Introduction

1 Languages

1.1 Natural Languages

Natural languages have:

- Words
- Types of words
- Syntax
- Meaning of words (semantics)

Natural languages are useful for communication between humans, but only if they speak the same language, otherwise we need translation

1.2 Programming Languages

Communication between humans and computers can't be done by default as a human speaks a natural language and a computer "speaks" machine languages.

The solution to this:

- Using a programming language
- Understandable by humans
- But much more structured than natural languages
- But still very far from being "machine" language

2 Compilers

We need a way to write something in some programming language and "translate" it into executable machine language

Compiler: a computer program that transforms:

- A program, written in a programming language (source program/source language)
- To an equivalent program in another language (target program/target language)

Any language can be a target language.

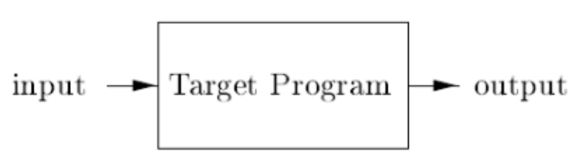
The most common target program is an executable machine-language program

3 Compilers vs Interpreters

3.1 Compilers

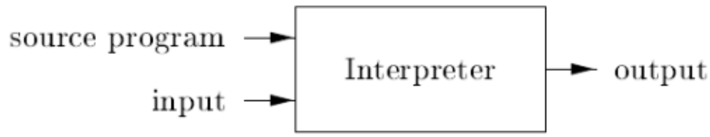
After compilation, the target program:

- Is called by the user
- Processes input and produces output



3.2 Interpreters

Directly executes the operations specified in the source program on inputs supplied by the user



3.3 Comparison

While mapping input to output:

- The machine-language target program (built by a compiler) is faster
- An interpreter gives better error diagnostics (it executes the source program statement by statement)

4 Compilers

Compilers and interpreters must:

- Detect lexical/syntactic/semantic inconsistencies
- Propose solutions wherever possible

4.1 Structure of a compiler

Analysis part (front end)

- Breaks the source program into constituent pieces
- Imposes grammatical structure on them (according to the rules of the source language)
- Creates intermediate representation (IR) of the source program
- Provides informative error messages (if errors detected)
- Collects all the important information in a symbol table and passes the symbol table and IR along to the synthesis part

Synthesis part (back end)

- Uses the symbol table and the IR
- Generates and optimises the target code

Main phases of the analysis part

- Lexical analysis (scanning)
- Syntax analysis (parsing)
- Semantic analysis

Main phases of the synthesis part

- Target code generation
- Target code optimization

4.2 Strict separation

Why separate analysis and synthesis parts?

- Can combine different analysis/synthesis parts in a modular way
- Create new compilers
- For L languages and M machines
  - We need only L+M modules instead of $L \times M$

5 Internal phases of a compiler

5.1 Lexical analysis/scanning

- Reads the input program as a character stream
- Groups the characters into meaningful sequences (lexemes)

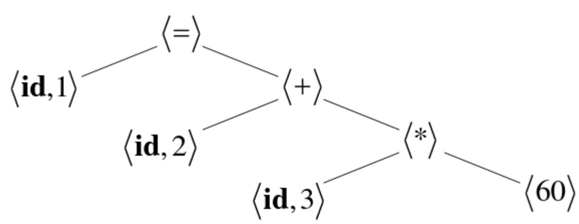
position=initial + rate * 60

- For each lexeme creates a tone (acts as one single symbol)

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

5.2 Syntax analysis/Parsing

- Uses the tokens from the lexical analysis
- Creates a tree like intermediate representation of the source program
- Syntax tree/parse tree
  - Interior nodes: operations
  - Children nodes: arguments of the operations



5.3 After syntax analysis

All the subsequent phases use:

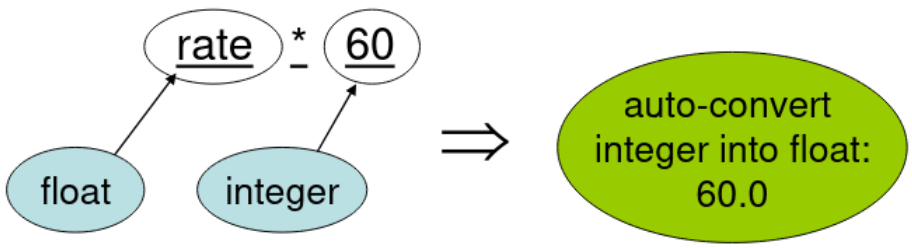
- The symbol table (i.e. all the parts of the source program)
- The syntax tree (i.e. the grammatical structure of the program)

In order to:

- Analyse further the program
- Produce the target code

5.4 Semantic analysis

- Uses the symbol table and syntax tree
- Checks for semantic consistency with the source language definition
- Stores the semantic information in the symbol table or the syntax tree (for further use)
- Important part is type checking and automatic type conversion (coercion)



5.5 Intermediate code generation

- Explicit low-level machine-like code
- It must be easy to both produce and to translate into target machine code
- Usually a tree address code
  - Simple instructions
  - Three operands per instruction

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

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

5.6 Intermediate code optimization

- Improve the intermediate code into a "better" code
- Better can be
  - Faster
  - Shorter
  - Consume less power

```
 t1 = inttofloat(60)
 t2 = id3 * t1              t2 = id3 * 60.0
 t3 = id2 + t2      →      id1 = id2 + t2
id1 = t3
```

6 Symbol table management

Store in symbol table (except variable names):

- Attributes with additional information, e.g.:
  - Type
  - Storage address
  - Scope (where in the program the value is used)
- Also, for the case of procedure names:
  - Number and types of its arguments
  - Method of passing each argument (value or reference)

7 Passes

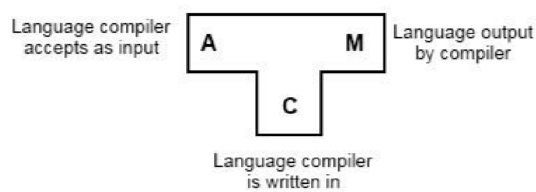
In practice, phases of the compiler may be ”grouped” together in passes

8 Evolution of compilers

Generations of programming languages

- 1st generation - 0s and 1s
- 2nd generation - assembly languages
- 3rd generation - higher level (Java, C)
- 4th generation - specific applications (SQL)

Can use previous compilers for new languages
T diagrams: a set of ”puzzle pieces”



Recursive use of compilers (and T diagrams):

- 1st compiler written in S translating A to T
- 2nd compiler written in T translating S to T
- Third compiler written in T translating A to T