Compiler Design

Condensed Notes

May 6, 2025

Contents

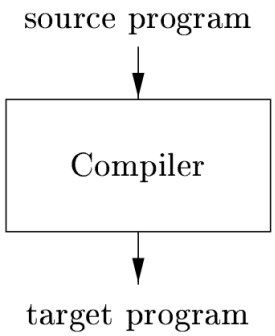
Chapter 1

Introduction

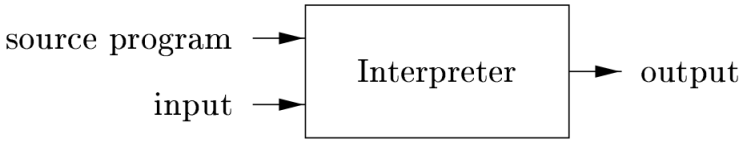
The software system that translates a source code to a form in which the computer can understand is called compilers.

1.1 Language Processors

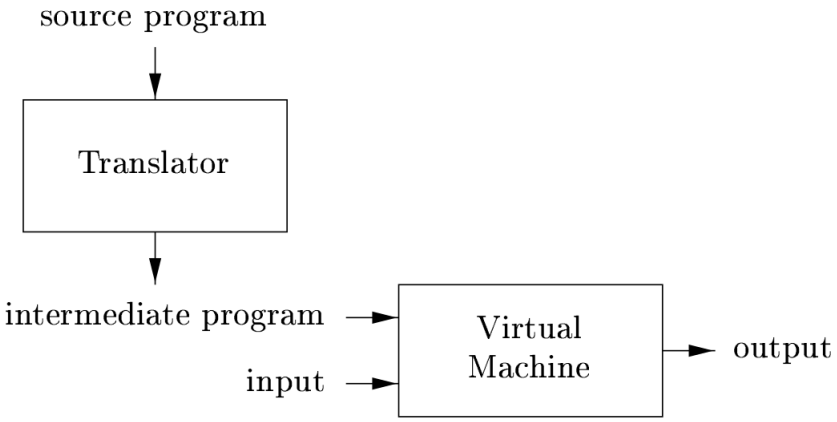
- **compilers**: a compiler is a program that can read a program in one lan- guage — the source language — and translate it into an equivalent program in another language — the target language.



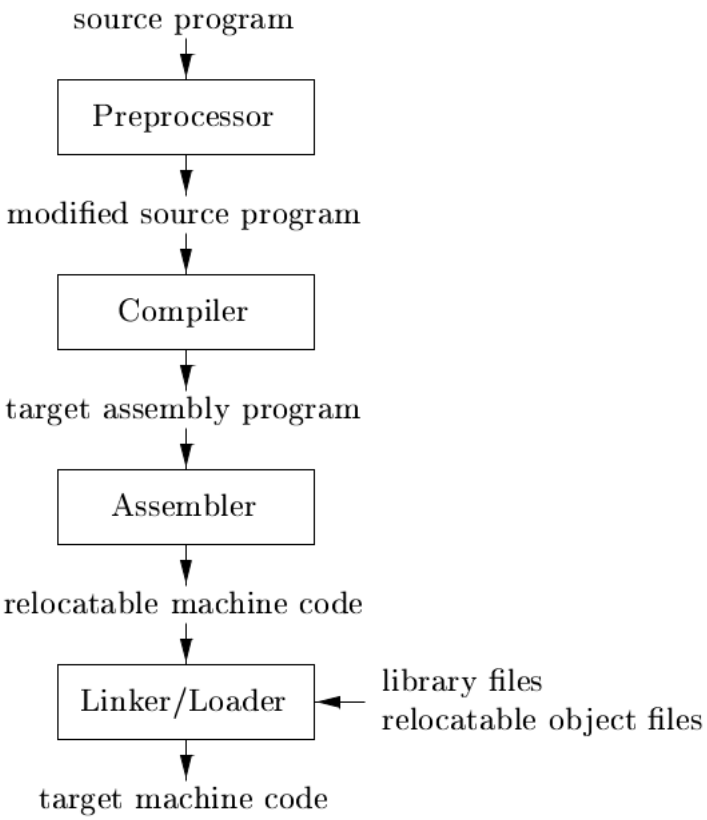
- **interpreter**:An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations speci ed in the source program on inputs supplied by the user.



- certain languages like java uses the hybrid of the above two classes it uses a compiler to compile a source code to a machine independent code which is interpreted to give the result.



- In general the different phases of compiler are as follows



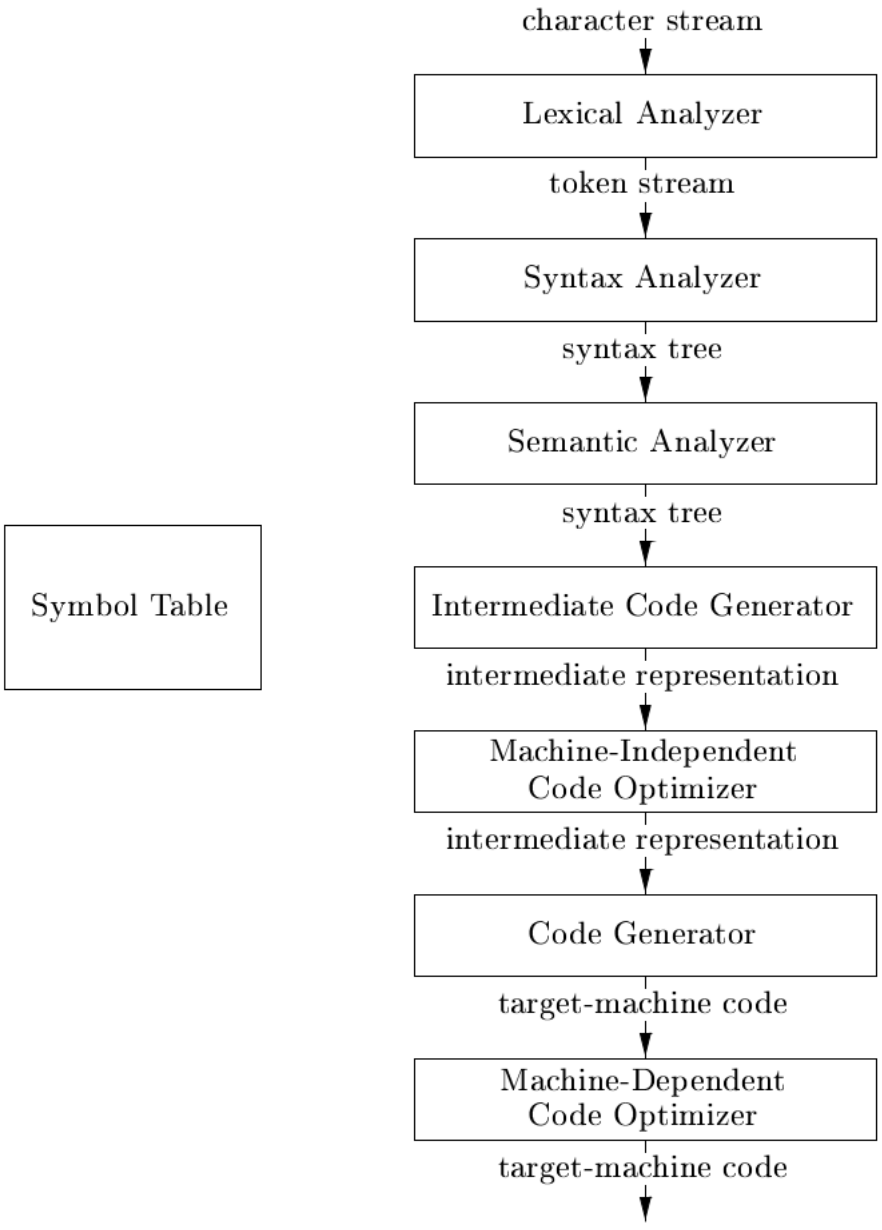
1.2 Structure of Compiler

There are two main parts of compiler -

- Analysis : The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages.The analysis part also collects information about the source program and stores it in a data structure called a symbol table , which is passed along with the intermediate representation to the synthesis part.

- Synthesis : The synthesis part constructs the desired target program from the interme- diate representation and the information in the symbol table.

Different phases of compiliation process:



In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed

explicitly.

The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise pro- duced from an unoptimized intermediate representation.

1.2.1 Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes . For each lexeme, the lexical analyzer produces as output a token of the form

⟨**token-name, attribute-value**⟩

token-name is an abstract symbol that is used during syntax analysis.

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

For **example**, suppose a source program contains the assignment statement

position = initial + rate * 60

after lexical analysis as the sequence of tokens will be

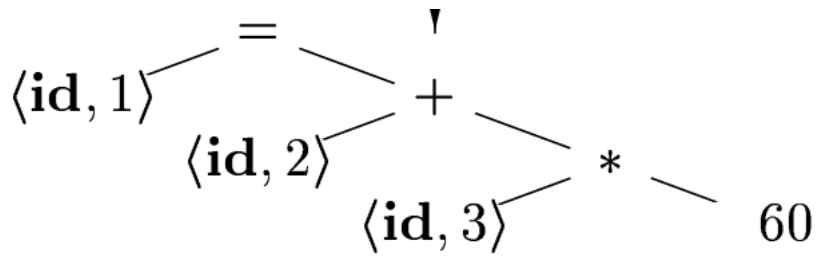
$\langle id, 1\rangle\langle=\rangle\langle id, 2\rangle\langle+\rangle\langle id, 3\rangle\langle*\rangle\langle 60\rangle$

In this representation, the token names =, +, and * are abstract symbols for the assignment, addition, and multiplication operators

1.2.2 Syntax Analysis

The second phase of the compiler is syntax analysis or parsing . The parser uses the rst components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

The above code will be represented a follows



This representation takes into account the precedence of operators.

1.2.3 Semantic Analysis

Does the following from the syntax tree

- Overall Semantics : check the overall grammar of the program.
- Type Checking : ensures the type of operand and operators are same.
- Coercions : if type is missmatche and if possible typecast the mismathced lexeme.

1.2.4 Intermediate Code Generator

It generates an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

1.2.5 Code Optimizer

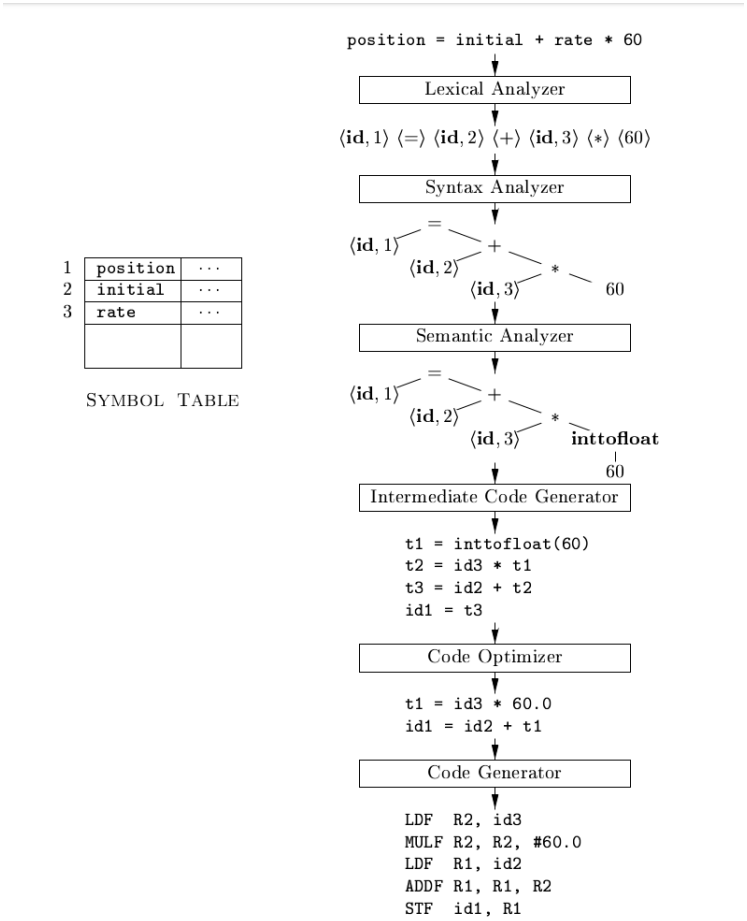
The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

1.2.6 Code Generator

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

1.2.7 Summary

The following figure is summary of different phases of compiliation



1.3 Basics of Programming language

1.3.1 Static vs Dynamic Policy

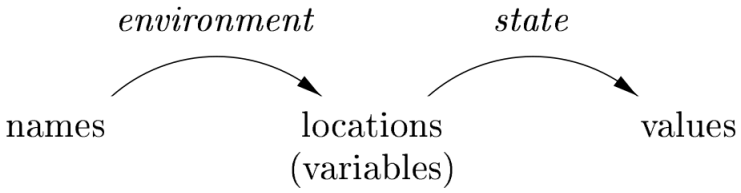
if a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a static policy or that the issue can be decided at compile time .

A policy that only allows a decision to be made when we execute the program is said to be a dynamic policy or to require a decision at run time .

The scope of declaration of a variable is an issue where we encounter such policies A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses dynamic scope .

1.3.2 Environments and States

In order to give way to the Scope system of variables whereby a variable decleared with same name but in different scopes would refer to different values in the memory(the store), a two stage mapping is used in which the names of the variable is dynamically mapped to the location of the memory (l-value) the this l-value is dynamically mapped to its r-value as shown below.



To summarise

- The *environment* is a mapping from names to locations in the store. Since variables refer to locations (l-values" in the terminology of C), we could alternatively de ne an environment as a mapping from names to variables. Thus environment may be dynamically changed when scope of a variable with same name changes while programme execution.

- The *state* is a mapping from locations in store to their values. That is, the state maps l-values to their corresponding r-values, in the terminology of C.Thus state may be dynamically changed when the value of a variable is changed while programme execution.

- Not always these mapping is done dynamically the compiler may decide in certain cases to optimize the code and map them statically but mostly dynamic mapping is used. eg. #define MAX 1000

Chapter 2

A Simple Syntax-Directed Translator

2.1 Introduction

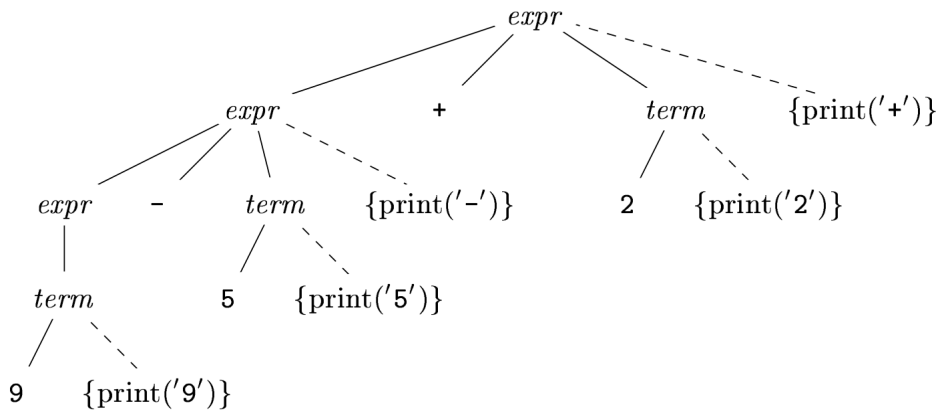
A syntax directed translator as it sounds simply translates a language to intermediate code from the syntax rules, which is usually in the form of context free grammers. At this point we are not bothered about the simantics of the language.

2.2 Syntax-Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar as shown below.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1$ + $term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1$ - $term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow$ 0	$term.t = '0'$
$term \rightarrow$ 1	$term.t = '1'$
...	...
$term \rightarrow$ 9	$term.t = '9'$

Attributes : these are special field (sort of variables) used to hold certain values while translation, example the 't' attribute in all the rules above.An alternative approach is to not use memory in the form of attributes but rather execute programme fragments as shown below

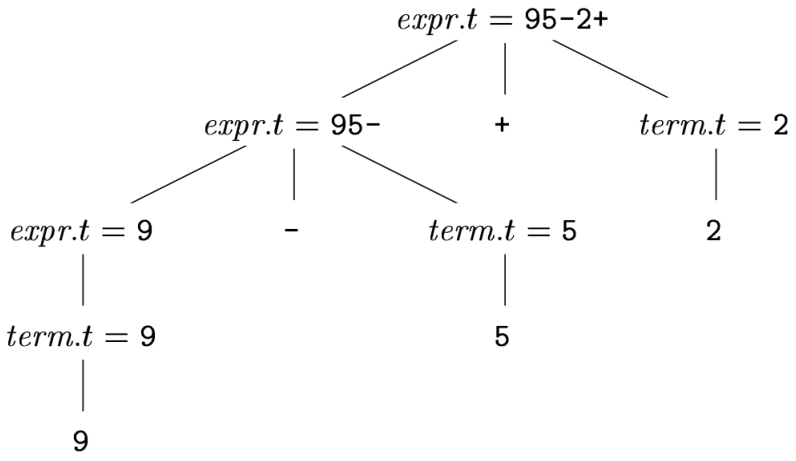


This will print the translation on tree traversal (in this case postfix expression).

(Syntax-directed) translation schemes:A translation scheme is a notation for attaching program fragments to the productions of a grammar. The program fragments are executed when the production is used during syn- tax analysis. The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

Synthesized Attributes:An attribute is said to be synthesized if its value at a parse-tree node N is de- termined from attribute values at the children of N and at N itself.

annotated parse tree:A parse tree showing the attribute values at each node is called an annotated parse tree as shown below.



Simple Syntax-Directed Definitions: A syntax directed definitions consists of CFG along with some symantic rules, if the order of terms in productions and symantic are same irrespective of additional characters then such definitions are called simple. example:

PRODUCTION	SEMANTIC RULE
$expr \rightarrow expr_1$ + $term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$

These enhancements over CFG makes translations easier.

Chapter 3

Omissions-Aho Lam

1.2.7 to 1.5,1.6.3 to 1.6.7