Programming Languages and Compilers

Qing Yi

class web site:
www.cs.utsa.edu/~qingyi/cs5363

A little about myself

- ***Qing Yi***
- *Ph.D.* Rice University, USA.
- Assistant Professor, Department of Computer Science
- **Office: SB 4.01.30**
- **Phone : 458-5671**

- ***Research Interests***
- Compilers and software development tools
  program analysis&optimization for high-performance computing
- Programming languages
  type systems, different programming paradigms
- Software engineering
  systematic error-discovery and verification of software

General Information

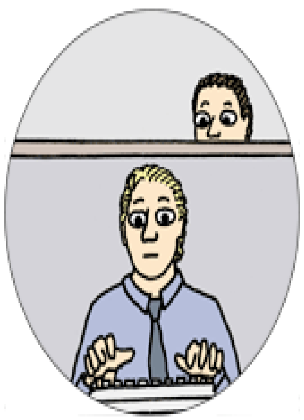
- Class website
  - www.cs.utsa.edu/~qingyi/cs5363
    - Check for class handouts and announcements
- Office hours: Mon 4-5pm and 7-8pm; by appointment
- Textbook and reference book
  - Engineering a Compiler
    - Second Edition. By Keith Cooper and Linda Torczon. Morgan-Kaufmann. 2011.
  - Programming Language Pragmatics,
    - by Michael Scott, Second Edition, Morgan Kaufmann Publishers, 2006
- Prerequisites
  - C/C++/Java programming
  - Basic understanding of algorithms and computer architecture
- Grading
  - Exams (midterm and final): 50%;
  - Projects: 25%; Homeworks: 20%;
  - Problem solving (challenging problems of the week): 5%

Outline

- Implementation of programming languages
  - Compilation vs. interpretation
- Programming paradigms (beyond the textbook)
  - Functional, imperative, and object-oriented programming
  - What are the differences?
- The structure of a compiler
  - Front end (parsing), mid end (optimization), and back end (code generation)
- Focus of class
  - Language implementation instead of design
  - Compilation instead of interpretation
    - Algorithms analyzing properties of application programs
    - Optimizations that make your code run faster

Programming languages

- Interface for problem solving using computers
  - Express data structures and algorithms
  - Instruct machines what to do
  - Communicate between computers and programmers



```
……………...
c = a * a;
b = c + b;
…………….
```

High-level (human-level) programming languages

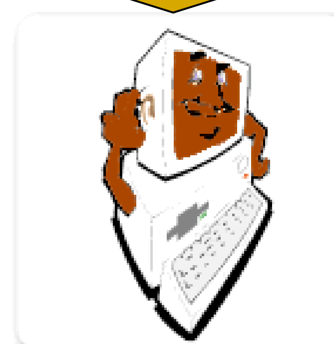
Easier to program and maintain
Portable to different machines

```
……….
00000
01010
11110
01010
……….
```

Low-level (machine-level) programming languages

Better machine efficiency

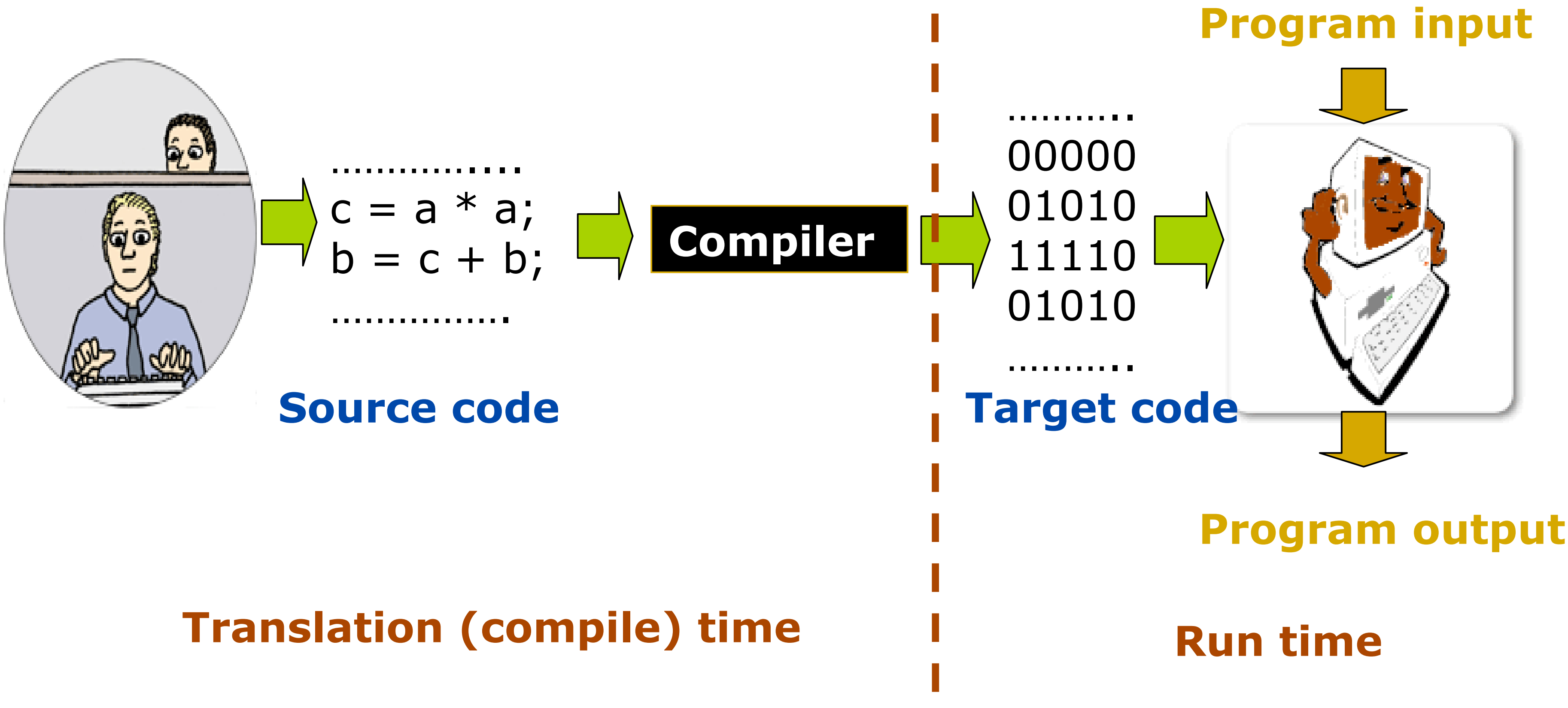
Program input



Program output

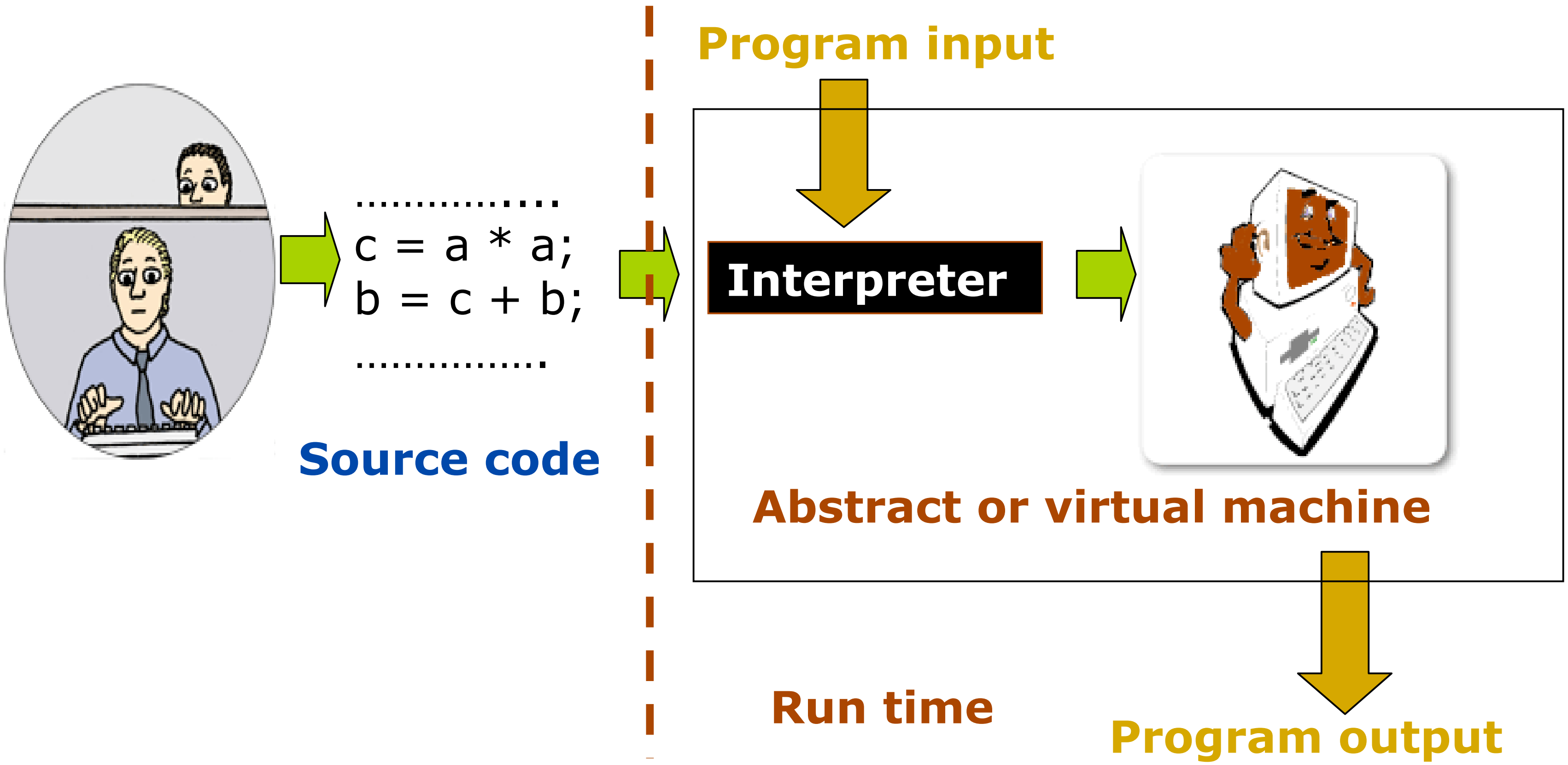
Language Implementation Compilers

- Translate programming languages to machine languages
- Translate one programming language to another



Language Implementation
Interpreters

- Interpret the meaning of programs and perform the operations accordingly



Compilers and Interpreters Efficiency vs. Flexibility

- Compilers
  - Translation time is separate from execution time
    - ✓ Compiled code can run many times
    - ✓ Heavy weight optimizations are affordable
    - ✓ Can pre-examine programs for errors
    - x Static analysis has limited capability
    - x Cannot change programs on the fly
- Interpreters
  - Translation time is included in execution time
    - x Re-interpret every expression at run time
    - x Cannot afford heavy-weight optimizations
    - x Discover errors only when they occur at run time
    - ✓ Have full knowledge of program behavior
    - ✓ Can dynamically change program behavior

Programming Paradigms

- Functional programming: evaluation of expressions and functions
  - Compute new values instead of modifying existing ones (disallow modification of compound data structures)
  - Treat functions as first-class objects (can return functions as results, nest functions inside each other)
  - Mostly interpreted and used for project prototyping (Lisp, Scheme, ML, Haskell, …)
- Imperative programming: express side-effects of statements
  - Emphasize machine efficiency (Fortran, C, Pascal, Algol,…)
- Object-oriented programming: modular program organization
  - Combined data and function abstractions
  - Separate interface and implementation
  - Support subtype polymorphism and inheritance
  - Simila, C++, Java, smalltalk,…
- Others (e.g., logic programming, concurrent programming)

A few successful languages

- Fortran --- the first high-level programming language
  - Led by John Backus around 1954-1956
  - Designed for numerical computations
  - Introduced variables, arrays, and subroutines
- Lisp
  - Led by John McCarthy in late 1950s
  - Designed for symbolic computation in artificial intelligence
  - Introduced high-order functions and garbage collection
  - Descendents include Scheme, ML, Haskell, …
- Algol
  - Led by a committee of designers of Fortran and Lisp in late 1950s
  - Introduced type system and data structuring
  - Descendents include Pascal, Modula, C, C++ …
- Simula
  - Led by Kristen Nygaard and Ole-Johan Dahl arround 1961-1967
  - Designed for simulation
  - Introduced data-abstraction and object-oriented design
  - Descendents include C++, Java, smalltalk …

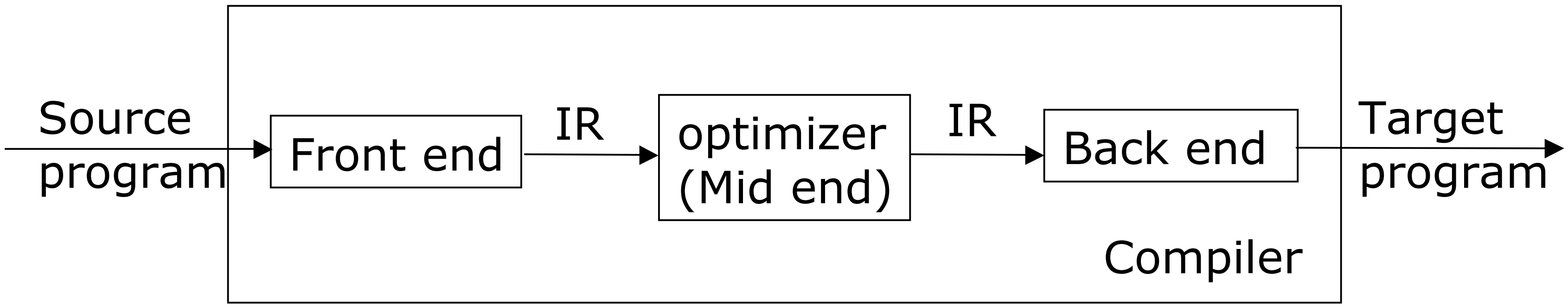
Categorizing Languages

- Are these languages compiled or interpreted (sometimes both)? What paradigms do they belong?
  - C
  - C++
  - Java
  - PERL
  - bsh, csh
  - Python
  - C#
  - HTML
  - Postscript
  - Ruby
  - …

Objectives of Compilers

- Fundamental principles of compilers
  - Correctness: compilers must preserve semantics of the input program
  - Usefulness: compilers must do something useful to the input program
  - Compare with software testing tools---which must be useful, but not necessarily sound
- The quality of a compiler can be judged in many ways
  - Does the compiled code run with high speed?
  - Does the compiled code fit in a compact space?
  - Does the compiler provide feedbacks on incorrect program?
  - Does the compiler allow debugging of incorrect program?
  - Does the compiler finish translation with reasonable speed?
- Similar principles apply to software tools in general
  - Are they sound? Do they produce useful results? How fast do they run? How fast are the generated code?

The structure of a compiler/translator



- Front end --- understand the input program
  - Scanning, parsing, context-sensitive analysis
- IR --- intermediate (internal) representation of the input
  - Abstract syntax tree, symbol table, control-flow graph
- Optimizer (mid end) --- improve the input program
  - Data-flow analysis, redundancy elimination, computation re-structuring
- Back end --- generate output in a new language
  - Native compilers: executable for target machine
    - Instruction selection and scheduling, register allocation

What is common and different in an interpreter?

Front end

- Source program

  for (w = 1; w < 100; w = w * 2);
- Input: a stream of characters
  - ‘f’ ‘o’ ‘r’ ‘(’ `w’ ‘=’ ‘1’ ‘;’ ‘w’ ‘<’ ‘1’ ‘0’ ‘0’ ‘;’ ‘w’…
- Scanning--- convert input to a stream of words (tokens)
  - “for” “(“ “w” “=“ “1” “;” “w” “<“ “100” “;” “w”…
- Parsing---discover the syntax/structure of sentences

  forStmt: “for” “(” expr1 “;” expr2 “;” expr3 “)” stmt

  expr1 : localVar(w) “=” integer(1)

  expr2 : localVar(w) “<” integer(100)

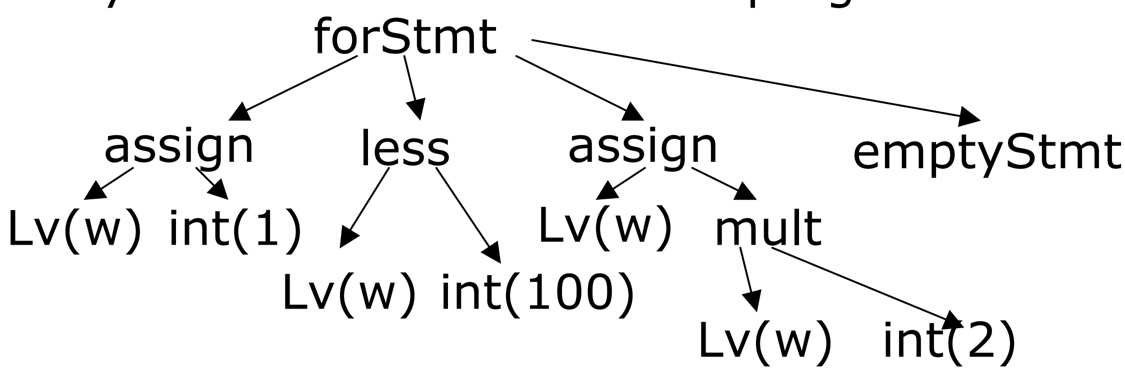
  expr3: localVar(w) “=” expr4

  expr4: localVar(w) “*” integer(2)

  stmt: “;”

Intermediate Representation

- Source program
  for (w = 1; w < 100; w = w * 2);
- Parsing --- convert input tokens to IR
  - Abstract syntax tree --- structure of program



- Context sensitive analysis --- the surrounding environment
  - Symbol table: information about symbols
    - v: local variable, has type “int”, allocated to register
  - At least one symbol table for each scope

More About The Front End

```
int w;
0 = w;
for (w = 1; w < 100; w = 2w)
        a = "c" + 3;
```

- What errors are discovered by
  - The lexical analyzer (characters ➔ tokens)
  - The syntax analyzer (tokens ➔ AST)
  - Context-sensitive analysis (AST➔symbol tables)

Mid end --- improving the code

Original code

```
int j = 0, k;
while (j < 500) {
    j = j + 1;
    k = j * 8;
    a[k] = 0;
}
```

Improved code

```
int k = 0;
while (k < 4000) {
    k = k + 8;
    a[k] = 0;
}
```

- Program analysis --- recognize optimization opportunities
  - Data-flow analysis: where data are defined and used
  - Dependence analysis: when operations can be reordered
  - Useful for program understanding and verification
- Optimizations --- improve program speed or space
  - Redundancy elimination
  - Improve data movement and instruction parallelism
  - In program evolution, improve program modularity/correctness

Back end --- Code Generation

- Machine code generation
  - Memory management
    - Every variable must be allocated with a memory location
    - Address stored in symbol tables during translation
  - Instruction selection
    - Assembly language of the target machine
    - Abstract assembly (three/two address code)
  - Register allocation
    - Most instructions must operate on registers
    - Values in registers are faster to access
  - Instruction scheduling
    - Reorder instructions to enhance parallelism/pipelining in processors
- Source-to-source translation
  - Program understanding --- output analysis results
  - Code generation/evolution/optimization --- output in high-level languages

Roadmap

- Week1-4 --- front end (parsing)
  - Regular expression and context-free grammar(wk1), NFA and DFA(wk2), top-down and bottom-up parsing(wk3), attribute grammar and type checking(wk4)
- Week5-9--- back end (code generation)
  - Intermediate representation(wk5), procedural abstraction and code shape(wk6-7), instruction selection(wk8)
- Week9-13 --- mid end (program optimizations)
  - Redundancy elimination(wk9), data-flow analysis and SSA(wk10), scalar optimizations(wk11), instruction scheduling(wk12), register allocation(wk13)
- Project: build a small compiler/translator/development tool
  - Needs to parse input in a small language, perform type checking, perform some analysis/optimization, then output the result
  - Intermediate projects are due by week 4, week 9, and week 11 respectively (dates will be posted at class web site)
- Implementation choices:
  - Understanding of concepts/algorithms: smaller size projects in scripting languages
  - Enjoys programming and debugging: larger projects in C/C++/Java