Compiler Writing

Qing Yi

class web site: www.cs.utsa.edu/~qingyi/cs4713

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 construction
  program analysis; optimizations for high-performance computing.
- Programming languages
  type systems, object-oriented design.
- Software engineering
  automatic structure discovery of software systems; systematic error-discovery and verification of software.

General Information

- Class website
  - www.cs.utsa.edu/~qingyi/cs4713
  - Check it often for slides, handouts and announcements
- Textbook
  - Compilers: Principles, Techniques, and Tools
    - Second edition
    - By Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Addison-Wesley.
- Prerequisites
  - Basic understanding of computer organization and algorithms
  - Ability to program in C and Java

What we will learn

- Understanding languages and compilers
  - How to implement different programming languages?
  - How to automatically parse a language?
    - Why are some languages harder to process than others?
  - How to translate a language into another language?
  - How to automatically improve the quality of programs?
- Implementation of compilers
  - Scanners and parsers
  - Symbol table management
  - Simple code optimization
  - Code generation
- Critical thinking
  - Why are things the way they are? Could they be different?

Class Objectives

- Understand compilers as a means to implement programming languages
  - compilation vs. interpretation
  - phases of a compiler
- Understand fundamental theories and algorithms
  - regular expressions and context-free grammars
  - NFA and DFA
  - top-down and bottom-up parsing
  - code generation and optimization algorithms
- Practice implementing compilers
  - Learn how to implement scanners and parsers
  - Learn how to implement significant algorithms

Requirements and grading

- Quizzes in class: 20% (you’re required to attend class)
  - I will hand out and collect quiz questions in class
  - You pay attention to the lecture and find out solutions
  - I will give you time to work on the quiz questions
  - You’ll know if you understand class materials
    - If not, interrupt me immediately
- Projects and homework: 50% (hands-on experience with compilers)
  - depend on our progress, but will cover lexical analysis, parsing and code generation.
- Exams: 30%
  - Two midterms --- selected from past quiz questions (with variation, of course)
  - The final is not required if you’ve done well on the midterms

Attendance and quizzes

- Q: I have the textbook and the class notes online, do I have to attend every class?
- A: Absolutely.
  - The lecture will cover more to enhance your overall understanding of the topics
  - The class notes are mostly abstract outlines of things to cover
  - Don't put off learning until the end of the term
    - Quizzes and projects count toward 70% of the grade
    - The quizzes and solutions are complimentary class notes
- What if I have to miss a class due to unusual situations?
- A: you can come to my office hours and make up missed quizzes. But you need to give me a good reason. Bad reasons include:
  - I have to prepare the exam of another class
  - I have to go to a job fair. They give out very cool stuffs
  - I forget to show up. I couldn't find a parking spot. …

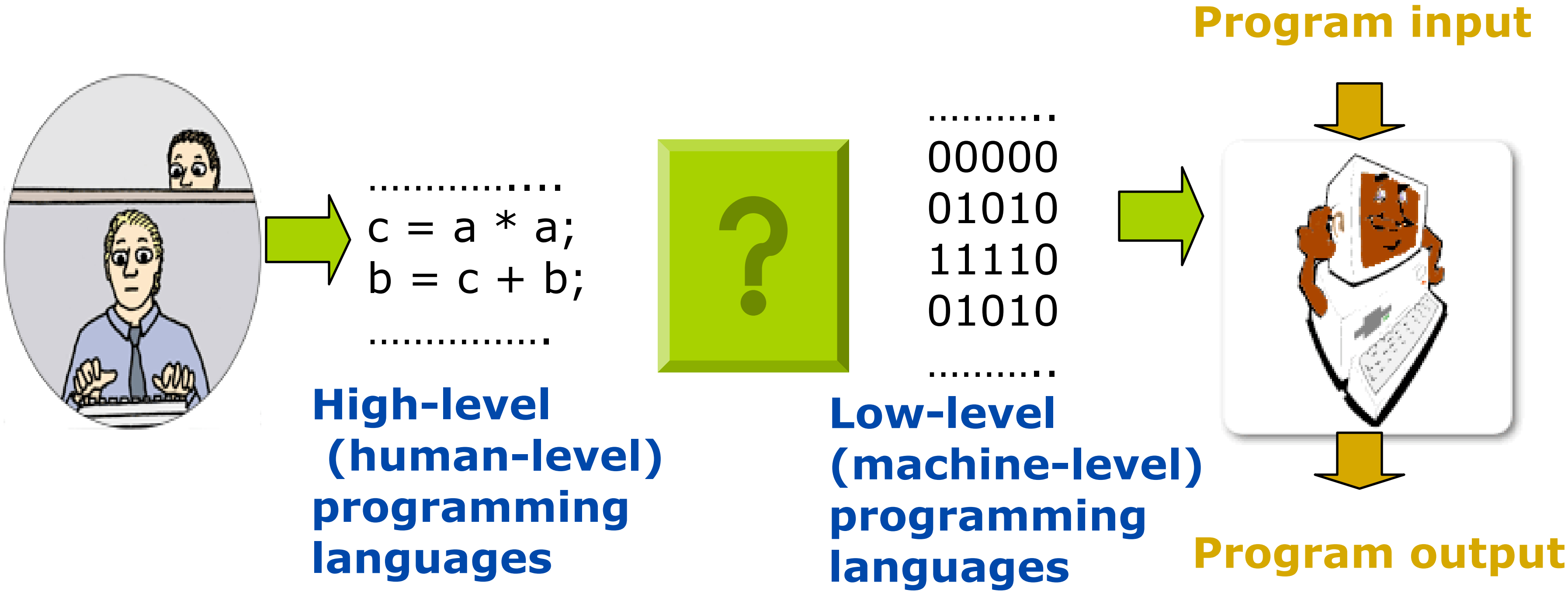
Self evaluation

- How am I doing? How do I know whether I'm getting an A?
- A: exams matter, but quizzes and projects count toward 70% of the grade
  - I can give you feedback on the quizzes and projects --- send me email, or sign up now.
  - You are likely getting an A if you do all of these
    - Attend every class and turn in the quiz solutions.
    - If your quiz solution show you do not yet understand the material, come to my office hours and fix it.
    - Your projects work well.
    - Prepare for the exams.
  - You might get a C or even fail the class if you do any of these
    - Skip a lot of classes. Do not turn in the quizzes.
    - Couldn't get your projects to work at all, and do not come to my office hours and ask for help.
    - Believe you already know everything and skip preparing for exams.

Programming Languages

- Natural languages
  - Tools for expressing information
    - ideas, knowledge, commands, questions, …
    - Facilitate communication between people
  - Different natural languages
    - English, Chinese, French, German, …
- Programming languages
  - Tools for expressing data and algorithms
    - Instructing machines what to do
    - Facilitate communication between computers and programmers
  - Different programming languages
    - FORTRAN, Pascal, C, C++, Java, Lisp, Scheme, ML, …

Levels of Programming Languages



Program input

…………….…
c = a * a;
b = c + b;
……………..

High-level
(human-level)
programming
languages

?

………..
00000
01010
11110
01010
………..

Low-level
(machine-level)
programming
languages

Program output

For future reference
programming language =>high-level language

Benefits of high-level languages

- ✓ Efficiency of programming
  - Higher level mechanisms for
    - Describing relations between data
    - Expressing algorithms and computations
  - Error checking and reporting capability
- ✓ Machine independence
  - Portable programs and libraries
- ✓ Maintainability of programs
  - Readable notations
  - High level description of algorithms
  - Modular organization of projects

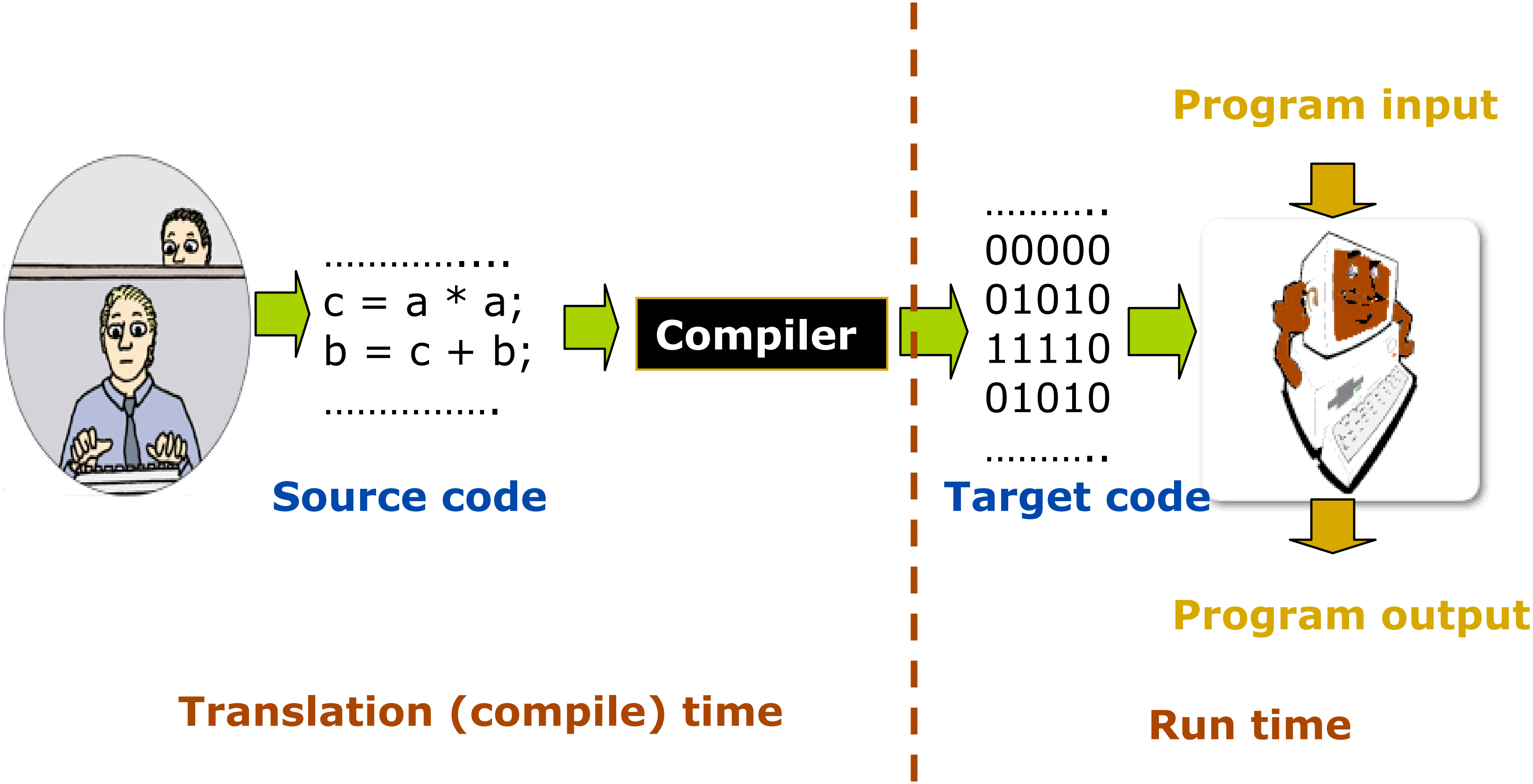
- X Machine efficiency
  - Extra cost of compilation / interpretation

Benefits of high-level languages

- ✓ Efficiency of programming
  - Higher level mechanisms for
    - Describing relations between data
    - Expressing algorithms and computations
  - Error checking and reporting capability
- ✓ Machine independence
  - Portable programs and libraries
- ✓ Maintainability of programs
  - Readable notations
  - High level description of algorithms
  - Modular organization of projects

- X Machine efficiency
  - Extra cost of compilation / interpretation

Implementing programming languages Compilation



Program input

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

Compiler

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

Source code

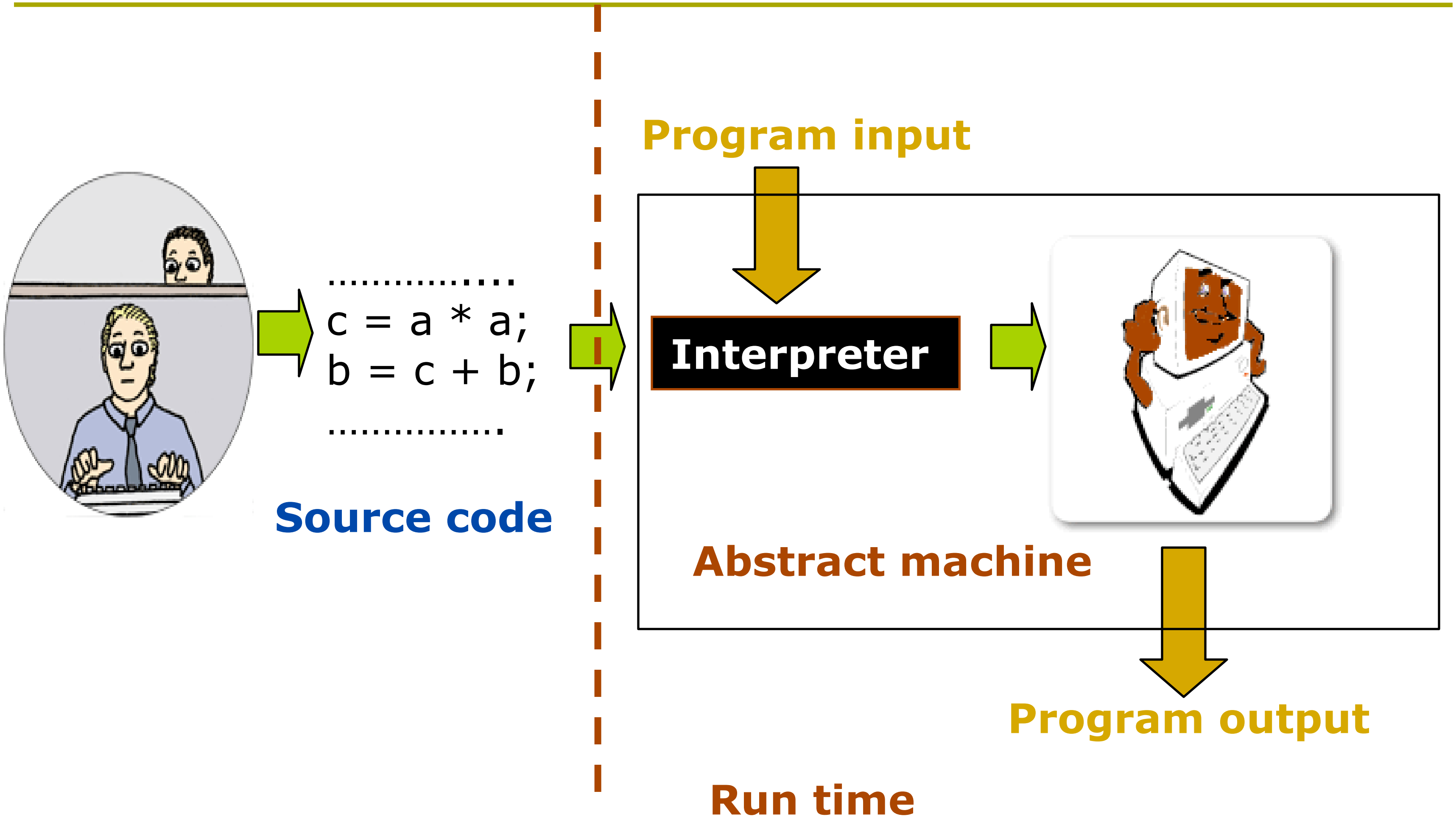
Target code

Program output

Translation (compile) time

Run time

Implementing programming languages Interpretation



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

Source code

Program input

Interpreter

Abstract machine

Program output

Run time

Are these languages compiled or interpreted (sometimes both)?

- C/C++
- Java
- PERL
- bsh, csh
- Python
- C#
- HTML
- Postscript
- …

Compilers and Interpreters Translation vs. Interpretation

- Compilers
  - Read input program → optimization → translate into machine code
- Interpreters
  - Read input program → interpret the operations
- Questions to think about
  - What are the tradeoffs of using compilers and interpreters?
  - What languages are compilers and interpreters written in?
  - What about the first compiler or interpreter?

Compilers and Interpreters Efficiency vs. Flexibility

- □ Compilers

  Translation time is separate from run time

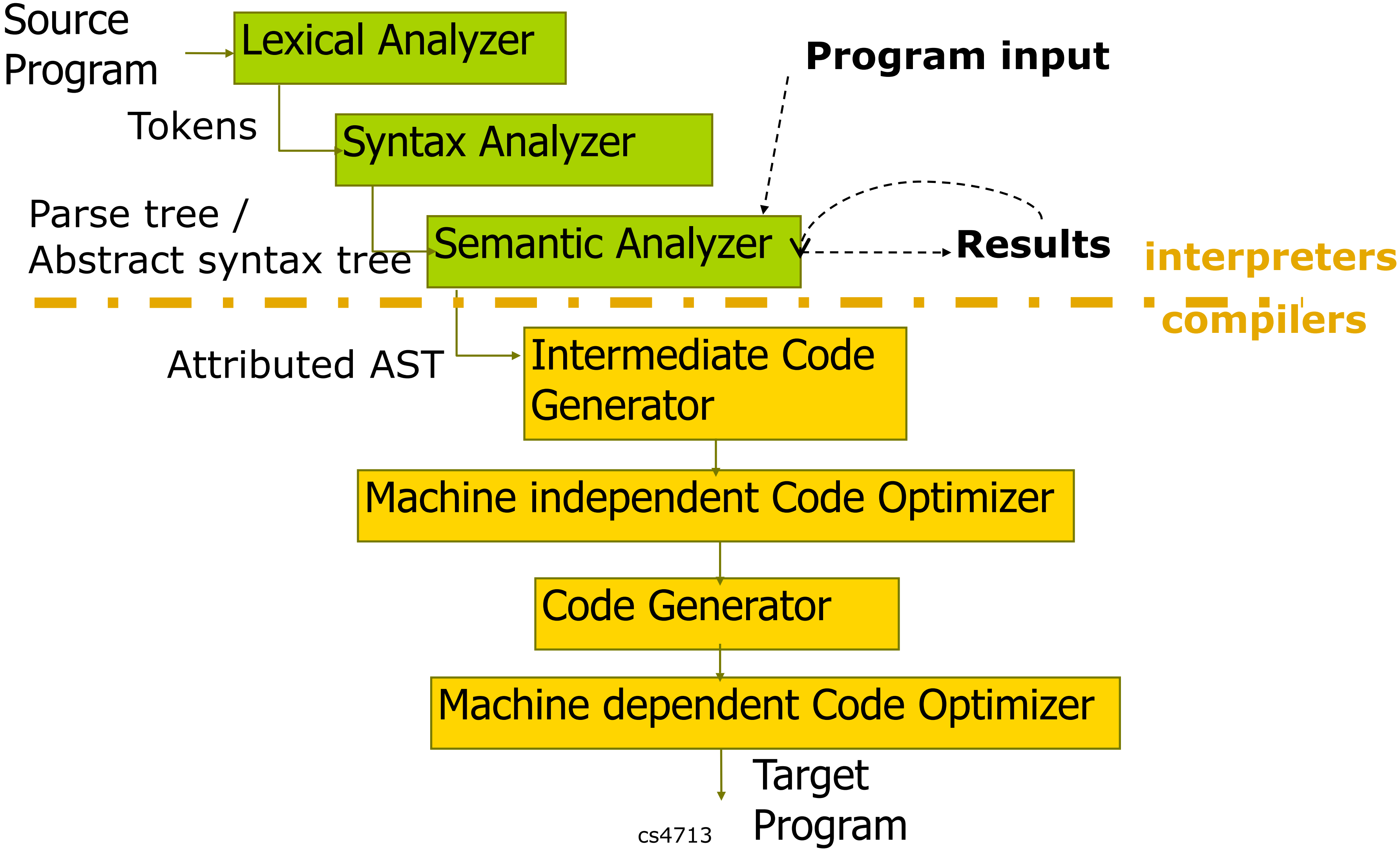
  - □ Each target code can run many times
  - □ Heavy weight optimizations are affordable
  - □ Can pre-examine programs for errors
  - x Static analysis has limited capability
  - x Cannot change programs on the fly

- □ Interpreters

  Translation time is included in run time

  - x Re-interpret each 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

Typical Implementation of Languages



Source Program

Lexical Analyzer

Tokens

Syntax Analyzer

Parse tree / Abstract syntax tree

Semantic Analyzer

Program input

Results

interpreters

compilers

Attributed AST

Intermediate Code Generator

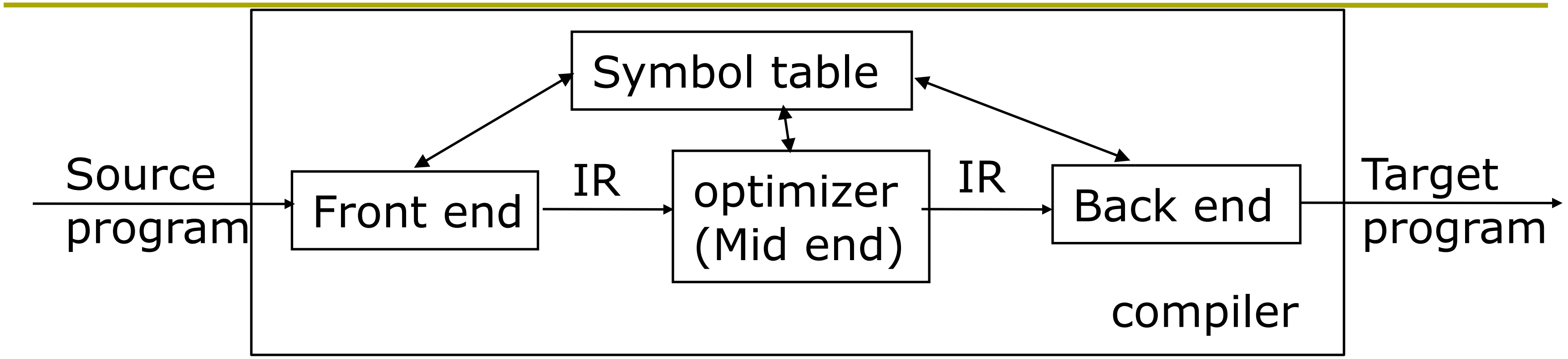
Machine independent Code Optimizer

Code Generator

Machine dependent Code Optimizer

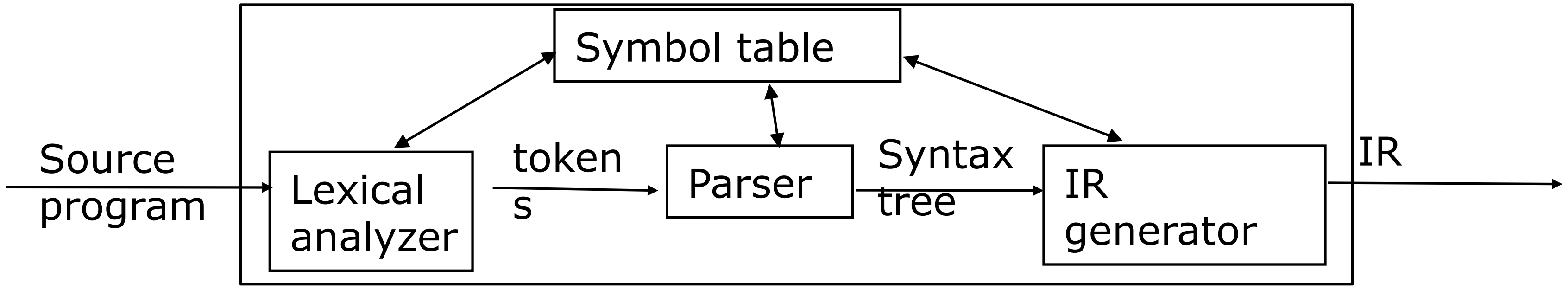
Target Program

Compiler structure



- Front end --- understand the source program
  - Scanning, parsing, context-sensitive analysis
- IR --- intermediate (internal) representation of the input
  - Abstract syntax tree, control-flow graph
- Optimizer (mid end) --- improve the input program
  - Data-flow analysis, redundancy elimination, computation re-structuring
- Back end --- generate executable for target machine
  - Instruction selection and scheduling, register allocation
- Symbol table --- record information about names(variables)

Compiler Frontend



- 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”…
  - <FOR> <LPAREN> <id,1> <ASSIGN> <int,1> <SEMICOLON> ...

Symbol table: 1

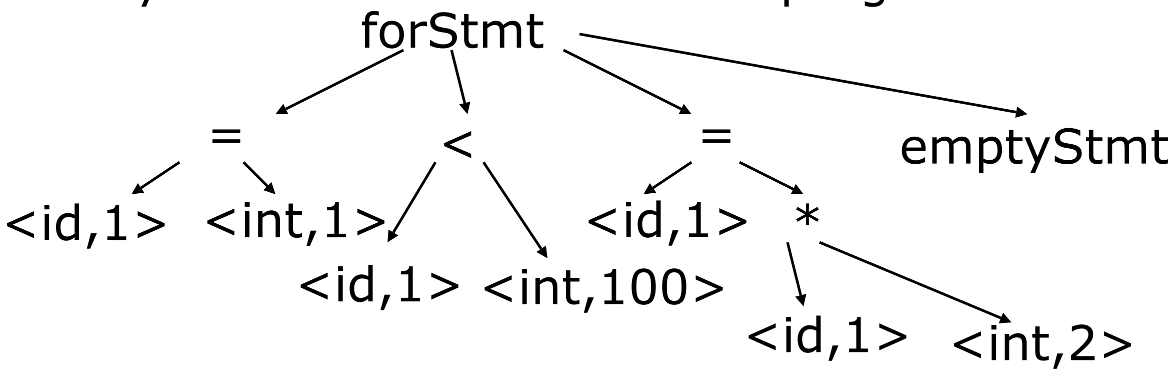
“w”	….	….
….	….	….

- Parsing---discover the syntax/structure of sentences
  - FOR <LPAREN> exp <SEMICOLON> exp <SEMICOLON> exp <RPAREN> 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
    - w: 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)
- How do you implement AST and symbol table

```
typedef struct ASTnode {
    AstNodeTag kind;
    union { symbol_table_entry* id_entry;
                int  num_value;
                struct ASTnode* opds[2];
              } description;
};
```

Mid end --- improving code quality

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
- Transformations --- improve target program speed or space
  - Redundancy elimination
  - Improve data movement and instruction parallelization

Back end --- 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

Objectives of compilers

- Fundamental principles
  - Compilers shall preserve the meaning of the input program --- it must be correct
    - Translation should not alter the original meaning
  - Compilers shall do something of value
    - They are not just toys
- How to judge the quality of a compiler
  - 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?
- What kind of compilers do you like?
  - Gnome compilers, Sun compilers, Intel compilers, Java compilers, C/C++ compilers, ……

Applications of Compiler technology

- Implementing high-level programming languages
  - Compilation vs. interpretation
  - C/C++, Fortran, Java, C#
- Optimizations for computer architectures
  - exploiting parallelism, memory hierarchy, and specialized architectures
- Program Translation
  - Binary translation, hardware synthesis, database query, compiled simulation
- Software productivity tools
  - Program analysis to prove correctness or report errors and to automatically discover code structure
  - Type checking, bounds checking, memory management, ...