CMPE 152: Compiler Design

August 29 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Basic Info

- Office hours
  - TuTh 3:00 – 4:00 PM
  - ENG 250
- Website
  - Faculty webpage: http://www.cs.sjsu.edu/~mak/
  - Class webpage: http://www.cs.sjsu.edu/~mak/
  - Syllabus
  - Assignments
  - Lecture notes

Permission Codes?

- If you need a permission code to enroll in this class, see the department's instructions at https://cmpe.sjsu.edu/content/Undergraduate-Permission-Number-Requests
- Complete the Google form at https://docs.google.com/a/sjsu.edu/forms/d/e/1FAIpQLSe9YgAea-QsgLZof-KIMmuQthoChL4micudyRukgWneiByN2A/viewform

A Compiler is a Translator

- A compiler translates a program that you've written
- ... in a high-level language
  - C, C++, Java, Pascal, etc.
- ... into a low-level language
  - assembly language or machine language
- ... that a computer can understand and eventually execute.

More Definitions

- source program: the program (application) that you write in a high-level language which the compiler will translate
  - Usually stored in a source file.

- source language: the high-level language in which you write your source program
  - Example: Pascal

More Definitions, *cont'd*

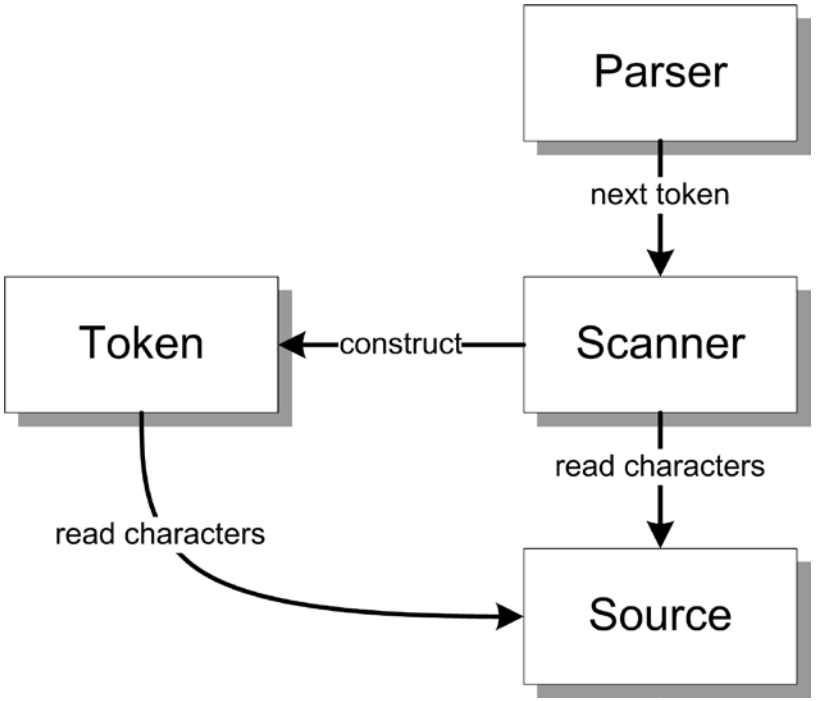
- object language: the low-level language (AKA target language) into which the compiler translates the source program
  - Do not confuse *object language* with *object-oriented language*.
  - Example: Jasmin assembly language
  - Example: Intel machine code
- object program: your program after it has been translated into the object language

More Definitions, *cont'd*

- target machine: the computer that will eventually execute the object program
  - Example: Your laptop's hardware
  - Example: The Java Virtual Machine (JVM)
    - The JVM runs on your workstation or laptop (any computer that supports Java)
- implementation language: the language that the compiler itself is written in
  - Example: Java

Take roll!

Conceptual Design (Version 1)



- Parser
  - Controls the translation process.
  - Repeatedly asks the scanner for the next token.
- Scanner
  - Repeatedly reads characters from the source to construct tokens for the parser.
- Token
  - A source language element
    - identifier (name)
    - number
    - special symbol (+ - * / = etc.)
    - reserved word
  - Also reads from the source
- Source
  - The source program

Token

- A low-level element of the source language.
  - AKA lexeme
- Pascal language tokens
  - Identifiers
    - names of variables, types, procedures, functions, enumeration values, etc.
  - Numbers
    - integer and real (floating-point)
  - Reserved words
    - **`BEGIN END IF THEN ELSE AND OR NOT`** etc.
  - Special symbols
    - **`+ - * / := < <= = >= > . , .. : ( ) [ ] { } '`**

Parser

- Controls the translation process.
  - Repeatedly asks the scanner for the next token.
- Knows the syntax ("grammar") of the source language's statements and expressions.
  - Analyzes the sequence of tokens to determine what kind of statement or expression it is translating.
  - Verifies that what it's seeing is syntactically correct.
  - Flags any syntax errors that it finds and attempts to recover from them.

Parser, *cont'd*

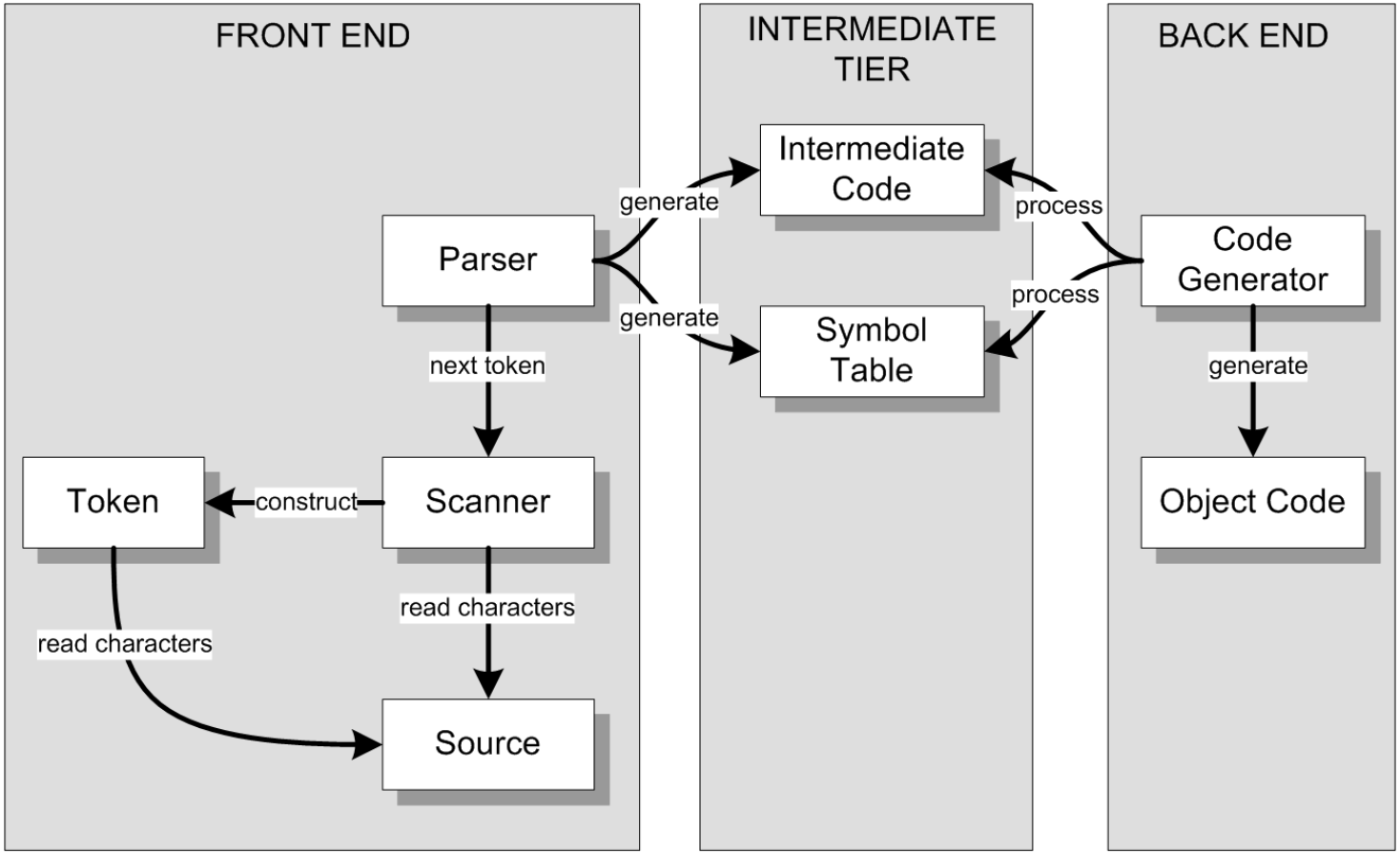
- What the parser does is called parsing.
  - It *parses* the source program in order to translate it.
  - AKA syntax analyzer

Scanner

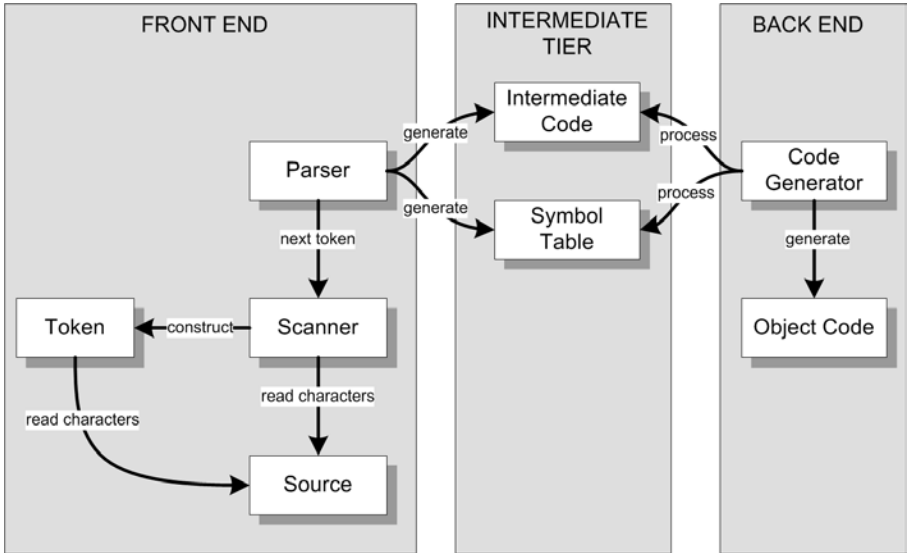
- Reads characters sequentially from the source in order to construct and return the next token whenever requested by the parser.
- Knows the syntax of the source language's tokens.
- What the scanner does is called scanning.
  - It *scans* the source program in order to extract tokens.
  - AKA lexical analyzer

Conceptual Design (Version 2)

- We can architect a compiler with three major parts:



Major Parts of a Compiler



- Front end
  - Parser, Scanner, Source, Token
- Intermediate tier
  - Intermediate code (icode)
    - "Predigested" form of the source code that the back end can process efficiently.
    - Example: parse trees
    - AKA intermediate representation (IR)
  - Symbol table (symtab)
    - Stores information about the symbols (such as the identifiers) contained in the source program.
- Back end
  - Code generator
    - Processes the icode and the symtab in order to generate the object code.

Only the front end needs to be source language-specific.

The intermediate tier and the back end can be language-independent!

What Else Can Compilers Do?

- Compilers allow you to program in a high-level language and think about your algorithms, not about machine architecture.
- Compilers provide language portability.
  - You can run your C++ and Java programs on different machines because their compilers enforce language standards.

What Else Can Compilers Do? *cont'd*

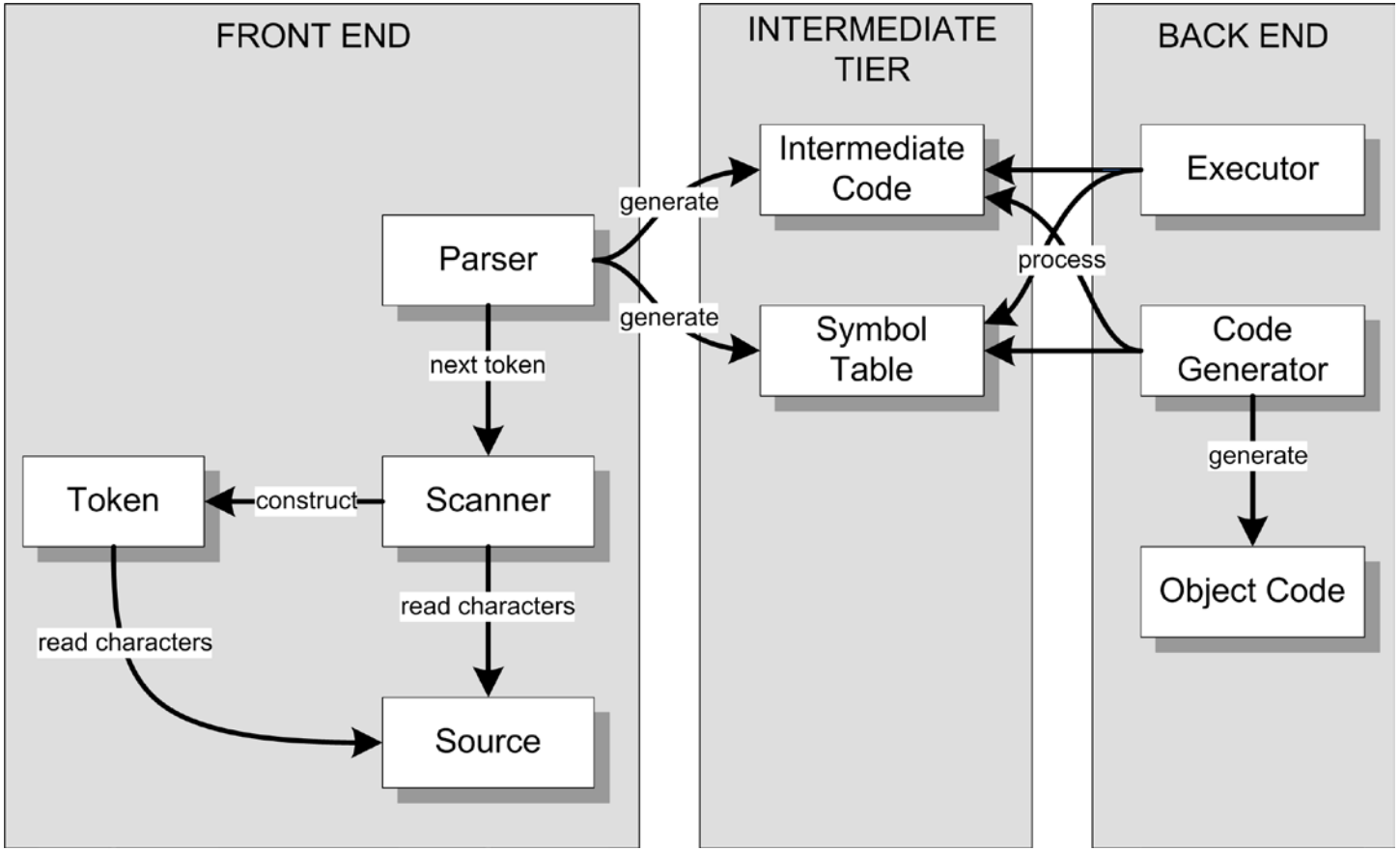
- Compilers can optimize and improve the execution of your programs.
  - Optimize the object code for speed.
  - Optimize the object code for size.
  - Optimize the object code for power consumption.

What about Interpreters?

- An interpreter executes a source program instead of generating object code.
- It executes a source program using the intermediate code and the symbol table.

Demo

Conceptual Design (Version 3)



- A compiler and an interpreter can both use the same front end and intermediate tier.

Comparing Compilers and Interpreters

- A compiler generates object code, but an interpreter does not.
- Executing the source program from object code can be several orders of magnitude faster than executing the program by interpreting the intermediate code and the symbol table.
- But an interpreter requires less effort to get a source program to execute
  - ➔ faster turnaround time

Demo

Comparing Compilers and Interpreters, *cont'd*

- An interpreter maintains control of the source program's execution.
- Interpreters often come with interactive source-level debuggers that allow you to refer to source program elements, such as variable names.
  - AKA symbolic debugger

Comparing Compilers and Interpreters, *cont'd*

- Therefore ...
  - Interpreters are useful during program development.
  - Compilers are useful to run released programs in a production environment.
- In this course, you will ...
  - Modify an interpreter for the Pascal language.
  - Develop a compiler for a language of your choice.
  - You can invent your own programming language!

Key Steps for Success

- Whenever you develop a complex program such as a compiler or an interpreter, key first steps for success are:

  1. Design and implement a proper framework.
  2. Develop initial components that are well-integrated with the framework and with each other.
  3. Test the framework and the component integration by running simple end-to-end tests.

- Early component integration is critical, even if the initial components are greatly simplified and don't do very much.

Key Steps for Success, *cont'd*

- Test your framework and components and get them working together as early as possible.
- The framework and the initial components then form the basis upon which you can do further development.
- You should always be building on code that already works.