Compiler Design

Satvik Gupta

March 1, 2023

Compiler is a program that can read a program in a *source* language and translate it into an equivalent program in the *target* language. A compiler must also report any errors in the source program that it detects while translating.

Interpreters are another kind of language processors that don't produce a target program. Instead, they perform line by line execution of the source program on the inputs that the user provides.

Compiler produced target programs are generally faster than performing the same execution using an interpreter.

Interpreters are better at error diagnostics because they execute the source statement by statement (hence they can catch run time errors as well).

Structure of a Compiler

Two main parts of a compiler:

1. **Analysis Part**
   - Breaks the source program into pieces.
   - Imposes a specific grammatical structure.
   - This structure is used to generate an intermediate representation of the source.
   - Informs user of any syntax or semantic errors in the source program.
   - Collects information about the source program and stores it in a data structure called *symbol table*.
   - The symbol table and the intermediate representation are passed to the synthesis part.
   - AKA front-end of the compiler.
2. **Synthesis Part**
   - Constructs target program from the intermediate representation and symbol table.
   - AKA Backend

Phases of Compiler

The compilation process can be broken down into the following phases.

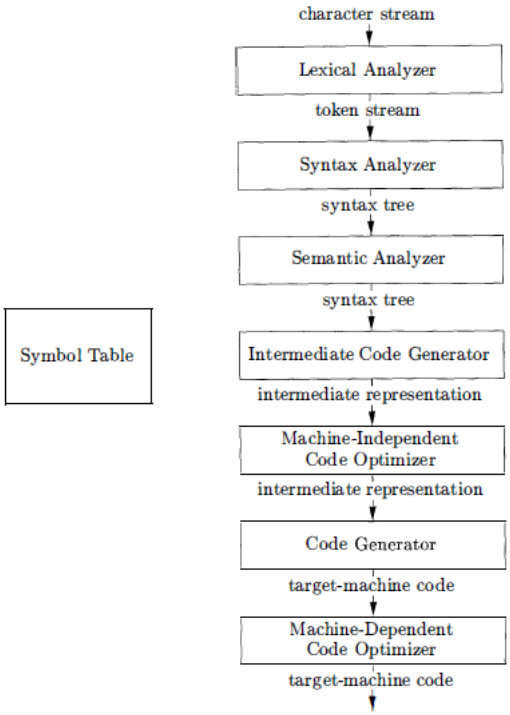

Figure 1: Phases of Compiler

- Phases may be grouped together in practice.
- Symbol table used by all phases.
- One or both of the optimization phases mentioned in the figure may be absent.

1. Lexical Analysis

AKA Scanning.

- The lexical analyzer reads the source code as *characters*.
- The characters are grouped into meaningful sequences called *lexemes*. For example, a variable name would consist of a group of characters - and be grouped into a single lexeme by the lexical analyzer.
- For each lexeme, the lex analyzer produces a *token*. Tokens take the form of a pair:

$$< token\_name, attribute\_value >$$

  Token name is an abstract name used during syntax analysis.

  Attribute value points to an entry in the symbol table for this token.

For example, let's taken the line:

```
position = initial+rate*60
```

Lex Analysis of this would be:

1. `position` would be a lexeme. It would be mapped into the token $< id, 1 >$.

   id stands for identifier, and is an abstract symbol. **1** points to the symbol table entry for `position`. The symbol table entry for an identifier holds information about that particular identifier, such as its name and type.

2. `=` is a lexeme that is mapped to a token <=>. No attribute value is necessary for this token. We could have used an abstract symbol here as well. For convenience we've taken =.

3. `initial` is a lexeme mapped to $< id, 2 >$. 2 points to symbol table entry for `initial`.
4. `+` is mapped to $< + >$
5. `rate` is mapped to $< id, 3 >$. 3 points to symbol table entry for `rate`.
6. `*` is mapped to $< * >$
7. 60 is mapped to $< 60 >$. We can also map it to $< number, 4 >$ and have 4 point to numerical value of 60.

The final output would be

$$< id, 1 >< = >< id, 2 >< + >< id, 3 >< * >< 60 >$$

2. Syntax Analysis

AKA Parsing.

- Reads tokens created by lex analysis.
- Converts them into a tree like representation.
- This representation shows the grammatical structure of the nodes.
- Syntax trees are generally used for this. In these, the interior node represents an operation and its children represent the arguments of that operation.

For our example above, the syntax tree would be:

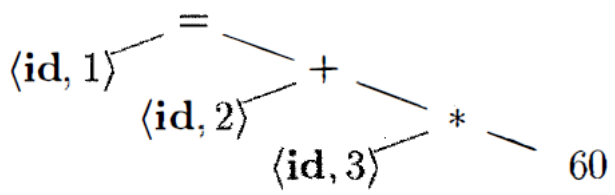

Figure 2: Syntax Tree

The tree depicts the ordering:

1. $< id, 3 >$ (`rate`) must be multiplied by 60 first.
2. The result of that multiplication must be added to $< id, 2 >$(`initial`).
3. The result of the addition should be assigned to $< id, 1 >$(`position`).

3. Semantic Analysis

- Semantic analysis uses the Syntax tree and symbol tables and checks for semantic consistency with the language definition.
- Saves type information in the syntax tree or in symbol table.

Type checking is an important part of semantic analysis.

Compiler must throw errors if types do not match, eg. if a floating point number is used to index an array.

Compiler may also perform type-conversions or coercions if the language permits it. For eg, if `+` is applied to a float and an integer, the compiler may convert/coerce the integer into a float.

For eg, let's assume `initial`,`position`, and `rate` are floats, and the lexeme 60 is an integer.

If the language permits it, the semantic analyzer can convert the 60 into a float.

The modified syntax tree will be as shown.

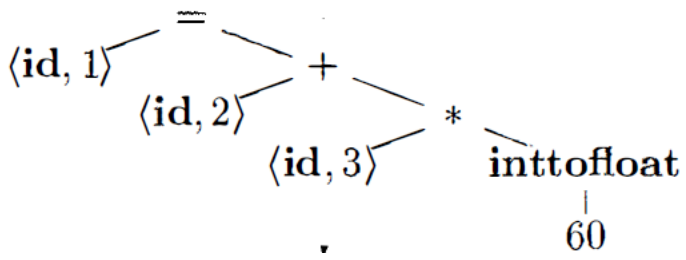

Figure 3: Modified Syntax Tree

4. Intermediate Code Generation

Compilers may have some intermediate code format that runs on an abstract machine. It is a low level, machine-like code.

It should be - easy to produce. - easy to translate to target machine.

For example, we can have a *3-address code* format, with a maximum of 3 operands per instruction. The RHS can have a maximum of 1 operator. This helps preserve the order of operations. Each operand can act as a register.

The intermediate code in this format, for our example:

```
t1=inttofloat(60)
t2=id3*t1
t3=id2+t2
id1=t3
```

5. Code Optimization

The intermediate code produced can be optimized. Optimization may be for shorter code, or less memory consuming code, or code that consumes less power.

In our above example we can optimize it as:

- `inttofloat(60)` can be performed just once at compile time
- `t3` is only used to transfer value into `id1`, it can be eliminated.

The final optimized code will be

```
t1=id3*60.0
id1=id2+t1
```

6. Code Generation

Finally we have to convert our intermediate code into machine/target language code. If it is machine code, register assignment is important.

Our example code can be converted to machine code as

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

- The `F` at the end of every instruction tells the machine we are dealing with floating point operations
- LDF loads the value of id3 into the register R2.
- MLF multiplies R2 with 60.0 and stores the value in R2. # is used to signify that 60.0 is a constant.
- ADDF adds R1 and R2 and stores the result in R1.
- STF finally stores the value in R1 to the location of id1.

Storage allocation for identifiers is not considered here, it depends on the language being compiled.

Symbol Tables

They contain variable names and attributes of the variables. Attributes may be

- name
- type
- scope
- (for procedures) the number and types of arguments, and method of passing the argument, return type.

Symbol Tables should have quick read and write access.

Passes

- Several phases can be grouped into one pass.
- One pass may be something that reads from an input file and writes to an output file, and have multiple phases in it.
- Front-end phases (lexical, syntax, semantic analysis, and intermediate code generation) may be one pass - that write an intermediate code to a file.
- Code Optimization can be an optional pass.
- Backend pass may be for producing code for a target machine.

Using passes, we can produce different compilers for different machines.

- Different source compilers for a single machine, by combining various front-ends with a single backend
- Compilers for different machines for a single language, by combining a front-end with various backends.