Translation

- A translator transforms source code (a program written in one language) into object code (the equivalent program in another language, presumably the computer’s native language).
- Such translators include:
  - **Assemblers** – translators where the source language (language of the source code) is a symbolic equivalent of the machine language.
  - **Compilers** – translators where the source language is a higher-level language and the object language is either assembly language or machine language.
  - **Loader** (or **Link Editor**) – assembles one or more object program (together with library routines) into a single program that the computer can run with all its addresses accessible.
  - **Preprocessor** – which perform work in preparation for compiling
Compiled vs. Interpreted Language

Most programming languages are designed with the intent of implementing it through compilation or interpretation.

- **Compiled languages include:**
  - C, C++, FORTRAN, Pascal and Ada.
  - Their runtime routines are limited to supporting a few operations without a close analogue in the machine language (e.g., input/output)

- **Interpreted languages include:**
  - LISP, ML, Perl, Postscript, Prolog and Smalltalk.
  - The translator produces an intermediate form of the program which the executor can interpret quickly.
The Java Virtual Machine

• While Java is closer to C++ than to LISP in form, it is translated into an intermediate representation called bytecodes.
  – These bytecodes are interpreted by the Java Virtual Machine.
  – The time needed to interpret the bytecodes is relatively small compared to the transmission time for Java applets.

The Translation Process

• The translation process may be fairly simple (as in the case of Perl, Prolog or LISP), especially if one is willing to write a software interpreter and accept poor execution speed.
• Translation process is usually divided into 2 parts: analysis of the source program and synthesis of the object program
Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)
Syntax Analysis

• The syntax analysis portion of a language processor nearly always consists of two parts:
  – A low-level part called a *lexical analyzer*
    (mathematically, a finite automaton based on a regular grammar)
  – A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Advantages of Using BNF to Describe Syntax

• Provides a clear and concise syntax description
• The parser can be based directly on the BNF
• Parsers based on BNF are easy to maintain
Reasons to Separate Lexical and Syntax Analysis

• *Simplicity* - less complex approaches can be used for lexical analysis; separating them simplifies the parser
• *Efficiency* - separation allows optimization of the lexical analyzer
• *Portability* - parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis (Scanning)

• Lexical analysis involves the recognition of the elementary constituents of a program.
  – These are the keywords, operators, comments, delimiters, identifiers, literals, etc.
  – The individual characters of the source program must be grouped together to form these constituents.
• The scanner must identify each lexeme and associate with it the grammatical component within the program with which the lexeme is associated. We call the grammatical component the *token*.
• The formal model for lexical analysis is a finite automaton.
• This is sometimes complicated by the difficulty in recognizing where the boundaries between tokens is:
  – e.g., DO 10 I = 1, 5  \( \text{DO 10 I = 1.5} \)
Implementing A Scanner

- We can construct a lexical analyzer (or scanner) by one of three methods:
  - Write a formal description of the regular expressions that we wish to accept and use a software tool to generate a scanner automatically.
  - Write a program that simulates the finite automaton that recognizes the regular expressions that we wish to accept.
  - Construct a table that describes the finite automaton and write a program that uses the data in the table.

Finite-State Automata

- A Finite Automaton is a state machine that changes state as it processes each character is an expression and then:
  - accepts the expression if it is in a final state (the expression belongs to the language) or
  - reject the expression if it is not in a final state (the expression does not belong to the language)
FA To Recognize Optionally Signed Integers

Converting \((0+1)^*01(0+1)^*\) to an FA
```c
#include <ctype.h>
#include <stdio.h>

/* The token as an enumerated type */
typedef enum {PLUS, TIMES, LPAREN, RPAREN, EOL, NUMBER, ERROR} TokenType;

int numval; /* computed numeric value of a NUMBER */
int curr_char; /* Current character */
```
TokenType getToken(void)
{
    while (((curr_char = getchar()) == ' ')
        ; /* Skip white space */
    if (isdigit(curr_char))
    {
        /* recognize a NUMBER token */
        numval = 0;
        while (isdigit(curr_char))
        {
            /* compute numeric value */
            numval = 10 * numval
            + curr_char - '0';
            curr_char = getchar();
        }
        /* put back last character onto input */
        ungetc(curr_char, stdin);
        return(NUMBER);
    }
    else
    {
        /* recognize a special symbol */
        switch(curr_char) {
            case '(': return (LPAREN);
            case ')': return (RPAREN);
            case '+': return (PLUS);
            case '*': return (TIMES);
            case '\n': return (EOL);
            default: return (ERROR);
        }
    }
}
int main(void)
{
    TokenType token;
    do {
        token = getToken();
        switch(token) {
            case PLUS: printf("PLUS\n"); break;
            case TIMES: printf("TIMES\n"); break;
            case LPAREN: printf("LPAREN\n"); break;
            case RPAREN: printf("RPAREN\n"); break;
            case EOL: printf("EOL\n"); break;
            case NUMBER: printf("NUMBER: %d\n", numval);
                break;
            case ERROR: printf("ERROR: %c\n", curr_char);
        }
    } while (token != EOL);
    return(0);
}
Syntactic Analysis (Parsing)

- The grammatical structure of the program is identified (e.g., statements, procedures, expressions, etc.)
- The parser must recognize how lexemes are grouped to form expressions, statements, declarations, etc.
- The actions of semantic analysis are usually initiated by the parser.
- The formal model for the parser is the pushdown automaton.

The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program
Types of Parsers

- Parsers can be either **top-down** or **bottom-up**:
  - Top-down parsers build the parse-tree starting from the root building until all the tokens are associated with a leaf on the parse tree.
  - Bottom-up parsers build the parse-tree starting from the leaves, assembling the tree fragments until the parse tree is complete.

Top-down Parsers

```
Sentence
/   \
Subject phrase  Predicate
     /   \         /   \
   definite article Adjec-tives noun
      /   \      /   \      /   \
   The Adjec-tives The noun
```

Top-down parsing assumes a certain minimum structure as we start building the parse tree.
Bottom-up parsers

Bottom-up parsers **shift** by each token, **reducing** them into a non-terminal as the grammar requires.

Nb: Until we finish building the predicate, we have no reason to reduce anything into the nonterminal *Sentence*

---

Types of Parsers (continued)

- Parsers can be either **table-driven** or **handwritten**:
  - Table-driven parsers perform the parsing using a driver procedure and a table containing pertinent information about the grammar. The table is usually generated by automated software tools called **parser generators**.
  - Handwritten parsers are hand-coded using the grammar as a guide for the various parsing procedures.
Types of Parsers (continued)

• LL(1) and LR(1) parsers are table-driven parsers which are top-down and bottom-up respectively.
• Recursive-descent parsers are top-down hand-written parsers.
• Operator-precedence parsers are bottom-up parsers which are largely handwritten for parsing expressions.

What is top-down parsing?

• Top-down parsing is a parsing-method where a sentence is parsed starting from the root of the parse tree (with the “Start” symbol), working recursively down to the leaves of the tree (with the terminals).
• In practice, top-down parsing algorithms are easier to understand than bottom-up algorithms.
• Not all grammars can be parsed top-down, but most context-free grammars can be parsed bottom-up.
An example of top-down parsing

Let’s consider the expression grammar:

$$E ::= E + T | T$$

$$T ::= T * F | F$$

$$F ::= id | const | (E)$$

How will it begin parsing the expression:

$$3 * x + y * z$$

LL(k) grammars

• Top-down grammars are referred to as LL(k) grammars:
  – The first L indicates Left-to-Right scanning.
  – The second L indicates Left-most derivation
  – The k indicates k lookahead characters.

• We will be examining LL(1) grammars, which spot errors at the earliest opportunity but provide strict requirements on our grammars.
LL(1) grammars

- LL(1) grammars determine from a single lookahead token which alternative derivation to use in parsing a sentence.
- This requires that if a nonterminal A has two different productions:
  \[ A ::= \alpha \quad \text{and} \quad A ::= \beta \]
  - that \( \alpha \) and \( \beta \) cannot begin with the same token.
  - \( \alpha \) or \( \beta \) can derive an empty string but not both.
  - if \( \beta \Rightarrow^* \epsilon \), \( \alpha \) cannot derive any string that begins with a token that could immediately follow A.

Converting an expression grammar into LL(1) form

- If our expression grammar is originally:
  \[
  E ::= E + T \mid T \\
  T ::= T * F \mid F \\
  F ::= \text{id} \mid \text{const} \mid (E)
  \]
- We must convert to the following form if it is to be LL(1):
  \[
  E ::= T E' \\
  E' ::= + T E' \mid \epsilon \\
  T ::= F T' \\
  T' ::= * F T' \mid \epsilon \\
  F ::= \text{id} \mid \text{const} \mid (E)
  \]
Once the grammar is in LL(1) form, we create a table showing which production we use in parsing each nonterminal for every possible lookahead token:

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>E'</th>
<th>T</th>
<th>T'</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>const</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. $E ::= TE'$
2. $E' ::= +TE'$
3. $E' ::= \epsilon$
4. $T ::= FT'$
5. $T' ::= *FT'$
6. $T' ::= \epsilon$
7. $F ::= id$
8. $F ::= const$
9. $F ::= ( E )$

**The Parsing Algorithm**

Processing context-free expressions requires the use of a stack. The Parsing algorithm uses a stack:

Place the start symbol in a node and push it onto the stack.
Fetch a token
**REPEAT**
   - Pop a node from the stack
   - IF it contains a terminal, match it to the current token (no match indicates a parsing error) and fetch another token
   - ELSE IF it contains a nonterminal, look it up in the production table using the nonterminals and the current token. Place the variables in REVERSE order on the stack
UNTIL the stack is empty
Recursive-Descent Parsing

• Recursive-descent parsing is a top-down parsing technique which shows a series of recursive procedures to parse the program.

• There is a separate procedure for each individual nonterminal.

• Each procedure is essentially a large if-then-else structure which looks for the appropriate tokens when the grammar requires a particular terminal and calls another procedure recursively when the grammar requires a nonterminal.

Recursive-Descent Parsing of Expressions

```c
#include <ctype.h>
#include <stdlib.h>
#include <stdio.h>

int token; /* holds the current input character for the parse */

/* declaration to allow arbitrary recursion */
void command(void);
int expr(void);
int term(void);
int factor(void);
int number(void);
int digit(void);
```
void error(void) {
    printf("parse error\n");
    exit(1);
}

void getToken(void) {
    /* tokens are characters */
    token = getchar();
}

void match(char c) {
    if (token == c) getToken();
    else error();
}

void command(void) /* command -> expr '\n' */ {
    int result = expr();
    if (token == '\n') /* End the parse and print the result */
        printf("The result is %d\n", result);
    else
        error();
}
int expr(void)
/* expr \rightarrow term \ ('\+' \ term \ ) */
{
    int result = term();
    while (token == '+') {
        match('+');
        result += term();
    }
    return(result);
}

int term(void)
/* term \rightarrow factor \ ('\*' \ factor \ ) */
{
    int result = factor();
    while (token == '*') {
        match('*');
        result *= factor();
    }
    return(result);
}
int factor(void)
/* factor -> '(' expr ')' | number */
{
    int result;
    if (token == '(') {
        match('(');
        result = expr();
        match(')');
    }
    else
    result = number();
    return(result);
}

int number(void)
/* number -> digit {digit } */
{
    int result = digit();
    while (isdigit(token))
        /* The value of a number with a new trailing digit is its previous value shifted by a decimal place plus the value of the new digit */
        result = 10 * result + digit();
    return(result);
}
int digit(void)
/* digit -> '0' | '1' | '2' | '3' | '4'
 | '5' | '6' | '7' | '8' | '9' */
{
    int result;
    if (isdigit(token)) {
        /* The numeric value of a digit character
           is the difference between its ASCII
           value and the ASCII value of the
           character '0' */
        result = token - '0';
        match(token);
    } else
        error();
    return(result);
}

void parse(void)
{
    getToken(); /* Get the first token */
    command(); /* Call the parsing
                procedure for the start symbol */
}

int main(void)
{
    parse();
    return(0);
}
Bottom-up Parsing

- Bottom-up parsers parse a program from the leaves of a parse tree, collecting the pieces until the entire parse tree is built all the way to the root.
- Bottom-up parsers emulate pushdown automata:
  - requiring both a state machine (to keep track of what you are looking for in the grammar) and a stack (to keep track of what you have already read in the program).
  - making it fairly easy to automate the process of creating the parser
  - ensuring that all context-free grammars can be parsed by this method.

Bottom-up parsers as shift-reduce parsers

- Bottom-up parsers are frequently called shift-reduce parsers because of their two basic operations:
  - A shift involves moving pushing the current input token onto the stack and fetching the next input token.
  - A reduce involves popping all the variables that comprise the right-sentential form for a nonterminal and replacing them on the stack with the equivalent nonterminal that appears on the left-hand side of that production.
  - While shifting involve pushing and reducing involve popping, do not think of them as equivalent: a shift also involve advancing the input token stream and a reduce involves zero or more pops followed by a push.
LR(k) grammars

• Bottom-up grammars are referred to as LR(k) grammars:
  – The first L indicates Left-to-Right scanning.
  – The R that is second indicates Right-most derivation
  – The k indicates k lookahead characters.
• There should be no need for anything more than a single lookahead, i.e, an LR(1) grammar.

An example - a LR(0) grammar
An LR(0) grammar does not use a lookahead character to determine the action that it will take - the current token will be used to determine the state into which it will go.

Consider the following grammar:

\[ E ::= E + T | T \]
\[ T ::= + F | - F | F \]
\[ F ::= \text{id} | \text{const} \]
An example - a LR(0) grammar (continued)

Let’s write out our grammar and add to it a special first production with a special start symbol $S$:

1. $S ::= E \, \$ \quad$ (indicates that the expression is followed by EOF)
2. $E ::= E + T$
3. $E ::= T$
4. $T ::= +F$
5. $T ::= -F$
6. $T ::= F$
7. $F ::= id$
8. $F ::= const$

---

The LR(0) parse table

<table>
<thead>
<tr>
<th>GOTO</th>
<th>ACTION</th>
<th>$+$</th>
<th>$-$</th>
<th>id</th>
<th>const</th>
<th>$S$</th>
<th>$E$</th>
<th>$T$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>r6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>s</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>r7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>acc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tracing LR(0) parsing

There are 3 parsing operations:

Shift - moving a token and state onto the stack (we find the state using the GOTO table).

Reduce n - we pop enough items from the stack to form the right side of production n and then we push the nonterminal on its left side of production n on to the stack, together with the state indicated by the GOTO table.

Accept - we accept the program as completely and correctly parsed and terminate execution.

Tracing LR(0) parsing - an example

Example - the expression \(-27 + x\)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

We place the state 0 and the EOF marker $ on the stack. The action for state 0 is shift. We place the - and GOTO(0, -) = 5 on the stack.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>const</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

The action for state 5 is shift. We place the constant on the stack together with GOTO(5, const) = 7.

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>$</td>
<td></td>
</tr>
</tbody>
</table>

The action for state 7 is reduce by production 8. Pop the const (and state 7). Push F and GOTO(5,F) = 11.
### Tracing LR(0) parsing - an example (continued)

<table>
<thead>
<tr>
<th>State</th>
<th>Symbol</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>Shift</td>
</tr>
<tr>
<td>8</td>
<td>+</td>
<td>Shift</td>
</tr>
<tr>
<td>6</td>
<td>id</td>
<td>Shift</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>Reduce</td>
</tr>
<tr>
<td>10</td>
<td>T</td>
<td>Reduce</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>Reduce</td>
</tr>
<tr>
<td>0</td>
<td>$</td>
<td>Reduce</td>
</tr>
</tbody>
</table>

- The action for state 11 is reduce by production 5. Pop the - and F (along with states 5 and 11) and push the T together with GOTO(0,T) = 2.
- The action for state 2 is reduce by production 3. Pop the T (and state 2). Push the E and GOTO(0,E) = 1.
- The action for state 1 is shift. We move the + onto the stack together with GOTO(1, +) = 8.
- The action for state 6 is reduce by production 7. We pop the id and state 6. We push F and GOTO(8, F) = 3.
- The action for state 3 is reduce by production 6. We pop the F and state 3. We push T and GOTO(8, T) = 10.
Tracing LR(0) parsing - an example (continued)

The action for state 10 is reduce by production 2. We pop the T (and state10), the + (and state8) and the E (and state1). We push the E and GOTO(0,E) = 1.

The action for state 1 is shift. We push the $ and GOTO (1,E) = 12 onto the stack.

The action for state 12 is accept. The only item on the stack (excluding the $s) is E, which is the start symbol in our expression grammar.

YACC and Bison

- YACC (Yet Another Compiler Compiler) is a program that automatically generates a parser based on specifications written in a syntax similar to BNF.
- Bison is its GNU equivalent.
YACC Format

{%/* code to insert at beginning of the parser */
/* Other YACC definitions, if necessary */
/* grammar and associated actions */
/* auxiliary procedures */
%
%
%
command: expr \n {printf("The result is: %d\n", $1);} 

expr : expr + term {$$ = $1 + $3; } 
      | term {$$ = $1; } 

term : term * factor {$$ = $1 * $3; } 
     | factor {$$ = $1; } 

;
factor: number { $$ = $1; } \\
| '(' expr ')' { $$ = $2; } \\
;
number: number digit { $$ = 10 * $1 + $2; } \\
;
digit: '0' { $$ = 0; } \\
| '1' { $$ = 1; } \\
| '2' { $$ = 2; } \\
| '3' { $$ = 3; } \\
| '4' { $$ = 4; } \\
| '5' { $$ = 5; } \\
| '6' { $$ = 6; } \\
| '7' { $$ = 7; } \\
| '8' { $$ = 8; } \\
| '9' { $$ = 9; } \\
;

%%
main()
{
    yyparse();
    return(0);
}
int yylex(void)
{
    static int done = 0;
    int c;
    if (done) return(0); /* stop parsing */
    c = getchar();
    if (c == '\n')
    /* next call will end parsing */
        done = 1;
    return(c);
}
int yyerror(char *s)
{
    /* allows for print error message */
    printf("%s\n", s);
}

Semantic Analysis

- Semantic analysis is the phase where the meaning of the syntactic constructs is recognized and synthesis of the object program is begun.
- While it is possible for the semantic analyzer to produce an object program, the end result of this phase is usually a language-independent, machine-independent intermediate representation of the program.
Functions of the Semantic Analyzer

The most common functions of semantic analysis include:

• Symbol-Table Management
• Insertion of Implicit Information
• Error Detection
• Macro processing and Compile-Time Operations

Symbol-Table Management

• The symbol table is one of a translator’s central data structures.
• The symbol table hold data regarding every lexeme in a program, including keywords, operators, identifiers and literals.
• The data within the symbol table is assembled during the analysis phases of translation and used during the generation phases.
Insertion of Implicit Information

- Some information in the source program is implicit and must be made explicit, e.g., the type of variables declared by default in FORTRAN.

Error Detection

- All three analysis phases must be prepared to handle incorrect programs.
- Syntactic errors involve the incorrect use of grammatical constructs.
- Semantic errors involve cases where the semantics are in error, e.g., incompatible data types.
Macro processing and Compile-Time Operations

- A macro, its simplest form, is a piece of text that is inserted into a program where the appropriate macro call appears. In more complex form, it may involve the replacement of formal parameters with their actual values.
- An example of compile–time operations is conditional compilation, where a segment of source code will include compiled depending on the validity of a test condition.

Synthesis of the Object Program

The final stages of the translation process involve the generation of the object program. This includes:

- Optimization
- Code Generation
- Linking and Loading
Optimization

- Optimization is improving the efficiency of an object program (execution time and/or storage requirements), usually by removing inefficiencies created by the automated translation process.
- Optimization may be local (confined to a small section of code which will always be executed as a unit) or global (tracing through the logical sequence of instructions)

An Example of Optimization

- The statement
  \[ A = B + C + D \]
- creates the intermediate code
  - \( \text{Temp1} = B + C \)
  - \( \text{Temp2} = \text{Temp1} + D \)
  - \( A = \text{Temp2} \)
  which generates the object code
    - Load B  \((\text{Step a})\)
    - Add C
    - Store Temp1
    - Load Temp1  \((\text{Step b})\)
    - Add D
    - Store Temp2
    - Load Temp2  \((\text{Step c})\)
    - Store A
Code Generation

- After the intermediate representation is optimized, object code is created based on this representation, usually in machine language.
- This object code may need optimization itself.
- The object code may be executable or may need linking.

Linking and Loading

- The various object modules must be combined into one executable program.
- References to external variables and procedures must be resolved and external procedures must be includes in the executable module. This information is found in the **loader table**.
Bootstrapping

• It is common to write a compiler in the source language.
• Once the compiler is completed, it is used to translate itself into an executable program. This is known as bootstrapping.
• Frequently the first compilation of a compiler is done by hand due to the lack of a working compiler.

Diagnostic Compilers

• Production compilers usually concentrate on creating object code that can be executed efficiently. Consequently, these compilers do not always offer error messages that are useful, especially to novice programmers.
• It was particularly popular in the 1960s to create compilers that could quickly translate programs and create compiler-time and runtime error messages that were particularly helpful to novice programmers.
• These diagnostic compilers include WATFOR and PL/C.