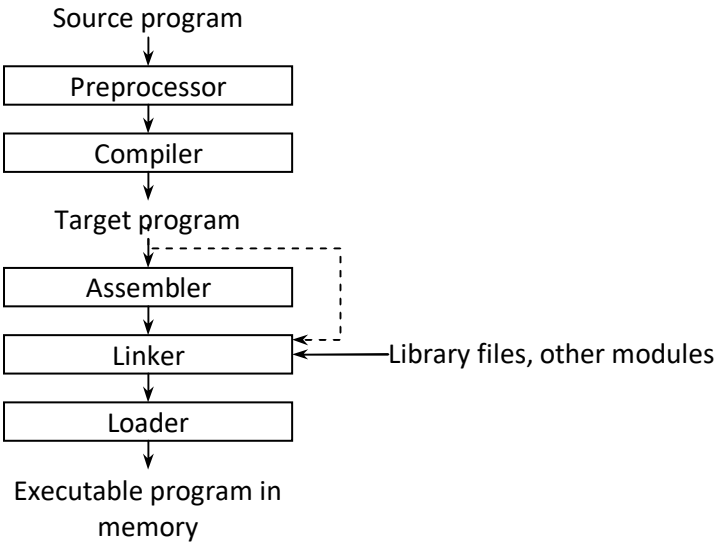
- C, C++
- HTML
- Java
- Visual Basic

A source program may be divided into modules stored in separate files.
A preprocessor collects the modules.
A preprocessor also handles macros and preprocessor directives.

A compiler may produce assembly language programs as output.
Assembly languages are ease to produce and debug compared to machine languages.
An assembler converts assembly language programs into executable machine language programs.

Modules of the source program may be compiled separately.
A linker handles external memory addresses, *i.e.* a situation where the code in one file refers to a variable or a function in another file.
A linker also handles library functions.

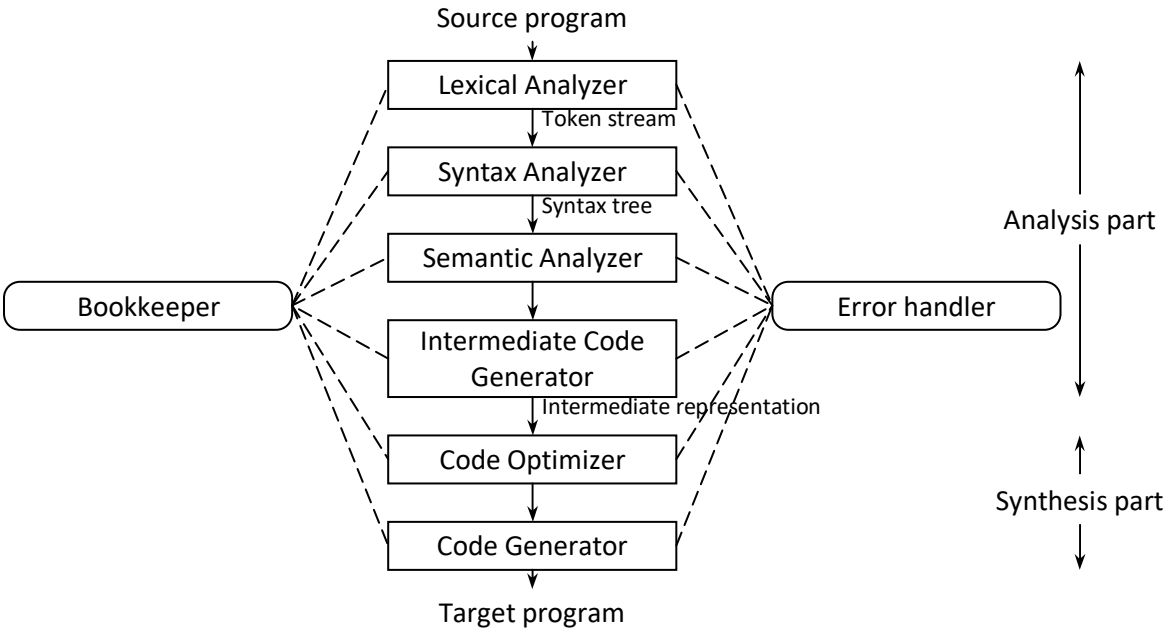
A loader puts all the executable code into the main memory for execution.
A loader is often a part of the operating system.



The Structure of a Compiler

A compiler operates as a sequence of phases.
Each phase transforms one representation of the program into another.
Logically similar tasks are grouped in a particular phase.

Block diagram of a compiler –



The analysis part or front-end breaks up the source program into constituent pieces.
The synthesis part or back-end constructs the target program.

The first phase of a compiler is called lexical analysis or scanning.
The lexical analyzer reads the source program character-by-character and groups them into meaningful sequences called lexemes.
For each lexeme, the lexical analyzer produces as output a token of the form <token-name, attribute-value>.

The tokens are passed on to the next phase, the syntax analyzer.
The syntax analyzer uses the token-name.
The attribute-value is used by subsequent phases.
Older representation: <type, value>.
Blanks separating lexemes are discarded.

The second phase of the compiler is syntax analysis or parsing.
The syntax analyzer, or parser, uses the token-name component of the tokens to create an internal representation of the program called the syntax tree.
A node in the syntax tree represents an operation and its children nodes represent the arguments of the operation.

The semantic analyzer checks that the source program semantically consistent with the language definition.
For example, break and continue statements should be used inside the body of a loop or a switch statement only.
It also gathers type information for use by the subsequent phases.

The intermediate code generator produces a low-level machine language-like intermediate code for the program.
The intermediate code should be easy to produce and should be easy to translate into the target language.
It facilitates code optimization and retargeting.

The code optimizer attempts to improve the intermediate code so that the compiler can produce faster, shorter and lesser energy consuming target programs.
Code optimization is optional.
Optimizing compilers spend lot of time performing optimizations.
Simple optimizations that produce good results are also available.
Code optimization is a misnomer!
Code improvement is a more suitable term.
Finding the optimal code is an NP-complete problem.

The code generator converts the intermediate code into the target language.
If the target language is a machine language, then registers or memory locations are selected for each variable.
Registers should be used judiciously.

The bookkeeper manages all the identifiers used in the program like variables and function names.
The bookkeeper uses a data structure called the symbol table.
It has one record for each name with fields for the necessary attributes.
It should be quick to search, store and retrieve.
It is filled in by the early phases and used by the later phases.
Attributes of a variable: type, scope, etc.
Attributes of a function name: number of arguments, type and method of passing of each argument, type of return value, etc.
Implemented as arrays, linked lists, hash tables, etc.

If a phase finds an unexpected situation, then it invokes the error handler.
The error handler displays an error message and tries to recover from the error.
The compiler should try to process the whole program and detect all errors.

Types of errors –
- Lexical error
- Syntax error / parsing error
- Semantic error

For example, a = b + c / 2

Phases represent a logical organization of the compiler.
While implementing a compiler, several phases may be grouped together into a pass.
A pass reads the representation of the program once, typically from a file.
Typically, number of phases ≥ number of passes.
Single-pass compiler and multi-pass compiler.
For example: Pass I: front-end, Pass II: code optimization, Pass III: code generation.

Common internal representation
SL1, SL2, … SLn → IR → TL1, TL2, … TLm
n front-ends + m back-ends instead of nxm compilers
For example: ACK, GCC.

Compiler construction tools
Specialized tools to construct specific parts of compilers, often using sophisticated algorithms.
- Scanner generator
- Parser generator
- Syntax-directed translation engine
- Data flow analysis engines
- Code generator generator
- Compiler construction toolkits
- Compiler compiler

The Evolution of Programming Languages

1940s -	First modern computers supporting machine languages
Early 1950s -	Assembly languages, later supporting parameterized macros
Late 1950s -	Fortran, Cobol, Lisp
Today -	Large number of programming languages

The first compiler

1952 -	Grace Hopper coined the term compiler. However, the A-0 compiler was more of a linker-loader.
1957 -	J. W. Backus and his team developed the first compiler, for Fortran (18 person-year effort, 2.5 years time).


Generation of programming languages –
1st generation - Machine languages
2nd generation - Assembly languages
3rd generation - High-level languages
(Fortran, Cobol, Lisp, Algol, C, C++, Java)
4th generation - Application-specific languages
(SQL, Postscript, VHDL, ADL)
5th generation - Logic- and constraint-based language
(Prolog, OPS5)

Another classification –
Imperative – how? (C, C++, Java)
Declarative – what? (ML, Haskell, Prolog)

Turing complete language
von Neumann language

Object oriented language (Simula 67, Smalltalk)

Scripting languages
Interpreted
Shorter but slower programs
High-level operators for combining computations
E.g. – JavaScript, Perl, PHP, Python, Ruby, Tcl.

Advances in programming languages + Advances in computer architecture → Advances in compiler construction

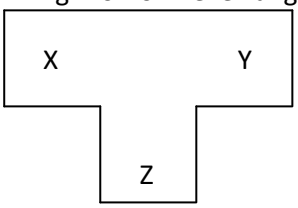
Applications of Compiler Technology

- Implementation of high-level programming languages
- Optimizations for computer architectures
    - Parallelism (parallelizing compiler, concurrency preserving compiler, VLIW)
    - Memory hierarchies
- Design of new computer architectures
    - RISC
    - Specialized architectures
- Program translations
    - Binary translation
    - Hardware synthesis
    - Database query interpreters
    - Compiled simulation
- Software productivity tools
    - Type checking
    - Bounds checking
    - Memory management tools

Bootstrapping

A compiler is characterized by three languages, *viz*. source language, target language and language of implementation.
Representation – C_Z^{XY}
X: high-level language, Y: low-level language, Z: high- or low-level language

Bratman's T-diagram / tombstone diagram –



Bootstrapping is the process of writing a compiler in the language which it is supposed to compile.
A compiler obtained by bootstrapping is called a self-hosting compiler.

Step I: $C_{C++}^{PL/NSIT\ ML} \rightarrow \boxed{C_{ML}^{C++\ ML}} \rightarrow C_{ML}^{PL/NSIT\ ML}$ [PL/NSIT compiler]

Step II: $C_{PL/NSIT}^{PL/NSIT\ ML} \rightarrow \boxed{C_{ML}^{PL/NSIT\ ML}} \rightarrow C_{ML}^{PL/NSIT\ ML}$ [Self-hosting PL/NSIT compiler]

Object Oriented Implementation of Compilers

Object oriented programming can be used to implement all phases and modules of a compiler.
Lexical analysis
- Tokens can be represented as objects.
- Alternatively, terminals may be modeled as objects representing the leaf nodes in a parse tree made up of objects.

Syntax analysis
- The nodes of the parse tree can be represented as objects of classes abstracting various programming constructs.
- Inheritance and delegation can be used to define related classes.
- Specialized parsing algorithms may be used to parse different programming constructs.

Semantic analysis
- Semantic analysis can be performed by visiting the objects in the parse tree and verifying their semantic correctness.
- Alternatively, semantic analysis can be done by sending a message to an object asking it to check its own semantic correctness and those of the nodes descending from it.

Intermediate code generation
- The instructions in the intermediate representation can be modeled as objects.

Code optimization
- Objects in a parse tree can also perform local optimizations on themselves using specialized optimizing techniques.

Code generation
- Final code can be generated by visiting the objects in the parse tree in the program execution order.

Bookkeeping
- Identifiers can be modeled as objects.

Error handling
- Errors can be modeled as objects having functions to recover from the errors they represent.
- Alternatively, error handling may be performed using exceptions.

Object oriented programming improves modularity, maintainability, extensibility, portability and performance of compilers while decreasing source code size and development time.
Object oriented programming can be used to implement compiler compilers too.

Miscellaneous

- Object code: relocatable machine code that is usually not directly executable
- Cousin of compiler
- Optimizing compiler
- Cross compiler
- Half compiler
- Incremental compiler
- Just-in-time (JIT) compiler / dynamic compiler
- Silicon compiler

[Sections: 1.1-1.3, 1.5]

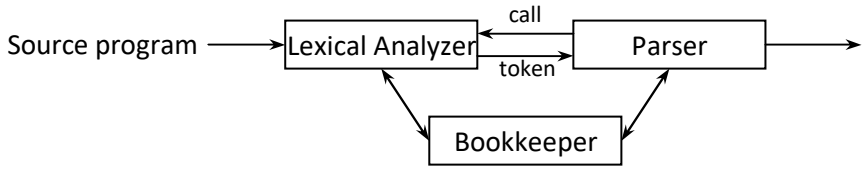
UNIT 2
LEXICAL ANALYSIS

The Role of the Lexical Analyzer

An atomic meaningful sequence of characters in the source program is called a lexeme.

The lexical analyzer reads the characters of the source program, groups them into lexemes and produces as output a sequence of tokens.
This sequence of tokens is send to the parser for syntax analysis.
The lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser.
If the lexical analyzer discovers a lexeme that is an identifier, then that identifier is entered into the symbol table if it is not there already.

Whenever the parser needs a token, it calls the lexical analyzer.
The call causes the lexical analyzer to read the characters from the source program until it can identify the next lexeme and produce for it the next token, which it returns to the parser.



Other tasks performed by the lexical analyzer –
- Stripping of comments and whitespace
- Expansion of macro
- Correlating error messages produced by the compiler with the source program
    - Keeping track of the source file name
    - Keeping track of the number of newline characters read so that a line number can be associated with each error message

A lexical analyzer may be divided into a cascade of two tasks –
- Scanning or preprocessing consisting of deletion of comments, compaction of consecutive whitespace characters and expansion of macros.
- Tokenization of the source program.

Why lexical and syntax analyses are implemented as separate phases?
- Simplicity of design
- Efficiency: specialized algorithms can be used
- Portability: input device specific issues can be limited to lexical analyzer

Types of tokens: keywords, identifiers, constants, operators and punctuators.

Lexical error
For example, too long identifier, too long numeric constant, illegal constant, undefined operator.
Simplest recovery mechanism is called panic mode.
Delete successive characters in the source program till a valid lexeme can be identified.
This may confuse the parser.

Input Buffering

The lexical analyzer often has to look ahead a few characters beyond the current character to be sure that it is identifying the correct lexeme.

Two pointers to the source program are maintained –

- begin: marks the beginning of the next lexeme whose extent we are trying to determine
- forward: looks ahead until a pattern match is found
- E.g. – E = m * c ** 2

```
      ^   ^
      |   |
      |   forward
      begin
```

When a lexeme is found –

- forward is set to its rightmost character
- a token is produced
- begin is set to the character just right of the lexeme just found

Sentinel is a special character used to represent the end of the buffer or source program.
It should not be a part of the source language.
The eof character is a suitable choice.

Read: RE, NFA and DFA.

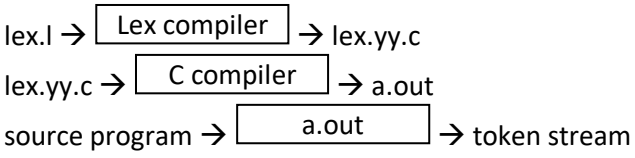
Regular expressions are used to specify lexemes.
NFAs are used to identify lexemes.
Transition diagrams of NFAs can be drawn (starred states, tokens returned).

E.g. – relational operators
keywords (if, int, …)
identifiers (letter(letter|digit|_)*) (maximum size)

E.g. – Token stream for a program.

The Lexical Analyzer Generator Lex

Lex produces a lexical analyzer which identifies lexemes which are specified by regular expressions.
The input notation is called the Lex language.
The tool itself is called the Lex compiler.
Developed by M. E. Lesk in 1975.
Variants: flex, jflex, jlex, etc.



This lexical analyzer –

- is often used as a subroutine of the parser
- returns the token-name as an integer
- saves the attribute value in the global variable yylval

Structure of Lex programs
declarations
%%
translation rules
%%
auxiliary functions

Declarations: variables, manifest constants (identifiers to stand for constants)
Translation rules: pattern {action}
- Pattern: regular expression based on declarations
- Action: C code fragment

Auxiliary functions: additional C functions

Conflict resolution –
- prefers longer pattern over shorter (<= and <)
- for same size of pattern, use order in the lex program (keyword and identifier)

[Sections: 3.1, 3.2;
Read: 3.3, 3.4, 3.6, 3.7;
Lab: 3.5]

UNIT 3
SYNTAX ANALYSIS

The Role of the Parser

The syntax of a programming language describes the proper form of its programs.
The semantics of the language defines what its programs means, *i.e.* what they do when executed.

For specifying syntax, we use context-free grammar (CFG), also called type 2 grammar.
Read: context-free grammar.
For specifying semantics, we use informal descriptions and suggestive examples.

CFG also helps to guide the translation process.
Syntax-directed translation is a grammar-oriented translation process.

Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived then report syntax error.

For a syntactically correct program, the parser constructs a parse tree and passes it to the subsequent phases for further processing.
In reality, however, the parse tree need not be constructed explicitly because the next phases may be interleaved with parsing.

The precision of the parsing methods allows syntax errors to be detected early.
Most parsers detect an error when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar of the language.
These parsers have viable-prefix property, *i.e.* they can detect an error as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Error handling strategies –
Quit after detecting the first error.

Panic mode recovery:	Discard tokens one at a time until a predefined synchronizing token is found. Synchronizing tokens are usually delimiters, like ; and }, whose role in the source language is clear and unambiguous.
Phrase-level recovery:	On detecting an error, the parser performs local correction on the remaining input so that parsing can be continued. For example – replace a comma by a semicolon, delete a semicolon, insert a semicolon, etc. This may lead to an infinite loop.
Error productions:	The grammar is augmented using error productions for expected errors. When an error production is encountered, necessary recovery action is undertaken.

Context-Free Grammars

A parse tree pictorially shows how the start symbol of the grammar derives a string in the language.

E.g. –
Grammar: $E \rightarrow E + T \mid T$

T → T * F | F
F → (E) | id
String: (a+b)*c

From left to right, the leaves of the parse tree forms the yield of the tree which is the string generated by or derived from the nonterminal at the root of the parse tree.
The language generated by a grammar is the set of all strings that can be generated by some parse tree.

It can be proved theoretically, that for any context-free grammar there exists a parsing algorithm that takes $O(n^3)$ time to parse a string of n terminals.
However, linear time algorithms suffice to parse essentially all languages used typically.
A parser should be able to construct the parse tree by reading the source program only once.

There are three general types of parsers: universal, top-down, bottom-up.

Universal parsing methods like Cocke-Younger-Kasam algorithm and Earley's algorithm can parse any grammar.
However they are too inefficient to use in a compiler.

A top-down parser builds a parse tree from the top (root) to the bottom (leaves).
A bottom-up parser starts from the leaves and work its way up to the root.

Programs are always parsed left-to-right.

LL and LR parsers can parse a large subset of grammars.
LL parsers are often hand-implemented.
LR parsers are usually constructed using automated tools.

In a leftmost derivation, the leftmost nonterminal in each sentential form is replaced.
We write, $\alpha \underset{lm}{\Rightarrow} \beta$.
In a rightmost derivation, the rightmost nonterminal in each sentential form is replaced.
We write, $\alpha \underset{rm}{\Rightarrow} \beta$.

An ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for at least one string.
Most parsers work for unambiguous grammars only.

In an abstract syntax tree for an expression, each interior node represents an operator and the children of the node represent the operands of the operator in the correct order.
Abstract syntax tree is also called syntax tree.
Parse tree is also called concrete syntax tree.

E.g. – parse tree, yield, leftmost derivation, rightmost derivation, syntax tree.
E.g. –
Grammar: E → E + E | E * E | id
String: a+b*c

Writing a Grammar

Sometimes an ambiguous grammar can be rewritten to eliminate ambiguity.

stmt → if expr then stmt
| if expr then stmt else stmt

stmt → matched_stmt
| open_stmt
matched_stmt → if expr then matched stmt else matchted stmt
open_statement → if expr then matched_stmt else stmt
| if expr then matched_stmt else open_stmt

A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \underset{lm}{\Rightarrow} A\alpha$ for some string α.
Top-down parsers cannot handle left recursion.

Immediate left recursion
$A \rightarrow A\alpha \mid \beta$
Rewrite as
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$

General left recursion (example)
A → Ba | c
B → Ab | d

Left factoring
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
Rewrite as
$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2$

Some syntactic constructs cannot be specified by grammar alone.
These constructs are handled by the semantic analyzer.

- Checking that a variable has been defined before it is used.
- Checking that the number of actual parameters in a function call matches with the number of formal parameters in the function definition.
- Checking that break and continue statements have been used within loops or switch statements.

Top-Down Parsing

Top-down parsing can be viewed as finding a leftmost derivation for an input string.

At each step, the key problem is that of determining the production to be applied for a nonterminal, say A.
Once an A-production is chosen, the rest of the parsing process consists of matching the terminals in the production body with the input string.

A recursive-descent parser consists of a set of mutually recursive procedures, one for each nonterminal.
Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.
General recursive-descent may require backtracking, *i.e.* it may require repeated scans over the input.

```
void A()
        {Choose an A-production, say A → X1X2…Xk
         for (i = 1 to k)
                {if (Xi is a nonterminal)
                        call Xi();
                 else if (Xi equals the current input symbol)
                        advance the input to the next symbol;
                 else
                        error();
                }
        }
```

A left recursive grammar can cause a recursive-descent parser to go into an infinite loop.
When we try to expand a nonterminal A, we may find ourselves again trying to expand A without having consumed any input token.

Transition diagrams are useful for visualizing recursive-descent parsers.
E.g. – E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

A predictive parser is a special case of recursive-descent parser that does not require backtracking.
A predictive parser chooses the correct A-production by looking ahead at the input string a fixed number of tokens.
Typically the look ahead is of one token only.

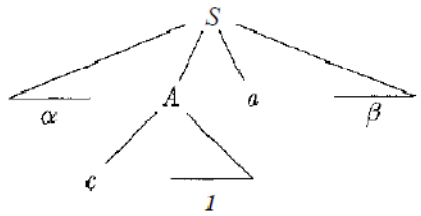
Three steps of predictive parsing –

Step I: Calculate FIRST and FOLLOW sets.

Define FIRST(α) to be the set of terminals that begin strings derived from α.
If $\alpha \Rightarrow \varepsilon$, then ε is also in FIRST(α).

Define FOLLOW(A) to be the set of terminals that can appear immediately to the right of A in some sentential form; *i.e.* the set of terminals *a* such that there exists a derivation of the form $S \overset{*}{\Rightarrow} \alpha A a \beta$, for some *a* and β.
If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A), where $ denotes the end of token stream.

E.g. –



FIRST(A) = {c}
FOLLOW(A) = {a}

Operative part

To compute FIRST(X) apply the following rules until no more terminals or ε can be added to any FIRST set.
1. If X is a terminal, then FIRST(X) = {X}.
2. If X is a nonterminal and X → $Y_1Y_2…Y_k$ is a production for some k ≥ 1.
2a. If for some i, *a* is in FIRST(Y_i) and ε is in FIRST(Y_1), FIRST(Y_2), ... FIRST(Y_{i-1}), then place *a* in FIRST(X).
2b. If ε is in FIRST(Y_1), FIRST(Y_2), ... FIRST(Y_k), then place ε in FIRST(X).
3. If X → ε is a production, then place ε in FIRST(X).

To compute FOLLOW(A) apply the following rules until no more terminals can be added to any FOLLOW set.
1. Place $ in FOLLOW(S), where S is the start symbol.
2. If A → αBβ is a production, then everything in FIRST(β) except ε is in FOLLOW(B).
3. If A → αB is a production or A → αBβ is a production with FIRST(β) containing ε, then everything in FOLLOW(A) is in FOLLOW(B).

E.g. – FIRST(F) = FIRST(T) = FIRST(E) = {(,id}
FIRST(E') = {+,ε}
FIRST(T') = {*,ε}
FOLLOW(E) = FOLLOW(E') = {),$}
FOLLOW(T) = FOLLOW(T') = {+,),$}
FOLLOW(F) = {+,*,),$}

Step II: Construct parsing table.

Predictive parsers can be constructed for LL(1) grammars.

A grammar is LL(1) if and only if whenever A → α | β are productions the following conditions hold.
1. For no terminal *a* do both α and β derive strings beginning with *a*.
2. At most one of α and β can derive ε.
3. If $\alpha \overset{*}{\Rightarrow} \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW (A). Likewise, if $\beta \overset{*}{\Rightarrow} \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW (A).

Operative part

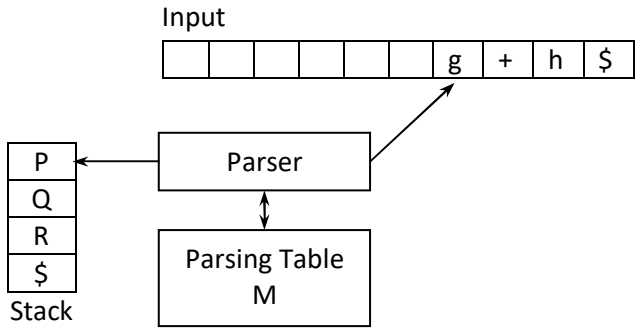
For each production A → α in the grammar do the following.
1. For each terminal *a* in FIRST(α), add A → α to M[A,a].
2. If ε is in FIRST(α), then for each terminal *b* in FOLLOW(A) add A → α to M[A,b].

Vacant cells in the parsing table correspond to syntax errors.
Multiple entries in a cell may be due to ambiguity or left recursion.

E.g. –

Nonterminal	Input symbol					
	id	+	*	(	)	$
E	E → T E’			E → T E’		
E’		E’ → + T E’			E’ → ε	E’ → ε
T	T → F T’			T → F T’		
T’		T’ → ε	T’ → * F T’		T’ → ε	T’ → ε
F	F → id			F → (E)		

Step III: Table driven parsing.



Operative part

```
Let a be the leftmost symbol
Let X be the symbol at the top of the stack
While (a ≠ $ and X ≠ $)
        {If (X = a)
                Pop the stack and advance the input pointer
         Else if (M[X,a] = X → Y1Y2…Yk)
                {Pop X from the stack
                 Push Yk…Y2Y1 on to the stack with Y1 on the top
                }
         Else
                Error()
        }
```

E.g. –

Matched	Stack	Input	Production
	E$	id+id$	E → T E’
	TE’$	id+id$	T → F T’
	FT’E’$	id+id$	F → id
	idT’E’$	id+id$	—
id	T’E’$	+id$	T’ → ε
id	E’$	+id$	E’ → + T E’
id	+TE’$	+id$	—
id+	TE’$	id$	T → F T’
id+	FT’E’$	id$	F → id
id+	idT’E’$	id$	—
id+id	T’E’$	$	T’ → ε
id+id	E’$	$	E’ → ε
id+id	$	$	Accept

Bottom-Up Parsing

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves and working up towards the root.

There is a general style of bottom-up parsing known as shift-reduce parsing.
The grammars for which shift-reduce parsers can be built are called LR grammars.
Although it is too much work to build an LR parser by hand, automatic parser generators make it easy to construct efficient LR parsers from suitable grammars.

Bottom-up parsing is the process of 'reducing' a string of the source language to the start symbol of the grammar.
At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
The key decisions that are taken during bottom-up parsing are about when to reduce and about what production to apply.
A reduction is the reverse of a step in a derivation.

Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.

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.

A handle of a right-sentential form γ is a production A $\rightarrow \beta$ and a position of γ where β may be found such that replacing β at that position by A produces the previous right-sentential form in the rightmost derivation of γ.

$$S \underset{rm}{\overset{*}{\Longrightarrow}} \alpha Aw \underset{rm}{\Longrightarrow} \alpha\beta w \underset{rm}{\overset{*}{\Longrightarrow}} \gamma$$

For convenience, we call β the handle rather than $\alpha \rightarrow \beta$.
If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
A rightmost derivation in reverse can be obtained by 'handle pruning'.

Shift-reduce parsing 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.
The handle always appears at the top of the stack just before it is identified.

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.
It then reduces β to the head of the appropriate production.
The parser repeats this cycle until the stack contains the start symbol and the input is empty.

A shift-reduce parser can take four possible actions – shift, reduce, accept and error.

There are context-free grammars for which shift-reduce parsing cannot be used.

- Shift/reduce conflict
- Reduce/reduce conflict

E.g. –

Stack	Input	Action
$	id+id$	Shift
$id	+id$	Reduce by F → id
$F	+id$	Reduce by T → F
$T	+id$	Reduce by E → T
$E	+id$	Shift
$E+	id$	Shift
$E+id	$	Reduce by F → id
$E+F	$	Reduce by T → F
$E+T	$	Reduce by E → E + T
$E	$	Accept

Simple LR Parsers

LR(k) parser
k = number of input symbols of look ahead that are used in making parsing decisions
Typically, k = 0 or 1
If k is not mentioned, then k is assumed to be 1

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse.
States represent sets of 'items'.
An LR(0) item is a production with a dot at some position of the body.

E.g. –
Production: A → XYZ
Items: A → .XYZ
A → X.YZ
A → XY.Z
A → XYZ.
Production: A → ε
Item: A → .

One collection of sets of LR(0) items, called the canonical LR(0) collection, provides the basis for constructing a DFA that is used to make parsing decisions.
Such an automaton is called an LR(0) automaton.
Each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.

For a grammar G with start symbol S, we define an augmented grammar G' with start symbol S' and an extra production S' → S.
Acceptance occurs when the parser is about reduce by S' → S.

Three steps of SLR parsing –

Step I: Construct LR(0) automaton.

If I is a set of items, then CLOSURE(I) is the set of items constructed as follows.
1. Add every item in I to CLOSURE(I).

2. If A → α.Bβ is in CLOSURE(I) and B → γ is a production, then add the item B → .γ to CLOSURE(I) if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

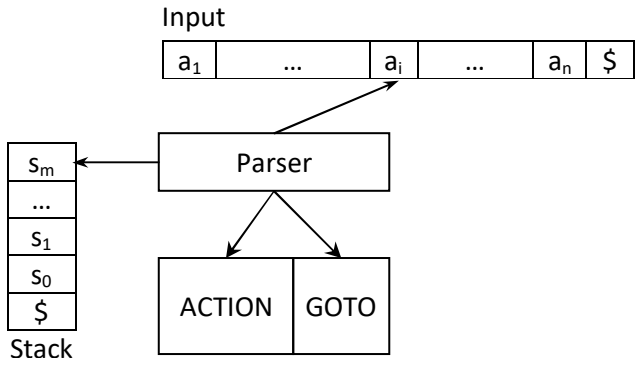
GOTO (I,X) is defined to be the closure of the set of all items [A → αX.β] such that [A → α.Xβ] is in I.

Canonical collection of sets of LR(0) items for G'
C = CLOSURE({[S' → .S]})
Repeat
    For (each set of items I in C)
        For (each grammar symbol X)
            If (GOTO(I,X) is not empty and not in C)
                Add GOTO(I,X) to C;
Until no more sets of items can be added to C

Step II: Construct SLR parsing table.

1. C = {I_0, I_1, … I_n}.
2. State i is constructed from I_i. The parsing actions for state i are determined as follows.
(a) If [A → α.aβ] is in I_i and GOTO (I_i,a) = I_j, then set ACTION[i,a] = "shift j".
(b) If [A → α.] is in I_i, then set ACTION[i,a] = "reduce A → α" for all *a* in FOLLOW(A). Here, A must not be S'.
(c) If [S' → S.] is in I_i, then set ACTION[i,$] = "accept".
3. If GOTO(I_i,A) = I_j, then GOTO[i,A] = j.
4. All blank entries are made "error".
5. The initial state of the parser is the one constructed from the set of items containing [S' → .S].

Step III: Table driven parsing.



Current configuration = ($s_0s_1…s_m$, $a_ia_{i+1}…a_n$).

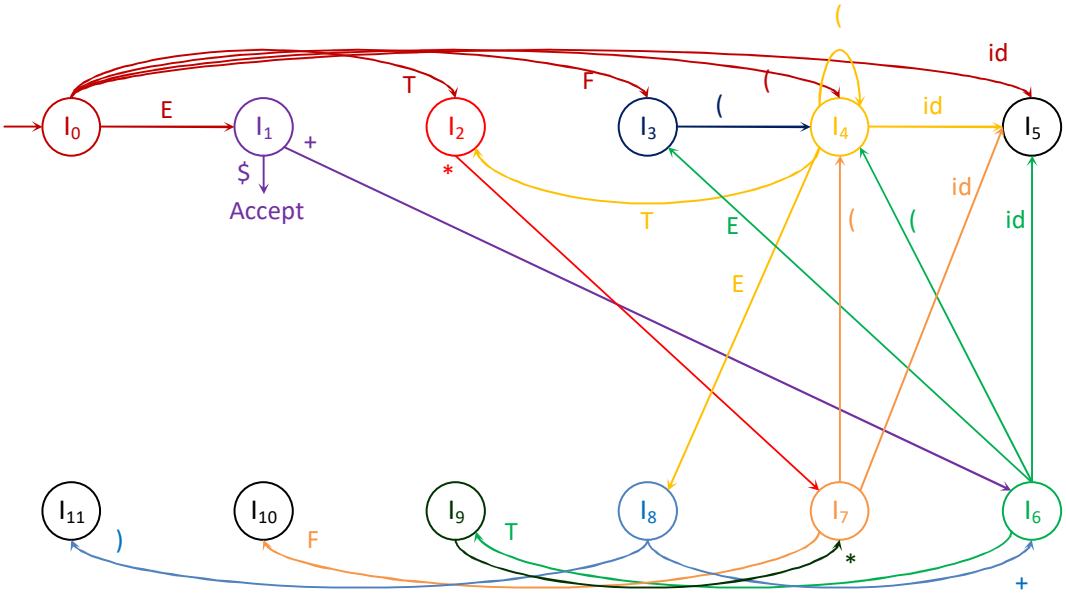
1. If ACTION[s_m,a_i] = shift s, then next configuration = ($s_0s_1…s_ms$, $a_{i+1}…a_n$).
2. If ACTION[s_m,a_i] = reduce A → β, then next configuration = ($s_0s_1…s_{m-r}s$, $a_ia_{i+1}…a_n$) where r is the length of β and s = GOTO[s_{m-r},A].
3. If ACTION[s_m,a_i] = accept, parsing complete.
4. If ACTION[s_m,a_i] = error, syntax error.

E.g. –
0. E' → E
1-2. E → E + T | T
3-4. T → T * F | F
5-6. F → (E) | id

I_0 = CLOSURE({[E' → .E]}) = {[E' → .E], [E → .E+T], [E → .T], [T → .T*F], [T → .F], [F → .(E)], [F → .id]}
I_1 = GOTO(I_0,E) = CLOSURE({[E' → E.], [E → E.+T]}) = {[E' → E.], [E → E.+T]}
Since [E' → E.] is in I_1, ACTION[I_1,$] = Accept
I_2 = GOTO(I_0,T) = CLOSURE({[E → T.], [T → T.*F]}) = {[E → T.], [T → T.*F]}
I_3 = GOTO(I_0,F) = CLOSURE({[T → F.]}) = {[T → F.]}
I_4 = GOTO(I_0,() = CLOSURE({[F → (.E)]}) = {[F → (.E)] , [E → .E+T], [E → .T], [T → .T*F], [T → .F], [F → .(E)], [F → .id]}
I_5 = GOTO(I_0,id) = CLOSURE({[F → id.]}) = {[F → id.]}
I_6 = GOTO(I_1,+)
I_7 = GOTO(I_2,*)
I_8 = GOTO(I_4,E)
I_9 = GOTO(I_6,T)
I_{10} = GOTO(I_7,F)
I_{11} = GOTO(I_8,))

Kernel items: The initial items and all items whose dot is not at the left end.
Non-kernel items.

LR(0) automaton



If transition on a terminal is available: Shift the target state.
Otherwise: Reduce using an item. Pop off states. Move from new top state using the transition for the nonterminal at the head of production.

si = shift state i.
ri = reduce by production numbered i.

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Stack	Input	Action
0	id+id$	s5
0 5	+id$	r6
0 3	+id$	r4
0 2	+id$	r2
0 1	+id$	s6
0 1 6	id$	s5
0 1 6 5	$	r6
0 1 6 3	$	r4
0 1 6 9	$	r1
0 1	$	acc

More Powerful LR Parsers

LR(1) items
General form: [A → $\alpha.\beta$,a]
An item [A → $\alpha.$,a] calls for reduction by A → α only if the next input symbol is a.

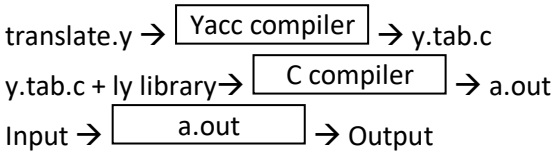
The canonical LR or CLR method makes full use of the look ahead symbol.
This method uses a large set of LR(1) items.

The look ahead LR or LALR method is based on the LR(0) sets of items and has many fewer states CLR parsers.
By carefully introducing look ahead into the LR(0) items, we can handle many more grammars with the LALR method than with the SLR method, and build parsing tables that are no bigger than the SLR tables.
LALR method combines the states that have the same kernel.
LALR is more commonly used.

Parsing method is same for SLR, CLR and LALR parsers.

Parser Generators

The Yet Another Compiler-Compiler (YACC) was developed by S. C. Johnson.
Produces LALR parsers.



Structure of Yacc programs
declarations
%%
translation rules
%%
supporting C routines

Example of translation rule –
```
expr : expr '+' term {$$ = $1 + $3;  /* semantic action */}
     I term {$$ = $1;}
     ;
```

[Sections: 4.1-4.7;
Lab: 4.9]

UNIT 4
INTERMEDIATE CODE GENERATION

The Role of the Intermediate Code Generator

In the analysis-synthesis model of a compiler, the front-end analyzes a source program and creates an intermediate representation from which the back-end generates the target program.

Ideally, the details of the source language are confined to the front-end and the details of the target machine to the back-end.

Syntax-directed translation is a grammar-oriented compilation technique.
It is often used to produce intermediate code.

Syntax-Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar.

An attribute is any quantity associated with a programming construct.
Examples of attributes are data types of expressions, the number of instructions in the generated code, the location of the first instruction in the generated code for a construct, etc.

A translation scheme is a notation for attaching program fragments to the productions of a grammar.
The program fragments are executed when the production is used during syntax analysis.
The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program.

A syntax-directed definition associates
1. with each grammar symbol, a set of attributes, and
2. with each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production.

An attribute is said to be synthesized if its value at a parse tree node N is determined from attribute values at the children of N and at N itself.
Synthesized attributes have the desirable property that they can be evaluated during a single bottom-up traversal of a parse tree.

Inherited attributes have their value at a parse-tree node determined from attribute values at the node itself, its parent, and its siblings in the parse tree.

E.g. – Let t be a string-valued attribute that represents the postfix notation for the expression generated by that nonterminal in a parse tree.

PRODUCTION	SEMANTIC RULES
$E \rightarrow E_1 + T$	$E.t = E_1.t \mid\mid T.t \mid\mid$ '+'
$E \rightarrow E_1$	$E.t = E_1.t$
$T \rightarrow 0$	T.t = '0'
$T \rightarrow 1$	T.t = '1'
…	…
$T \rightarrow 9$	T.t = '9'

A syntax-directed translation scheme is a notation for specifying a translation by attaching program fragments to productions in a grammar.
A translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly specified.
Program fragments embedded within production bodies are called semantic actions.
The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body.
E.g. – $E' \rightarrow + T \{print ('+')\} E'$

Three-Address Code

An instruction in the three-address format consists of one operator and up to three addresses.
Result = Operand1 Operator Operand2
An address can be a variable, a constant or a compiler generated temporary.

E.g. – x + y * z
Three-address code:
t1 = y * z
t2 = x + t1

The unraveling of multiple-operator expressions and nested flow of control make it easy for code optimization and code generation.

Common three-address instruction formats:
1. Binary operations: x = y op z
2. Unary operations: x = op y
3. Copy operations: x = y
4. Unconditional jumps: goto L
5. Conditional jumps: if x goto L
   if not x goto L
   if x relop y goto L
6. Function calls and returns: param x1
   param x2
   …
   param xn
   y = call p, n
   return y
7. Indexed copy operations: x = y [i]
   x [i] = y
8. Address and pointer assignments: x = &y

```
x = *y
*x = y
```

E.g. – do {i = i + 1;} while (a [i] < v);
Three-address code:

```
L:  t1 = i + 1
    i = t1
    t2 = i * 2      (assume sizeof(a[i]) = 2)
    t3 = a [t2]
    if t3 < v goto L
```

```
t4 = t3 < v
if t4 goto L
```

Three-address instruction may be implemented in three ways, *viz*. quadruples, triples and indirect triples.

Quadruples have four fields, *viz*. op, arg1, arg2 and result.
Temporaries may be created and entered in the symbol table.
This may clutter the symbol tables.
Easy to rearrange instructions and thus helps in code optimization.

E.g. – a = b * - c + b * - c
Quadruples:

```
t1 = - c
t2 = b  * t1
t3 = - c
t4 = b  * t3
t5 = t2 + t4
a = t5
```

Triples use three fields, *viz*. op, arg1 and arg2.
The result is referred to as the position of the instruction.
No temporary is needed.
Difficult to rearrange instructions.

E.g. –
Triples:

```
0: - c
1: * b (0)
2: - c
3: * b (2)
4: + (1) (3)
5: = a (4)
```

Indirect triples lists pointers to triples.
Code optimization may be performed by rearranging the pointers without modifying the triples.

E.g. –

```
(0)
(1)
(2)
(3)
```

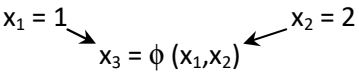
(4)
(5)

Static single assignment (SSA) form is an intermediate representation that facilitates code optimization.

Two distinctive aspects distinguish SSA from three-address code.
1. All assignments in SSA are to variables with distinct names; hence the term static single assignment.
2. SSA uses a notational convention called the ϕ – function to combine multiple definitions of the same variable.

E.g. –
$p_1 = a + b$
$q_1 = p_1 – c$
$p_2 = q_1 * d$
$q_2 = p_2 + q_1$

E.g. – if (flag) x = 1; else x = 2;



Types and Declarations

A declaration specifies the type of a name.
The type and the address of a name are put into the symbol table due to a declaration, so that the compiler can subsequently get them when the name appears in an expression.

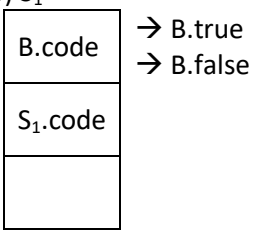
Translation of Expressions

X.code: attribute denoting three-address code for X
X.addr: attribute denoting address that holds the value of X
gen(): creates a new instruction
getaddr(): returns the address of a variable
newtemp(): creates a new temporary

$S \rightarrow id = E$;	S. code = E.code || gen (getaddr(id) '=' E.addr)
$E \rightarrow E_1 + E_2$	E.addr = newtemp()
	E.code = E_1.code || E_2.code || gen (E.addr '=' E_1.addr '+' E_2.addr)
$E \rightarrow (E_1)$	E.addr = E_1.addr
	E.code = E_1.code
$E \rightarrow id$	E.addr = getaddr (id)
	E.code = ''

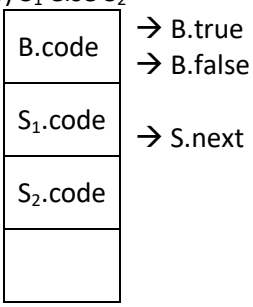
Control Flow

$S \rightarrow$ if (B) S_1



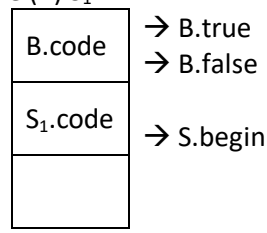
S.code = B.code || label (B.true) || S_1.code || label (B.false)

$S \rightarrow$ if (B) S_1 else S_2



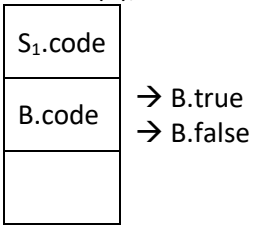
S.code = B.code || label (B.true) || S_1.code || label (B.false) || S_2.code || label (S.next)

$S \rightarrow$ while (B) S_1



S.code = label (S.begin) || B.code || label (B.true) || S_1.code|| label (B.false)

$S \rightarrow$ do S_1 while (B);



S.code = label (B.true) || S_1.code || B.code|| label (B.false)

Backpatching

An important issues when generating intermediate code is that of providing target addresses for forward jump instructions.
E.g. – if (B) S

In the backpatching technique, lists of jump instructions are passed as synthesized attributes.
When a jump instruction is generated, the target of the jump is temporarily left unspecified.
Such a jump instruction is put on a list that contains all jump instructions targeting that given label.

When the address of the label is determined, the target fields of all the jump instructions on the list are filled in.
This allows one-pass intermediate code generation.
Backpatch (p,i): inserts *i* as the target address for each of the instructions on the list pointed to by *p*.
Sometimes a marker nonterminal M is used in a production to cause a semantic rule to pickup the address of the next instruction to be generated.
E.g. – S → if (B) M S_1 {backpatch (B.true,M.addr);}
E.g. – S → if (B) M_1 S_1 else M_2 S_2 {backpatch (B.true,M_1.addr); backpatch (B.false,M_2.addr);}

Goto statements can be implemented by maintaining a list of unfilled jump instructions for each label and then backpatching the target when it is known.

Break statements can be implemented by keeping track of the enclosing loops, generating an unfilled jump instruction for the break statement and then backpatching.

Switch Statements

switch (E)
{case V_1: S_1
 case V_2: S_2
 …
 case V_{n-1}: S_{n-1}
 default: S_n
}

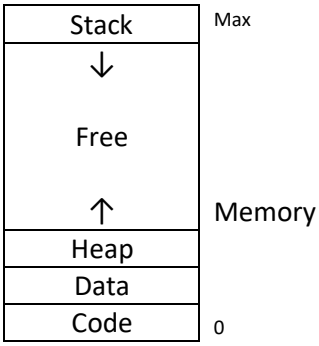
```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        …
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        …
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

[Sections: 2.3, 6.2-6.4, 6.6-6.8]

UNIT 5
RUNTIME ENVIRONMENT

Storage Organization

A compiler assumes a runtime environment in which the target programs will be executed.
The runtime environment deals with a variety of issues such as the layout and allocation of storage locations for the data items named in the source program, the mechanisms used to access these data items, the linkages between procedures, the mechanisms for passing parameters, etc.



Global variables are allocated static storage, *i.e.* in the data segment.
The locations of these variables are known at the compile time and remain fixed at the runtime.

Local variables have their lifetimes contained within one activation of a function.
They can be allocated storage on the stack.

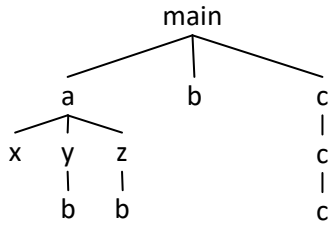
Some languages allow dynamically allocated variables.
C → malloc(), free()
C++ → new, delete
These variables can be allocated storage in the heap.

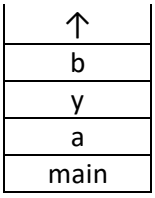
Garbage collection enables the runtime system to detect data items which will not be used further in the program and reuse the memory locations allocated to them even if the programmer has not returned their memory locations to the runtime environment.

Stack Allocation of Space

We can represent the activation of functions during the execution of a program by a tree, called an activation tree.
Each node represents one activation of one function.
The root represents the activation of the main function.
The children nodes represent the functions called by the function, in the given order.
A function can be called only when the function represented by the node just left of it has completed.

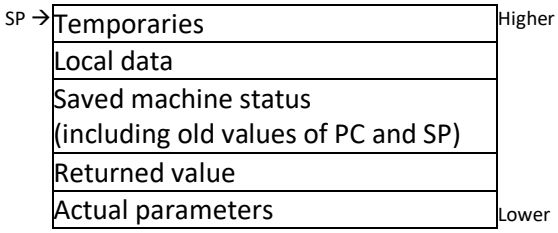
E.g. –





Function calls and returns are handled using the control stack.
Each live activation of a function has an activation record, also called frame.
The activation record of the main function is at the bottom of the control stack.
The sequence of activation records from the bottom to the top of the control stack represents the path in the activation tree from the root to the node representing the currently active function.

An activation record



Procedure calls are implemented by calling sequences which consists of code that allocates an activation record on the stack and enters information into its fields.
A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.
Calling sequences and the layout of activation records may differ greatly, even among compilers for the same language.

[Sections: 7.1, 7.2]

UNIT 6
CODE OPTIMIZATION

Basic Blocks and Flow Graphs

Partitioning three-address instructions into basic blocks –
Find those instructions in the intermediate code that are leaders, *i.e.* the first instructions in some basic block according to the following rules:
1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
For each leader, its basic block consists of itself and all instructions up to, but not including, the next leader or the end of the intermediate program.

The flow of control can only enter the basic block through the first instruction in the block.
Control will leave the block at the last instruction in the block.

Knowing when the value of a variable will be used next is essential for generating good code.
If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.

We assume for convenience that each procedure call starts a new basic block.

The nodes of the flow graph are the basic blocks.
There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B.
There are two ways that such an edge could be justified:
1. There is a conditional or unconditional jump from the end of B to the beginning of C.
2. C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

We say that B is a predecessor of C, and C is a successor of B.
Often we add two nodes, called the entry and exit.
There is an edge from the entry to the first executable node of the flow graph.
There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program.

Optimization of Basic Blocks

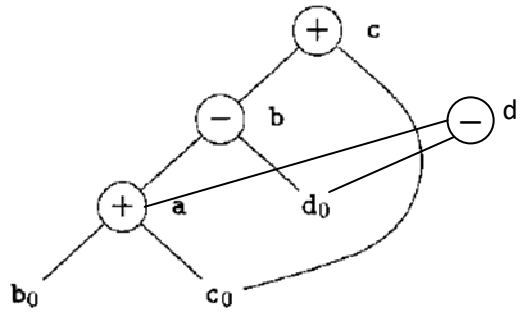
We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself.

Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph).

Constructing a DAG for a basic block –
1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

2. There is a node N associated with each statement S within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to S, of the operands used by S.
3. Node N is labeled by the operator applied at S, and also attached to N is the list of variables for which it is the last definition within the block.
4. Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph.

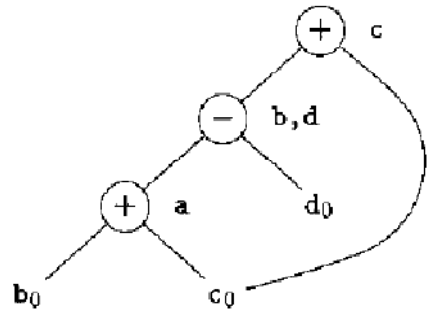
E.g. –
a = b + c
b = a - d
c = b + c
d = a - d



The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.

→ Eliminating local common subexpressions
Common subexpressions can be detected by noticing, as a new node M is about to be added, whether there is an existing node N with the same children, in the same order, and with the same operator.
If so, N computes the same value as M and may be used in its place.

E.g. –



→ Dead code elimination
Delete from a DAG any root (node with no ancestors) that has no live variables attached.

→ Algebraic simplification
Eliminating useless instructions: x = x + 0
Reduction in strength: $x^2 \equiv x * x$
Constant folding: 2 * 3.14 ≡ 6.28

Using commutative and associative laws

After performing these optimizations, the resultant DAG has to be converted back into a sequence of three-address instructions.

The Principal Sources of Optimization

Most global optimizations are based on data-flow analyses, which are algorithms to gather information about a program.

→ Global common subexpressions

→ Copy propagation
Eliminate the copy statement u = v and use v instead of u.

E.g. –
x = t3
a [t2] = t5
a [t4] = x

→ Dead-code elimination
A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
Dead code, or useless code, is 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.

→ Code motion
Decreases the amount of code in a loop.
Take an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and evaluate it before the loop.

E.g. –
while (i <= limit-2) { … }
Rewrite as:
t = limit-2;
while (i <= t) { … }

→ Induction variables and reduction in strength
A variable x is said to be an induction variable if there is a positive or negative constant c such that each time x is assigned, its value increases by c.
Induction variables can be computed with a single addition or subtraction instruction per loop iteration.
The transformation of replacing an expensive operation, such as multiplication, by a cheaper one, such as addition, is known as strength reduction.
It is often also possible to eliminate all but one of a group of induction variables.

E.g. –
B3: j = j - 1
    t4 = 4 * j (t4 is an induction variable)

```
    t5 = a [t4]
    if t5 > v goto B3
Rewrite as:
    t4 = 4 * j
B3: t4 = t4 - 4
    t5 = a [t4]
    if t5 > v goto B3
```

Loop Optimization

1. Induction variable analysis

2. Loop fission: improves locality of reference

```
for(i=0;i<100;i++)                    for(i=0;i<100;i++)
{a[i]=…                       →       a[i]=…
 b[i]=…                               for(i=0;i<100;i++)
}                                     b[i]=…
```

3. Loop fusion or loop combining: minimizes tests and jumps

4. Loop interchange: improves locality of reference

```
for(i=0;i<100;i++)                    for(j=0;j<100;j++)
for(j=0;j<100;j++)            →       for(i=0;i<100;i++)
a[j][i]=…                             a[j][i]=…
```

5. Loop reversal

```
for(i=0;i<100;i++)                    for(i=99;i>=0;i--)
a[99-i]=…                     →       a[i]=…
```

6. Loop unrolling: minimizes tests and jumps but increases code size

```
for(i=0;i<100;i++)                    for(i=0;i<100;i+=2)
a[i]=…                        →       {a[i]=…
                                       a[i+1]=…
                                      }
```

7. Loop splitting

```
for(i=0;i<100;i++)                    for(i=0;i<50;i++)
if(i<50)                              a[i]=…
a[i]=…                        →       for(;i<100;i++)
else                                  b[i]=…
b[i]=…
```

8. Loop peeling: special case of loop splitting

```
for(i=0;i<100;i++)                    a[0]=…
if(i==0)                              for(i=1;i<100;i++)
a[i]=…                        →       b[i]=…
else
b[i]=…
```

9. Unswitching

```
for(i=0;i<100;i++)                    if(x>y)
if(x>y)                               for(i=0;i<100;i++)
a[i]=…                                a[i]=…
                              →       else
else                                  for(i=0;i<100;i++)
b[i]=…                                b[i]=…
```

10. Loop test replacement: increases possibility of dead code elimination

```
i=0;
val=0;
while(i<100)
{val+=5;
 i++;
}
```

→

```
i=0;
val=0;
while(val<500)
{val+=5;
 i++;
}
```

[Sections: 8.4, 8.5, 9.1]

UNIT 7
CODE GENERATION

Issues in the Design of a Code Generator

A code generator has three primary tasks, *viz*. instruction selection, register allocation and assignment, and instruction ordering.
Instruction selection involves choosing appropriate target machine instructions to implement the intermediate code statements.
Register allocation and assignment involves deciding what values to keep in which registers.
Instruction ordering involves deciding in what order to schedule the execution of instructions.

We need to know instruction costs in order to design good code sequences.
However, accurate cost information is often difficult to obtain.

A key problem in code generation is deciding what values to hold in what registers.
Registers are the fastest storage unit on the target machine, but we usually do not have enough of them to hold all values.
Values not held in registers need to reside in the memory.
Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
The use of registers is often subdivided into two subproblems:
1. Register allocation: select the set of variables that will reside in registers at each point in the program.
2. Register assignment: select the specific register that a variable will reside in.
Finding an optimal assignment of registers to variables is difficult, even with single register machines.
Mathematically, the problem is NP-complete.

For each available register, a register descriptor keeps track of the variable name whose current value is in that register.

For each program variable, an address descriptor keeps track of the location or locations where the current value of that variable can be found.
The location might be a register, a memory address, a stack location, or some set of more than one of these.
The information can be stored in the symbol table entry for that variable name.

Peephole Optimization

A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.

The peephole is a small, sliding window on a program.
The code in the peephole need not be contiguous, although some implementations do require this.
It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

In general, repeated passes over the target code are necessary to get the maximum benefit.

→ Eliminating redundant loads and stores
LD a, R0
ST R0, a (can be deleted if unlabeled)

→ Eliminating unreachable code

→ Flow of control optimizations
    goto L1
    …
L1: goto L2
Replaced by:
    goto L2
    …
L1: goto L2 (can be deleted if there is no other jump to L1)

Similarly,
    if a < b goto L1
    …
L1: goto L2

→ Algebraic simplification and reduction in strength
x = x + 0
x = x * 1
$x^2 \equiv x * x$
pow (x, 0.5) ≡ sqrt (x)
x = x + 1 ≡ x ++
x = x - 1 ≡ x --
x = x * -1 ≡ neg (x)
x = x / -1 ≡ neg (x)
x = x * 2 ≡ ashl (x)
x = x / 2 ≡ ashr (x)

→ Use of machine idioms
The target machine may have hardware instructions to implement certain specific operations efficiently.
Detecting situations that permit the use of these instructions can reduce execution time significantly.
For example, some machines have auto-increment and auto-decrement addressing modes.

Register Allocation and Assignment

Local register allocation
Registers to hold values for the duration of a single basic block.
All live variables were stored at the end of each block.

Global register allocation
We might assign registers to frequently used variables and keep these registers consistent across block boundaries, *i.e.* globally.

Eliminates some load and store instructions.

Register allocation by graph coloring
When a register is needed for a computation but all available registers are in use, the contents of one of the registers must be stored, or spilled, into a memory location in order to free up a register.
Graph coloring is a simple, systematic technique for allocating registers and managing register spills.
In the method, two passes are used.
In the first, target machine instructions are selected as though there is an infinite number of symbolic registers; in effect, names used in the intermediate code become names of registers and the three-address instructions become machine language instructions.
In the second pass, for each procedure a register-interference graph is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined.
An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers.
A graph is said to be colored if each node has been assigned a color in such a way that no two adjacent nodes have the same color.
A color represents a register, and the color makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.
Although the problem of determining whether a graph is k-colorable is NP-complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice.
Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge).
Remove n and its edges from G to obtain a graph G'.
A k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.
By repeatedly eliminating nodes having fewer than k edges from the register-interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has k or more adjacent nodes.
In the latter case a k-coloring is no longer possible.
At this point a node is spilled by introducing code to store and reload the register.
A general rule is to avoid introducing spill code into inner loops.

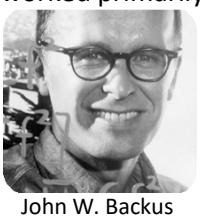
E.g. –
t3 = t1 + t2
t4 = t1 + t3
t5 = t3 + t4
t6 = t3 + t5
(a) 4 registers
(b) 2 registers

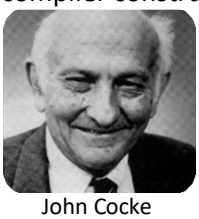
[Sections: 8.1, 8.7, 8.8]

TURING LAUREATES WHO HAVE WORKED ON
PROGRAMMING LANGUAGE DESIGN AND COMPILER CONSTRUCTION

Turing laureates who worked primarily on compiler construction:


Alan J. Perlis
(1966)

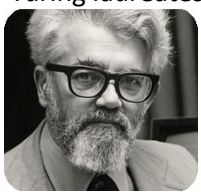

John W. Backus
(1977)

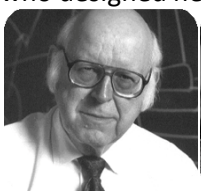

John Cocke
(1987)


Peter Naur
(2005)


Frances E. Allen
(2006)

Turing laureates who designed new programming languages:


John McCarthy
(Lisp, 1971)


Allen Newell and Herbert A. Simon
(IPL, 1975)


John W. Backus
(Fortran, 1977)

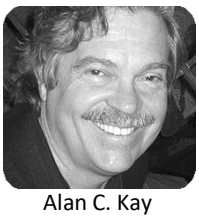

Kenneth E. Iverson
(APL, 1979)


Dennis M. Ritchie
(C, 1983)


Niklaus E. Wirth
(Pascal, 1984)


A. J. R. G. Milner
(ML, 1991)


O.-J. Dahl and Kristen Nygaard
(Simula, 2001)


Alan C. Kay
(Smalltalk, 2003)


Peter Naur
(Algol, 2005)

Turing laureates who also made some contribution in compiler construction:


Edsger W. Dijkstra
(1972)


Donald E. Knuth
(1974)


Robert W. Floyd
(1978)


C. A. R. Hoare
(1980)


Richard E. Stearns
(1993)

Last updated on 17 January 2017.