Basic Concepts

The difficulty of a compiler is not that any single part is difficult, but that there are so many parts that work together. To begin with just work on and eventually one reaches a "click point" where the system as a whole makes sense.

The Parts of a Compiler

Compilers are complex programs. Thus they are often broken up into several distinct chunks, called *passes*, that communicate via temporary files. Passes are only part of the compilation process, the full process of creating an executable image from source-code can involve several stages (preprocessing, assembly, linking, ans so forth). The assembler, linker ans so forth are not considered to be part of the compiler.

The structure of a typical four-pass compiler is as follows:

- **Preprocessor:** This typically handles macro substitutions, strip comments from the source code, and handle various housekeeping tasks to avoid burdening the compiler.
- **Compiler:** Generates intermediate language
  - *Lexical Analyzer*
  - *Parser*
  - *Code Generator*
- **Optimizer:** This improves the quality of the generated intermediate code.
- **Back End:** Translates the optimized code to real assembly language or some form of binary, executable code.

The Lexical Analyzer

A *phase* is an independent task used in the compilation process. The *lexical analyzer* phase of a compiler translates the input into a form that's more usable by the reset of the compiler. The lexical analyzer looks at the input stream as a collection of basic language elements called *tokens*. The original string that comprises the token is called a *lexme*. There is no one-to-one relationship between lexmes and tokens. In general, a lexical analyzer recognizes the token that matches the longest lexme.

A lexical analyzer translates lexemes into tokens. The tokens are typically represented internally as unique integers or an enumerated type. Both components are always required–the token itself and the lexeme, which is needed to differentiate the various **name** or **number** tokens from one another.

You can have a token for every input symbol, or several symbols can be merged into a single token–for example, the `>`, `>=`, `>>`, and `>>=`, can be treated either as four tokens, or as a single **comparison-operator** token–the lexme is used to disambiguate the tokens. In general, arithmetic operators with the same precedence and associativity can be grouped together, type-declaration keywords can be combined, and so forth.

The lexical analyzer is typically a self-contained unit that interfaces with the rest of the compiler via a small number of subroutines and global variables. The parser calls the lexical-analyzer every time it needs a new token, and the analyzer returns that token and associated lexme.

The Parser

Compilers are language translators. Consequently, a good deal of the theoretical side of the subject is borrowed from linguistics. One such concept is the idea of parsing. To *parse* an English sentence is to break it up into component parts in order to analyze it grammatically.

A compiler preforms this same process in the *parser* phase, though it usually represents the parsed sentence in a tree form.

This kind of graph is formally called a *syntax diagram*. You can expand the syntax tree, however, to show the grammatical structure as well as the syntactic structure, this is called a *parse tree*.

A *sentence* is a collection of tokens that follow a well-defined grammatical structure. In the case of a compiler, the sentence is typically an entire computer program.

A parser is a group of subroutines that converts a token stream into a parse tree, and a parse tree is a structural representation of the sentence being parsed. Another way to look at it is that the parse tree represents the sentence in a hierarchical fashion, moving from a general description of the sentence down to the specific sentence being parsed at the leaves.

The Code Generator

The last part of the compiler is the code generator. Most compiler generate code as the parse progresses. That is, the code is generated by the same subroutines that are parsing the input stream. It is possible for the parser to create a parse tree for the entire input file, which is then traversed by a distinct code generators. A third possibility is for the parser to create an intermediate-language representation of the input from which a syntax tree can be reconstructed by an optimization pass.

Though compilers can generate object code directly, they often defer code generation to a second program. Instead of generating machine language directly, they create a program in an *intermediate language* that is translated by the compiler's *back end* into actual machine language.

There are advantages and disadvantages to an intermediate-language approach to compiler writing. The main disadvantage is lack of speed. The main advantages are optimization, and flexibility. A few optimizations, such as constant folding can be done in the parser.

Intermediate languages give you flexibility as well. A single lexical-analyzer/parser front end can be used to generate code for several different machines by providing separate back ends that translate a common intermediate language to a machine-specific assembly language. Conversely, you can write several front ends that pare several different high-level languages, but which all output the same intermediate language. This way, compilers for several languages can share a single optimizer and back end.

A final use of an intermediate language is found in incremental compilers or interpreters. These programs directly execute intermediate code, rather than translating it to binary first.

The intermediate language utilized in this book is a subset of `C`.

Representing Computer Languages

In compiler applications, an good abstraction is one that describes the language being compiled in a way that reflects the internal structure of the compiler itself.

Grammars and Parse Trees

The most common method used to describe a programming language is a formal grammar.

Formal grammars are most often represented in a modified *Backus-Naur Form* (BNF). A BNF representation starts with a set of tokens, called *terminal* symbols, and a set of definitions, called *nonterminal* symbols. The definitions create a system in which every legal structure in the language can be represented. One operator is supported, the `::=` operator, translated by the phrase "is defined as".

Each rule is called a *production*. This grammar is a Context-free grammar. All symbols that are *italicized* are nonterminal symbols and the symbols in **boldface** are terminal symbols.

A parse tree is the tree of replacements from the grammar. Using the parse tree, Terminal symbols are always leaves in the tree, and nonterminal symbols are always interior nodes.

An Expression Grammar

This table represents a grammar that recognizes a list of one or more statements, each of which is an arithmetic expression followed by a semicolon.

statements	→	*expression* **;**
		expression **;** *statements*
expression	→	*expression* + *term*
		term
term	→	*term* * *factor*
		factor
factor	→	**number**
		(*expression*)

Since the grammar is recursive, it stands to reason that recursion can be used to parse the grammar. The recursion is also important from a structural perspective – it is the recursion that makes it possible for a finite grammar to recognize an infinite number of sentences.

There is a major problem in the grammar. The leftmost symbol on the right-hand side of several of the productions is the same symbol that appears on the left-hand side. This could cause a parser to loop forever, repetitively replacing the left most symbol in the right-hand side with the entire right-hand side. This happens in the production; *expression* → *expression* + *term*.

The parser must also be able to choose between one of several right-hand sides by looking at the next input symbol. It cannot make this decision in production 5 and 6, because both of the latter productions can start with the same set of terminal symbols. This is called a *conflict*, and one of the most difficult tasks of a compiler design is creating a grammar that has no conflicts in it. The next input symbol is called the *lookahead symbol* because the parser looks ahead at it to resolve a conflict.

Below is a corrected grammar that resolves these issues, although at the loss of readability, and understanding.

statements	→	
		expression **;** *statements*
expression	→	*term expression'*
expression'	→	+ *term expression'*

term	→	*factor term'*
term'	→	* *factor term'*
factor	→	**number**
		(*expression*)

The symbol is an end-of-input marker.

Syntax Diagrams

Syntax diagrams are useful in writing recursive-descent compilers because they translate directly into flow charts. They can also be used as a map that describes the structure of the parser.

In the syntax diagram, passing through a circled symbol removes a terminal from the input stream, and passing through a box represents a subroutine call that evaluates a nonterminal.

A Recursive-Descent Expression Compiler

Now we know enough to build a small compiler. Our goal is to take simple arithmetic expressions as input and generate code that evaluates those expressions at run time.

The Lexical Analyzer

The first order of business is defining a token set. With the exception of numbers and identifiers, all the lexmes are single characters. A **NUM_OR_ID** token is used both for numbers and identifiers; so, they are made up of a series of contiguous characters in the range `0-9`,`a-z`, or `A-Z`. The lexical analyzer translates a semicolon into a **SEMI** token, a series of digits into a **NUM_OR_ID** token, and so on. The lexical analyzer uses three external variables to pass information to the parser. `Text` points to the current lexeme; `Leng` is the number of characters in the lexeme; and `LineNum` is the current input line number.

This lexical analyzer uses a simple, buffered, input system, getting characters a line at a time from standard input, and then isolating tokens, one at a time, form the line. Another input line is fetched only when the entire line is exhausted. There are two main advantages to a buffered system. These are speed and speed. It is faster to read data in larger chunks. The second issue has to do with *lookahead* and *push back*. Lexical analyzers often must look ahead to

determine the lexeme, and then push back the unnecessary characters. This is easer done with a buffer, then the unbuffered input stream.

The first step is to loop through and skip all blank lines, and get to the first non white character. Next is the actual tokenization. Single-character lexemes are recognized in a switch case statement, and multiple-character `NUM_OR_ID` token is handled otherwise. When the function terminates `Text` points at the first character of the lexeme, and `Leng` holds its length.

The next time the function is called, it will advance the current pointer past the previous lexeme, and if the input buffer has been exhausted, it will get a new line.