Concepts of Programming Languages, CSCI 305, Fall 2021
Introduction to Programming Languages, Chapter 1, Aug. 30
Chapter 1 Introduction, 5-37

Example Language Tradeoffs:

- Support for assertions, some have, others don’t
- Short circuit evaluation, some have, others don’t
- Type coercion, explicit type coercion or implicit type coercion
- Make language be statically, dynamically typed, duck typing
- Scope can be static or dynamic
- Many ways to pass parameters: call-by-value, call-by-reference, call-by-name

Translating Programs

Assemblers

Assembly language to machine code

Compilers

One language to another (machine code, assembly, P-code, C, …)

Preprocessor

- Example tasks: remove comments, whitespace, expand abbreviations in the style of a macro assembler. Can have preprocessors that group characters into tokens.
- In general, preprocessors processes input data to output, and the output is used as input to another program

Modern languages use a combination of the above

Java

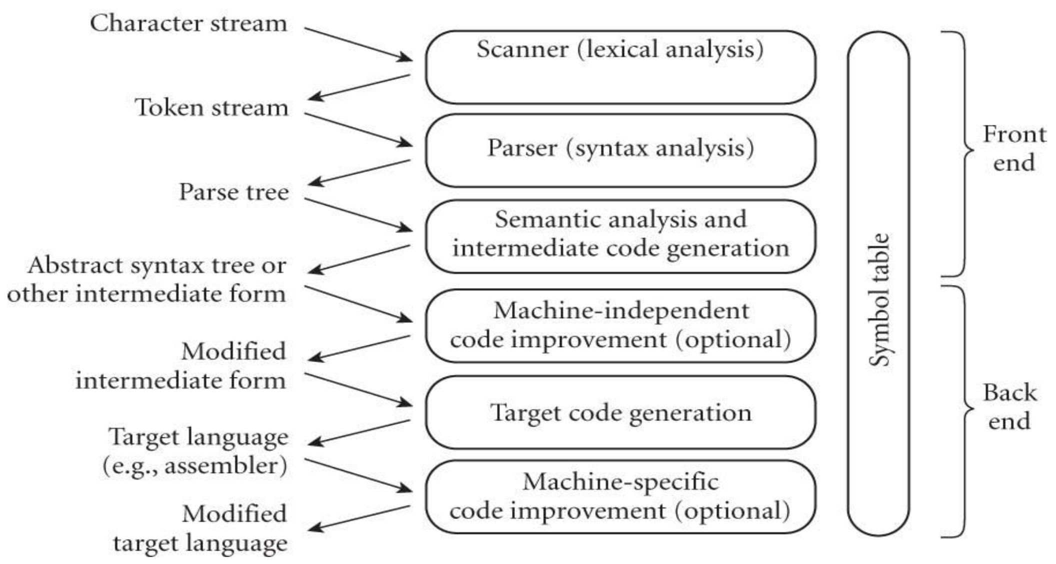
- Java compiles to byte code which runs on a Java virtual machine
- Running on the virtual machine was interpreted but now uses just-in-time compiling

C#

- C# compiles to .NET Common Intermediate Language
- CIL was interpreted but now uses just-in-time
- Over 15 languages compile to CIL - Visual Basic, J# (Microsoft Java), A# (Microsoft Ada), versions of C++, Python, Lisp, Ruby

Syntax, semantics, language translation

Phases of compilation (Figure 1.3, page 27):


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Target language (e.g., assembler)
Machine-specific code improvement (optional)
Modified target language
Symbol table
Front end
Back end

Front End – Figure out the meaning of a program

Lexical analysis

Using regular expressions, splits a character stream into a token stream.

Syntactic analysis

Using context-free grammar, builds a parse tree, sometimes called a concrete syntax tree, from the tokens and starts building a symbol table

Semantic analysis

Continue building the symbol table, reduce the complicated parse tree to an abstract syntax tree, and annotate the tree with useful information such as pointers into the symbol table

Back End – Construct an equivalent target program

Target code generation

Using the abstract syntax tree, generate code into the target language

Machine-specific code improvement

Code optimization. Some improvements were machine independent so could be made after semantic analysis. Other improvements are machine dependent.