Today we’ll take a quick look at typical parts of a compiler. This is to give a feeling for the overall structure.

We’ll then begin to consider a very simple example of a compiler — translates arithmetic expressions from infix to postfix notation.

Once we’ve seen how such a compiler can be built by hand, we’ll begin to study more powerful (automated) techniques, in detail.

We’ll focus on the “front end” — lexical analyzer through intermediate code generator.
Lexical Analysis (Scanning)

A source program is typically stored in a file that is essentially a sequence of bytes or characters. The function of a lexical analyzer (scanner) is to translate this stream of bytes into a sequence of “tokens.”

We’ll use an example to begin to flesh out the idea of a “token”...

Consider the following fragment of a program:

```c
{  
  float tab[10];  
  int x=2, y=3, z;  
  z = x + y * 10;  
}
```

Lexical analysis will eliminate white space, and turn character strings into tokens...

Lexical analysis might produce something like the following sequence of tokens and token-value pairs:

```c
{  
  <FLOAT><IDENTIFIER,tab><NUMBER,10>;  
  <NUMBER,2>,<IDENTIFIER,x>=<NUMBER,2>,<IDENTIFIER,y>=<NUMBER,3>,<IDENTIFIER,z>;  
  <IDENTIFIER,z>=<IDENTIFIER,x>+<IDENTIFIER,y>*<NUMBER,10>;  
}
```
Symbol Tables

The symbol table stores information about identifiers encountered in the source file during compilation. For example, it could be that

- a variable identifier has a name ("lexeme") and a type;
- a type identifier has a name and a type;
- a function identifier has a name, a parameter list, and a result type.

Typically, entries in the symbol table are partially constructed during lexical analysis, and they are used and elaborated in later steps.

Syntactic Analysis (Parsing)

Essentially, the task of a syntactic analyzer (parser) is to convert a sequence of tokens into a syntax tree.

For the previous example, we may have roughly the following syntactic structures:

- A block consists of declarations followed by statements.
- Each declaration consists of a type followed by a list of variable names.
- Each variable name has an optional pre-modifier, such as pointer (*), an identifier, and an optional post-modifier, which may be for array size or function type, or an initialization.
- A statement can be an assignment, which consists of a variable, followed by = and an expression.
- An expression can be an addition, a multiplication, \ldots

```plaintext
{ 
    float tab[10];
    int x=2, y=3, z;
    z = x + y * 10;
}
```
Below is a possible “abstract syntax tree” for the assignment

\[
z = x + y \times 10;
\]

Semantic Analysis

Semantic analysis includes activities like type checking.

Associate a type with each variable and each expression, and check for type consistency in

- variables: array element access, structure component access, pointer object access;
- expressions: the types of operands must be consistent with the operator;
- function calls: types of arguments must be consistent with function declaration;
- assignments.
At each stage in this process we may encounter errors in the input.

Some guidelines:

- Detect errors when possible.
- Identify them as specifically as possible: where they occur, what they are, etc.
- Don’t halt upon encountering an error. Instead try to continue in a reasonable fashion. (Involves some guessing.)
- Don’t try to repair errors.

Intermediate Code Generation

```
  =
  \ /
  \ 
  <ID,y> +
  \ /
  \ 
  <ID,x> *
  \ /
  \ 
  <ID,y> <NUMBER,10>
```

- Intermediate code:
  
  temp1 := 10
  temp2 := IDy * temp1
  temp3 := IDx + temp2
  IDz := temp3

- Intermediate code optimization:
  
  temp1 := IDy * 10
  IDz := IDx + temp1
Compiler Front End Construction Tools

- parser generators: yacc, bison.
  Implement powerful parsing algorithms too complex to be reliably written by hand.
  Input, at its simplest, is a “context-free grammar” specifying the syntactic structure of the source language.

- scanner generators: lex, flex.
  Generate lexical analyzers, quite similar to “finite automata.”
  Input, at its simplest, is a list of “regular expressions” specifying which input strings correspond to tokens.

- syntax-directed translation engines

We’ll play around with lex and yacc.

Simple Compiler Example

Translate infix expression to postfix expression.

Example:
\[
1 + 2 \rightarrow 12^+
\]
\[
3 - 1 + 2 \rightarrow 31^-2^+ \quad \text{(left associative operators)}
\]

- Rudimentary example of parsing, semantic analysis, intermediate code generation.
- Also, lexical analysis.

We’ll look this over quickly to get an overview of a very simple compiler front end.

We’ll also get a sense of how to do things by hand.

Later we’ll revisit scanning and parsing, in much greater detail.

(And we’ll use powerful, standard tools!)
Structure of Simple Example Compiler

infix expression

lexical analyzer (scanner)

syntax-directed translator

postfix expression

Syntax-directed translator combines parsing, semantic analysis and intermediate code generation.

To begin we’ll make things even simpler by assuming that the input has no whitespace and each token is represented by a single character

\[ + \quad - \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \]

so that scanning is trivial — just read the input character by character.

We begin with a grammar specifying the syntax of (our restricted notion of) infix expressions.

\[
\begin{align*}
expr & \rightarrow expr + digit \mid expr - digit \mid digit \\
digit & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

These rules are called productions.

The tokens, or terminal symbols, are

\[ + \quad - \quad 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \]

and the nonterminals are

\[ expr \quad digit \]

The start symbol is \( expr \).

The token strings that can be “derived” from the start symbol constitute the language defined by the grammar.

(We’ll make all this more precise later.)
Let’s draw a few parse trees for this grammar:

Translation scheme

Assuming that a parser could construct such parse trees using this grammar, it would not be difficult to synthesize a translation by embedding actions in the productions of the grammar, so that executing those actions during a depth-first traversal of the parse tree would produce the translation.
Modify the parse trees below to incorporate the actions we’ve added to the productions of the grammar:

```
expr      expr      expr
|         /\       /\
|         / \      / \
digit    expr + digit  expr + digit
| |           /\       \   
| |           / \  \    
| |           /   \      \  
1  digit  2  expr - digit  2
| |           |      |
| |           |      |
| |           |      |
1  digit  1  
|           |
|           |
3
```

Notice that executing the actions during a depth-first traversal indeed yields the postfix translation.

Next difficulty: How to implement a parser for this grammar?

For next time...

Continue reading Chapter 2, through 2.7.