COMP124

Computer Systems

24 | Code Translation | Grammars | EBNF

Dr Stuart Thomason

Code Translation

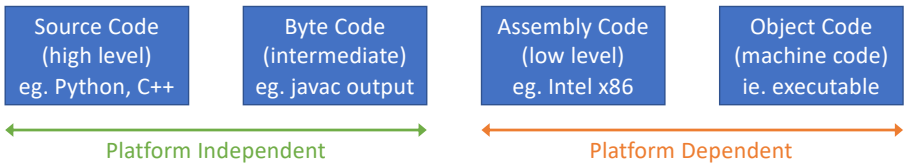
- Translation is the process of turning source code into object code
  - Source code is high level (statements and structure understood by humans)
  - Object code uses a specific instruction set and registers (understood by the CPU)

- Various special programs are involved in code translation
  - Compiler – Translates source code and stores resulting object code in a file
  - Interpreter – Translates source code and performs its actions directly
  - Assembler – Translates low level mnemonics into object code stored in a file
  - Linker – Combines object code files and library code into a final executable image

- Older languages (eg. C, C++) use a compiler and linker in the traditional manner
- Newer languages (eg. Java, Python) use both a compiler and interpreter together

Types of Code

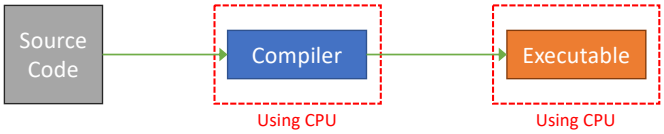
- There are different types of code depending on the stage of compilation



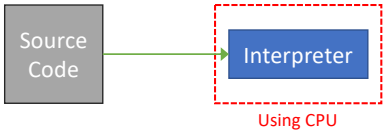
- Source code is portable (platform independent)
  - Write the code once as a simple text file
  - Then use a compiler that targets a specific platform
  - Compiler outputs assembly code or object code that will only run on the platform for which it was compiled (platform dependent)
  - You can use the same source code with another compiler on a different platform (eg. write a game in C++, then compile it for PC, XBox, PlayStation, Switch, etc.)

Compilers vs. Interpreters

- Compilers…
  - Take a source program and translate it as a whole into machine code
  - Translation and execution phases are separate



- Interpreters…
  - Take one line of code at a time, determine its meaning, then carry out that action
  - Translation and execution phases are interlaced (interpreter is the executable)
  - No stand-alone executable code is generated



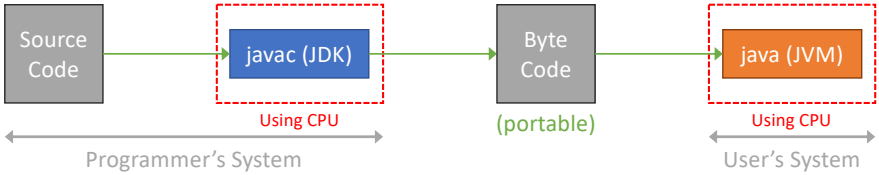
Comparing Translation Methods

- Compilers and interpreters behave in different ways
- Each method has advantages and disadvantages

Compilers	Interpreters
Translate entire file (with context)	Translate line by line (no look ahead)
Compilation process is fast	Interpretation is slow
Compiled code runs much faster	Interpreted code runs much slower
Run once (to make executable code)	Run every time source file is 'executed'
Check for syntax and semantic errors	Can only check for syntax errors
Programming tools can be complex	Programming environment is easier
Syntax errors hard to locate by line	Syntax errors easier to identify in code
Need careful organisation of source code	Useful for rapid prototyping
Debugging can be complicated	Debugging tools are easier to use
Generated object code is platform locked	Source code is portable across platforms

Javac & Java

- Java uses both a compiler (javac) and interpreter (java)
  - Java development kit (JDK) must be installed on the programmer's system
  - Source code is compiled into intermediate byte code
  - Byte code is portable (platform independent)
  - Java virtual machine (JVM) is an interpreter that processes the byte code
  - Java runtime environment (JRE) must be installed on user's system



- Some other languages use a similar approach but it all happens on the user's system
  - Python compiles source code into byte code and then immediately interprets it
  - Happens without the user knowing the byte code exists (might see .pyc files)

Interpreters and Regression

- From the Wikipedia page about interpreters…

  "Interpretation cannot be used as the sole method of execution. Even though an interpreter can itself be interpreted and so on, a directly executed program is needed somewhere at the bottom"

- New programming languages are regularly created to solve niche problems
  - Easier to design an interpreter and easier to code in an interpreted language
  - So newer languages tend to be interpreted

- But there will always be a need for compilers and people who know how they work
  - The interpreter itself needs to run as an executable program
  - A compiled language will be needed to write the code for the interpreter
  - Even if the interpreter is written in an interpreted language, a compiler will be needed to make the interpreter for that language

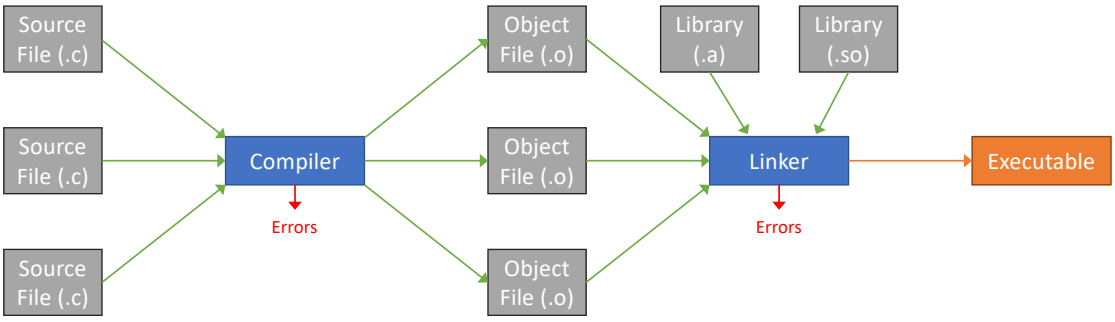
Types of Compiler

- Early compilers were single pass
  - Moves through the source code line by line and generates object code on the way
  - Faster and uses less memory
  - But cannot perform any optimisations
  - Requires variables (and other symbols) to be declared before use

- Modern compilers are multi pass
  - Reads the entire source file into an internal data structure
  - Analyses and augments the data structure in multiple stages
  - Slower and requires more memory
  - But can look at the entire code to see where optimisations can be made
  - Allows programmer to use (eg. on line 10) something defined later (eg. on line 30)
  - Final pass reads the internal data structure and generates object code

Linking

- Can be many source files each with a corresponding object file
- Source files also reference library code which exists in multiple object files
- The linker pulls all these together into the final executable program image



- Recall from a previous lecture the difference between static and dynamic linking

Components of Programming Languages

- Lexical component
  - List of all legal keywords allowed in the language (eg. import, public, else, while)
  - Meaning of each individual keyword (the action or effect it has)

- Syntactical component
  - Defines the form and structure of legal expressions (lines of code) in the language
  - For example, double private x; is not legal, whereas private double x; is legal

- Semantic component
  - Adds meaning to an expression made from keywords and syntax structures
  - Defines what should be done when executing a particular fragment of code
  - For example, the action to be taken when the code is an if-else fragment (test the condition, execute the if part when true, or the else part when false)

Language Grammars

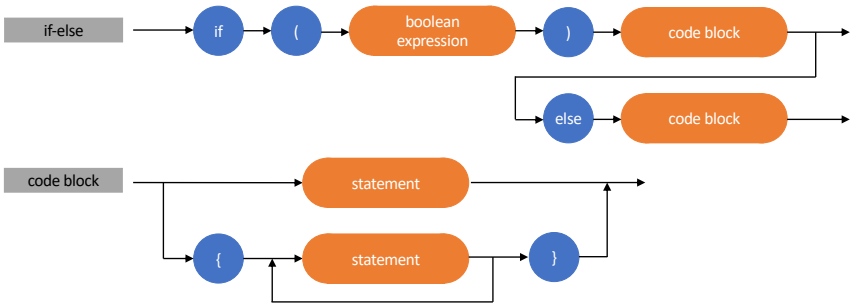
- A precise description of the programming language is needed so the translator can…
  - Analyse the source code for correctness
  - Apply optimisations where possible
  - Generate correct and unambiguous object code

- A grammar is a set of formal syntax rules
  - Complex rules are defined in terms of other rules
  - Simple rules are defined in terms of elementary symbols or keywords

- A source file is just a plain text file containing characters
  - Translator groups characters into tokens that represent keywords, brackets, etc.
  - A token is the smallest possible group of characters that has a valid meaning
  - Grammar rules define the correct order of tokens

Syntax Diagrams

- Grammar rules can be drawn as a set of diagrams (easier for a human to understand)



- Expressions are formed from one or more tokens
  - Some expressions are non-terminal (defined by other rules)
  - Other expressions are terminal (defined in their final form with simple characters)

Extended Backus-Naur Form

- The translator uses a more precise grammar called extended Backus-Naur form (EBNF)
- Rules are defined as productions of non-terminal and terminal expressions

```
<if-else>     → if<lbracket><expression><rbracket><code-block>
                [else<code-block>]

<code-block> → <statement>|<lbrace><statement><rbrace>
```

- Rules use some normal characters as part of their format
  - For example, [has a special meaning in the grammar rules
  - If we want to match a left square bracket we would use <lsbracket> (or similar)

- Every aspect of the language must be defined in this way, so there will be hundreds of rules for any reasonably complex programming language (Java has around 120)

EBNF Notation

- We can define grammar rules to match valid variable names in a language

```
<varname> → <letter>{<letter>|<digit>}
<letter>  → a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>   → 0|1|2|3|4|5|6|7|8|9
```

→	Defines a rule
|	OR
<name>	Non-terminal symbol
symbol	Terminal symbol
[tokens]	Optional tokens
{tokens}	Tokens repeated 0 or more times
*	Tokens repeated 0 or more times (suffix)
+	Tokens repeated 1 or more times (suffix)

- Another example using the suffix notation…

```
<integer> → <digit>+
```

EBNF Matching

- Rules written with EBNF provide an unambiguous definition of the entire language

```
<varname> → <letter>{<letter>|<digit>}
<letter>  → a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>   → 0|1|2|3|4|5|6|7|8|9
```

- In English we might say: “a variable name must start with a letter, followed by any quantity of letters and digits in any combination”
  - Valid: zbc99, a9fg28p
  - Invalid: 7gbh8, 89de4

- A compiler will try to match the source code against the EBNF grammar of the language
  - High level code uses whitespace, brackets and semicolons to help provide structure
  - Syntax errors generated if compiler cannot match every part of the source code
  - Error messages often don’t point to the exact line where the problem happened