A few basic definitions

*Translate* - *v*, a. to turn into one’s own language or another.
b. to transform or turn from one of symbols into another

*Translator* - *n*, someone or something that translates.

*Compilers* are translators that produce object code (machine-runnable version) from source code (human-readable version).

*Interpreters* are translators that translate only as much as is necessary to run the next statement of the program.
• **Source Language** - the language in which the source code is written

• **Target Language** - the language in which the object code is written

• **Implementation Language** - Language in which the compiler is written

---

**Example:**

Source language program

Compiler

Target language program

C++ or Java program

Compiler

Pentium machine language program

---

**Choice of an Implementation Language**

The implementation language for compilers used to be assembly language.

It is now customary to write a compiler in the source language.

Why? The compiler itself can then be used as a sample program to test the compiler’s ability to translate complex programs that utilize the various features of the source language.
The Compiling Process

Source Code → Compiler → Object Module → Linker → Executable version → Assembler version

The Interpretation Process

Source Code → Interpreter → Intermediate Version → Interpreter → Output → Input
**Source language** - designed to be machine-translatable (“Context-free grammar”)

e.g., FORTRAN, COBOL, Pascal, C, BASIC, LISP

- Portable, i.e., programs can be moved from one computer to another with minimal or no rewriting.
- The Level of Abstraction matches the problem and not the hardware.
- Does not require an intimate knowledge of the computer hardware

**Assembly language** - machine acronyms for machine language commands.

e.g., mov ax, 3

- Eliminates the worst of the details, but leaves many to be dealt with.

**Object Module** - a machine language version of the program lacking some necessary references.

e.g., on the Intel 8x86 (in real mode)

```
1011 1 000 0000 0000 0000 0011
```

mov (from immediate to register)

```
16-bit AX the immediate value
value reg.
```

**Load Module** - a machine language version that is complete with addresses of all variables and routines.
Other types of Compilers

There are compilers that do not necessarily follow this model:

*Load-and-go compilers* generate executable code without the use of a linker.

*Cross compilers* run on one type of computer and generate translations for other classes of computers.

*Cross-language compilers* translate from one high-level language to another. (e.g., C++ to C)

The organization of a compiler

- The various components of a compiler are organized into a **front end** and a **back end**.
- The front end is designed to produce some intermediate representation of a program written in the source language.
- The back end is designed to produce a program for a target computer from the intermediate representation.
Why Separate Front and Back Ends?

BASIC
COBOL
C++
Java
Ada

PC
JVM
MacIntosh
Linux Workstation

Why Generate Intermediate Code?

BASIC
COBOL
C++
Java
Ada

Quadruples
IBM PC
JVM
MacIntosh
Linux Workstation
Components of a Compiler - The Front End

Source Code → Lexical Analyzer (Scanner) → Tokens → Syntactic Analyzer (Parser) → Parse tree

Intermediate Code → Intermediate Code Generator → Annotated AST

Components of a Compiler - The Back End


Optimized Object Code → Machine-Dependent Optimizer
Lexical Analysis

- The lexical analyzer (or scanner) breaks up the stream of text into a stream of strings called "lexemes" (or token strings).
- The scanner checks one character at a time until it determines that it has found a character which does not belong in the lexeme.
- The scanner looks it up in the symbol table (inserting it if necessary) and determines the token associated with that lexeme.

Lexical Analysis (continued)

- **Token** - the language component that the character string read represents.
- Scanners usually reads the text of the program either a line or a block at a time. (File I/O is rather inefficient compared to other operations within the compiler.)
Syntactic Analysis

- A syntactic analyzer (or parser) takes the stream of tokens determines the syntactic structure of the program.
- The parser creates a structure called a parse tree. The parser usually does not store the parse in memory or on disk, but it does formally recognize program’s the grammatical structure.

Syntactic Analysis (continued)

The grammar of a language is expressed formally as

\[ G = (T, N, S, P) \]

where

- \( T \) is a set of terminals (the basic, atomic symbols of a language).
- \( N \) is a set of nonterminals (symbols which denote particular arrangements of terminals).
- \( S \) is the start symbol (a special nonterminal which denotes the program as a whole).
- \( P \) is the set of productions (rules showing how terminals and nonterminal can be arranged to form other nonterminals).
Syntactic Analysis (continued)

• An example of *terminal* would be
  PROGRAM, ID, and :=.
• An example of a *nonterminal* would be
  Program, Block and Statement.
• The *start symbol* in most cases would be
  Program
• An example of a *production* would be
  Block ::= BEGIN Statements END

Semantic Analysis

• Semantic analysis involves ensuring that the semantics (or meaning) of the program is correct.
• It is quite possible for a program to be correct syntactically and to be correct semantically.
• Semantic analysis usually means making sure that the data types and control structures of a program are used correctly.
Semantic Analysis (continued)

- The various semantic analysis routines are usually incorporated into the parser and do not usually comprise a separate phase of the compiling process.
- The process of generating an intermediate representation (usually an abstract syntax tree) is usually directed by the parsing of the program.

A More Realistic View of the Front End
Error detection in Source Programs

- All the previous stages analyze the program, looking for potential errors.

```pascal
FOR i != 1 TO n DO WriteLn;

↑
Lexical error

IF x > N THEN Y := -3; ELSE Y := 3;

↑
Syntactic error
```

Error Detection in Source Programs

```pascal
PROGRAM Average;
VAR Average : Integer;
    Sum, Val1, Val2, Val3 : Real;
BEGIN
    Val1 := 6.0;
    Val2 := 4;  ← Mixed-typed assignment
    Val3 := 37.5;
    Sum := Val1 + Val2 + Val3;
    Average := (Val1 + Val2 + Val3) DIV 3
END. { Average }
```

Semantic error
Intermediate Code Generation

• The intermediate code generator creates a version of the program in some machine-independent language that is far closer to the target language than to the source language.
• The abstract syntax tree may serve as an intermediate representation.

Object Code Generation

• The object code generator creates a version of the program in the target machine’s own language.
• The process is significantly different from intermediate code generation.
• It may create an assembly language version of the program, although this is not the usual case.
An example of the compiling process

```c
int main()
{
    float average;
    int x[3];
    int i, sum;

    x[0] = 3;
    x[1] = 6;
    x[2] = 10;
    sum = 0;
    for (i = 0; i < 3; i++)
    {
        sum += x[i];
    }
    average = sum/3;
    return(0);
}
```

An example of Lexical Analysis

The tokens are:

```
INT ID ( ) { FLOAT ID ; INT ID [ NUMLITERAL ] ; INT ID , ID ; ID [ NUMLITERAL ] = NUMLITERAL ;
and so on
```
A sample parse tree

Program

function-definition

type-specifier

INT

function-declarator

ID

function-body

type-decl-list

ID

function-body

{ } statement list

The corresponding Abstract Syntax Tree

ID

INT

VOID

decls

ID

ID

ID

Statements
The intermediate code for the example

main:
  x[0] = 3
  x[1] = 6
  x[2] = 10
  sum = 0
  i = 0

  t1:
    if i >= 3 goto t2:
    t3 = x[i]
    Sum = Sum + t3
    goto t1

  t2:
    Average := Sum / 3

The assembler code for the example

_main PROC NEAR ; COMDAT
  ; File C:\MyFiles\Source\avg3\avg3.c
  ; Line 4
  push ebp
  mov ebp, esp
  sub esp, 8
  push ebx
  push esi
  push edi
  ... ...
  mov DWORD PTR _x$[ebp], 3
  mov DWORD PTR _x$[ebp+4], 6
  mov DWORD PTR _x$[ebp+8], 10
  mov DWORD PTR _sum$[ebp], 0
  ... ...
The Symbol Table

- The symbol table tracks all symbols used in a given program.
- This includes:
  - Key words
  - Standard identifiers
  - Numeric, character and other literals
  - User-defined data types
  - User-defined variables

The Symbol Table (continued)

- Symbol tables must contain:
  - Token class
  - Lexemes
  - Scope
  - Types
  - Pointers to other symbol table entries (as necessary)
“Shaper” - an example of a translator

- Shaper is a “microscopic” language which draws rectangles, square and right isosceles triangles on the screen.
- Shaper has three statements:
  - `RECTANGLE [WIDE or LONG] Number BY Number`
  - `SQUARE SIZE Number`
  - `TRIANGLE SIZE Number`
- Example
  - `RECTANGLE LONG 6 by 5`
  - `RECTANGLE WIDE 15 BY 30`
  - `SQUARE SIZE 9`
  - `TRIANGLE SIZE 5`

The “Shaper” Translator

```c
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

enum tokentype {tokby, tokeof, tokerror, 
toklong, toknumber, 
tokrectangle, toksize, 
toksquare, toktriangle, 
tokwide};

char *tokenname[] = {"by","eof", "error", 
"long", "number", "rectangle", 
"size","square", "triangle", 
"wide"};
```
const int filenamesize = 40, 
    tokenstringlength = 15, 
    numtokens = 10;

int wordsearch(char *test, char *words[], 
    int len);

class scanner  
  { 
public:
    scanner(int argc, char *argv[]); 
    scanner(void); 
    ~scanner(void); 
    tokentype scan(char tokenstring[]);
private:
    tokentype scanword(char c, char tokenstring[]); 
    tokentype scannum(char c, char tokenstring[]); 
    ifstream infile;
  };

scanner::scanner(int argc, char *argv[]) 
  {
    char filename[filenamesize];

    // If there is only one argument, it must be
    // the program file for Shaper. That means
    // that we need the source file.
    // If there are two arguments, we have it
    // already as the second argument. If there
    // are more, there must be a mistake.

    if (argc == 1)    {
      cout << "Enter program file name\t?";
      cin >> filename;
    }
    else if (argc == 2)
      strcpy(filename, argv[1]);
else {
    cerr << "Usage: Shaper <filename>\n";
    exit(1);
}

infile.open(filename, ios::in);
if (!infile) {
    cerr << "Cannot open " << filename << endl;
    exit(1);
}

// scanner() - Default constructor for the scanner
scanner::scanner(void)
{
    char filename[filenamesize];

    cout << "Enter program file name\t?";
    cin >> filename;

    // Open the input file
    infile.open(filename, ios::in);
    if (!infile) {
        cerr << "Cannot open " << filename << endl;
        exit(1);
    }
}
scanner::~scanner(void)
{
  infile.close();
}

// scan() - Scan out the words of the language
tokentype scanner::scan(char tokenstring[])
{
  char c;

  // Skip the white space in the program
  while ( !infile.eof() &&
        isspace(c = infile.get()) )
    ;

  // If this is the end of the file, send the
  // token that indicates this
  if ( infile.eof() )
    return(tokeof);
// If it begins with a letter, it is a word. If
// begins with a digit, it is a number. Otherwise,
// it is an error.
if (isalpha(c))
    return(scanword(c, tokenstring));
else if (isdigit(c))
    return(scannum(c, tokenstring));
else
    return(tokerror);

// scanword() - Scan until you encounter
// something other than a letter.
// It uses a binary search to find
// the appropriate token in the
// table.
tokentype scanner::scanword(char c,
                                char tokenstring[])
{
    int i = 0;
tokentype tokenclass;

    // Build the string one character at a time.
    // It keep scanning until either the end of
    // file or until it encounters a non-letter
    tokenstring[i++] = c;
while (!infile.eof() &&
    isalpha(c = infile.get()))
    tokenstring[i++] = c;
tokenstring[i] = '\0';

    // Push back the last character
    infile.putback(c);

    // Is this one of the legal keywords for
    // Shaper? If not, it's an error
    if ((tokenclass =
        (tokentype)wordsearch(tokenstring,
         tokenname, numtokens))
        == -1)
        return(tokerror);
    else
        return(tokenclass);
}

//scannum() - It returns the token toknumber.
// The parser will receive the
// number as a string and is
// responsible for converting it
// into numerical form.
tokentype scanner::scannum(char c,
    char tokenstring[])
{
    int i = 0;

    // Scan until you encounter something that
    // cannot be part of a number or the end of
    // file
    tokenstring[i++] = c;
while (!infile.eof() &&
    isdigit(c = infile.get()))
    tokenstring[i++] = c;

tokenstring[i] = '\0';

// Push back the last character
infile.putback(c);
return(toknumber);
}

Managing the “Symbol Table”

//wordsearch() - A basic binary search to find a string in an array of strings
int wordsearch(char *test, char *words[],
    int len)
{
    int low = 0, mid, high = len - 1;

    // Keep searching as long as we haven't searched the whole array
    while (low <= high) {
        mid = (low + high)/2;
        if (strcmp(test,words[mid]) < 0)
            // search the lower half
            high = mid - 1;
else if (strcmp(test, words[mid]) > 0)  
    // search the upper half  
    low = mid + 1;
else
    // We found it!!
    return(mid);
}
// It isn't there  
return(-1);
}

Parsing A “Shaper” Program

class parser : scanner {
public:
    parser(int argcount, char *args[]);
    parser(void);
    void ProcProgram(void);
private:
    void ProcRectangle(void);
    void ProcSquare(void);
    void ProcTriangle(void);
    tokentype tokenclass;
    char tokenstring[tokenstringlength];
};
// parser() - A constructor that passes
// initial values to the base
// class
class parser::parser(int argc, char *argv[])
    : scanner(argc, argv)
{
    // Get the first token
tokenclass = scan(tokenstring);
}

// parser() - A default constructor
class parser::parser(void)
{
    // Get the first token
tokenclass = scan(tokenstring);
}

void parser::ProcProgram(void)
{
    // Get a token and depending on that token's
    // value, parse the statement.
    while (tokenclass != tokeof)
    {
        switch(tokenclass){
        case tokrectangle:
            ProcRectangle();
            tokenclass = scan(tokenstring);
            break;

        case toksquare:
            ProcSquare();
            tokenclass = scan(tokenstring);
            break;
        }
case toktriangle:
    ProcTriangle();
    tokenclass = scan(tokenstring);
    break;

default: cerr << tokenstring
    << " is not a legal"  
    << " statement\n"
    << endl;
    exit(3);
}

//ProcRectangle() - Parse the rectangle
// command and if there
// are no errors, it will
// produce a rectangle
// on the whose dimensions
// are set by the
// rectangle statement.
void parser::ProcRectangle(void)
{
    int shape, columns, rows;
    char tokenstring[tokenstringlength];

    // The next word should be wide or long to
    // indicate whether there are more rows or
    // columns. This is not really necessary for
    // the statement to work correctly, but is a
    // good simple illustration of how type
    // checking works.

if ((tokenclass = scan(tokenstring)) != tokwide 
    && tokenclass != toklong) {
    cerr << "Expected ""wide"" or ""long"" instead of " << tokenstring 
    << endl;
    exit(4);
}

// Get the number of columns and if it is a number
if ((tokenclass = scan(tokenstring)) != toknumber) {
    cerr << "Expected number instead of " << tokenstring << endl;
    exit(5);
}

// The token by is simply a separator but the grammar requires it.
if ((tokenclass = scan(tokenstring)) != tokby)
    cerr << "Expected ""by"" instead of " << tokenstring << endl;

// Get the number of rows and if it is a number
if ((tokenclass = scan(tokenstring)) != toknumber) {
    cerr << "Expected number instead of " << tokenstring << endl;
    exit(5);
}
}
void parser::ProcRectangle(void)
{
    int shape, columns, rows;
    char tokenstring[tokenstringlength];

    // The next word should be wide or long to indicate
    // whether there are more rows or columns. This is
    // not really necessary for the statement to work
    // correctly, but is a good simple illustration of
    // how type checking works.
    if ((tokenclass = scan(tokenstring)) != tokwide
        && tokenclass != toklong) {
        cerr << "Expected " << tokenstring << " instead"
            << endl;
        exit(4);
    }

    // The shape is indicated by whether this
    // token was wide or long
    shape = tokenclass;

    // Get the number of columns and if it is a number,
    // convert the character string into an integer
    if ((tokenclass = scan(tokenstring)) != toknumber) {
        cerr << "Expected number instead of "
            << tokenstring << endl;
        exit(5);
    }

    columns = atoi(tokenstring);

    // The token by is simply a separator but the
    // grammar requires it.
    if ((tokenclass = scan(tokenstring)) != tokby){
        cerr << "Expected " << tokenstring << endl;
    }
}
// Get the number of rows and if it is a
// number, convert the character string into
// an integer.
if ((tokenclass = scan(tokenstring)) != toknumber) {
    cerr << "Expected number instead of "
    << tokenstring << endl;
    exit(5);
}
rows = atoi(tokenstring);

// A long rectangle should have more rows than
// columns and a wide rectangle will have the
// opposite. This illustrates how type
// checking works on a facile level.

if (shape == toklong && columns < rows
    || shape == tokwide
    && columns > rows) {
    cerr << "A " << tokenname[shape]
        << " rectangle cannot be " << columns
        << " by " << rows << endl;
    exit(6);
}
DrawRectangle(columns, rows);