Programming Language Principles

Programming Language Theory

Topics

- What is a Computer?
- Turing Machine
- **How to implement a PL?**
  - **Compiler & Interpreter**

How to Implement a PL?

- Human: high-level languages are more easy to understand.
- Computer: it can only understand machine instructions.
- PL implementation: implement a process of ***translating high-level language to low-level language***, so that a program written by human can be directly executed by a computer.
- Such translation is done by a compiler or an interpreter.
- Eventually *PL implementation is equivalent to compiler or interpreter implementation*.

Compiler vs. Interpreter

- Both a compiler and an interpreter convert code written by human to low-level code which a machine can execute.
- Compiler
  - complete code → executable program.
- Interpreter
  - read & evaluate expressions → execute instructions.

Compiler vs. Interpreter

- Compiler
    - Focus on execution performance and efficiency of a generated executable program.
    - Relatively difficult to connect code and execution.
    - Find errors at compile time.
- Interpreter
    - Easy to implement, but slower.
    - Runtime errors → directly linked to code.
    - Can execute partial code (only some expressions).

Compiler vs. Interpreter

C++

```cpp
#include<iostream>
using namespace std;

int main() {
    int a = 10, b = 5;
    cout << a / b << endl;

    return 0;
}
```

Compiler

Python

```
>>> a = 10
>>> b = 5
>>> a / b
2.0
>>>
```

Interpreter

Compiler

- Convert code in high-level language to machine instructions executable in a target machine.
- Translator between humans and machines.
- A compiler generates code in an object (target) language, and its output is often called an object file.
- Then these object files are combined into one executable program.

Compilation Steps

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation

Lexical Analysis

- The first phase of a compiler.
- Convert source code into a series of **tokens**.
  - e.g.) keywords, literals, identifiers, numbers, operators, etc.
  - int a = 10; → int (keyword), a (identifier), = (operator), 10 (number literal), ; (symbol).
- Remove whitespaces and comments.
- If a token is invalid, it causes an error.

Syntax Analysis

- Now we have a series of tokens.
- In syntax analysis step, we verify whether the sequence of the tokens follows correct syntax.
- This step is often called “Parsing”.
- Producing Abstract Syntax Tree (AST) or Parse Tree, which represents syntactic structure of source code.
- Code which cannot be parsed → syntactically incorrect!

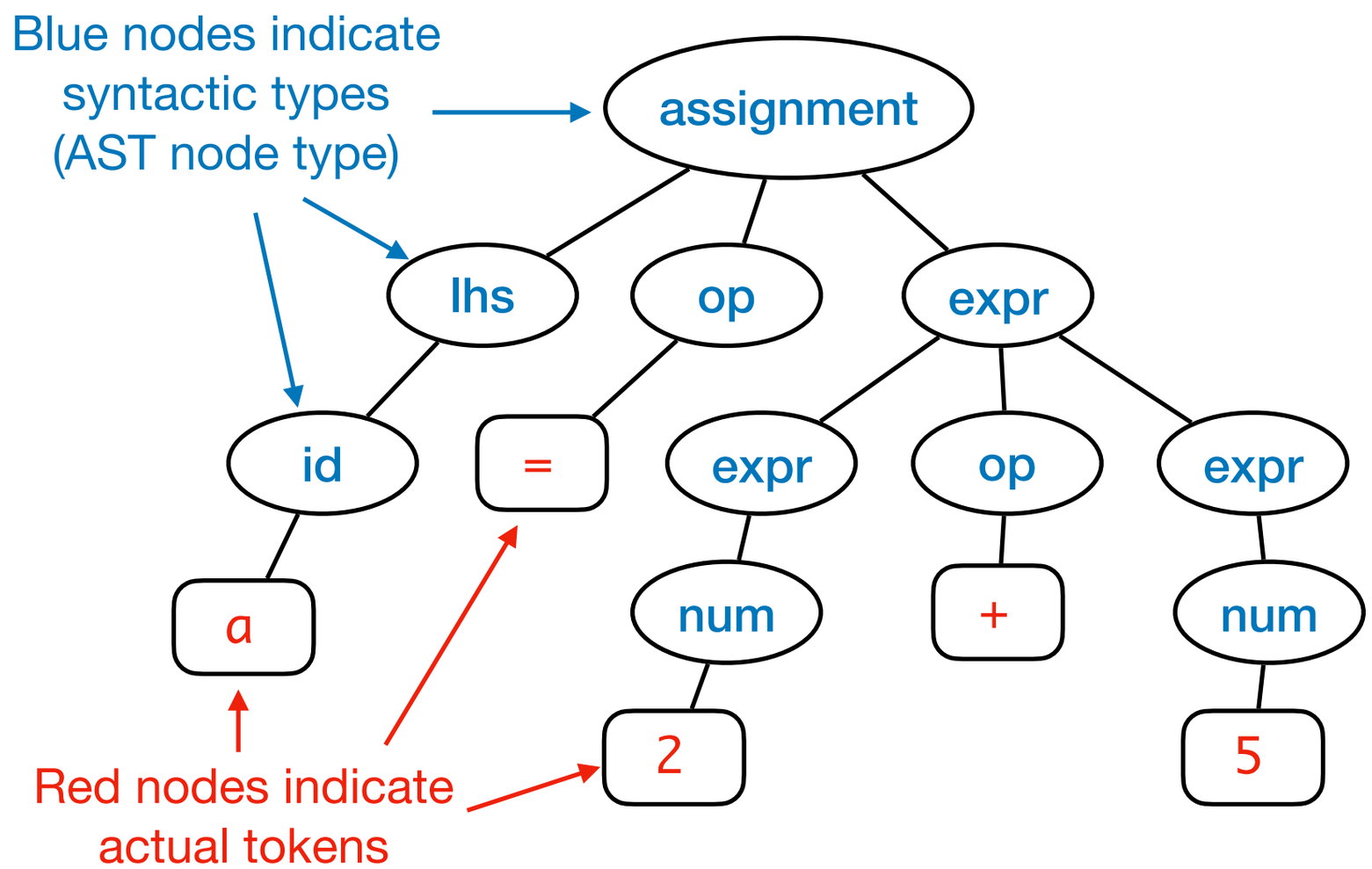
Syntax Analysis

```
Code: a = 2 + 5;
```

Lexical Analysis

```
Tokens: a (id)
        = (op)
        2 (num)
        + (op)
        5 (num)
        ; (sym)
```

Parse Tree



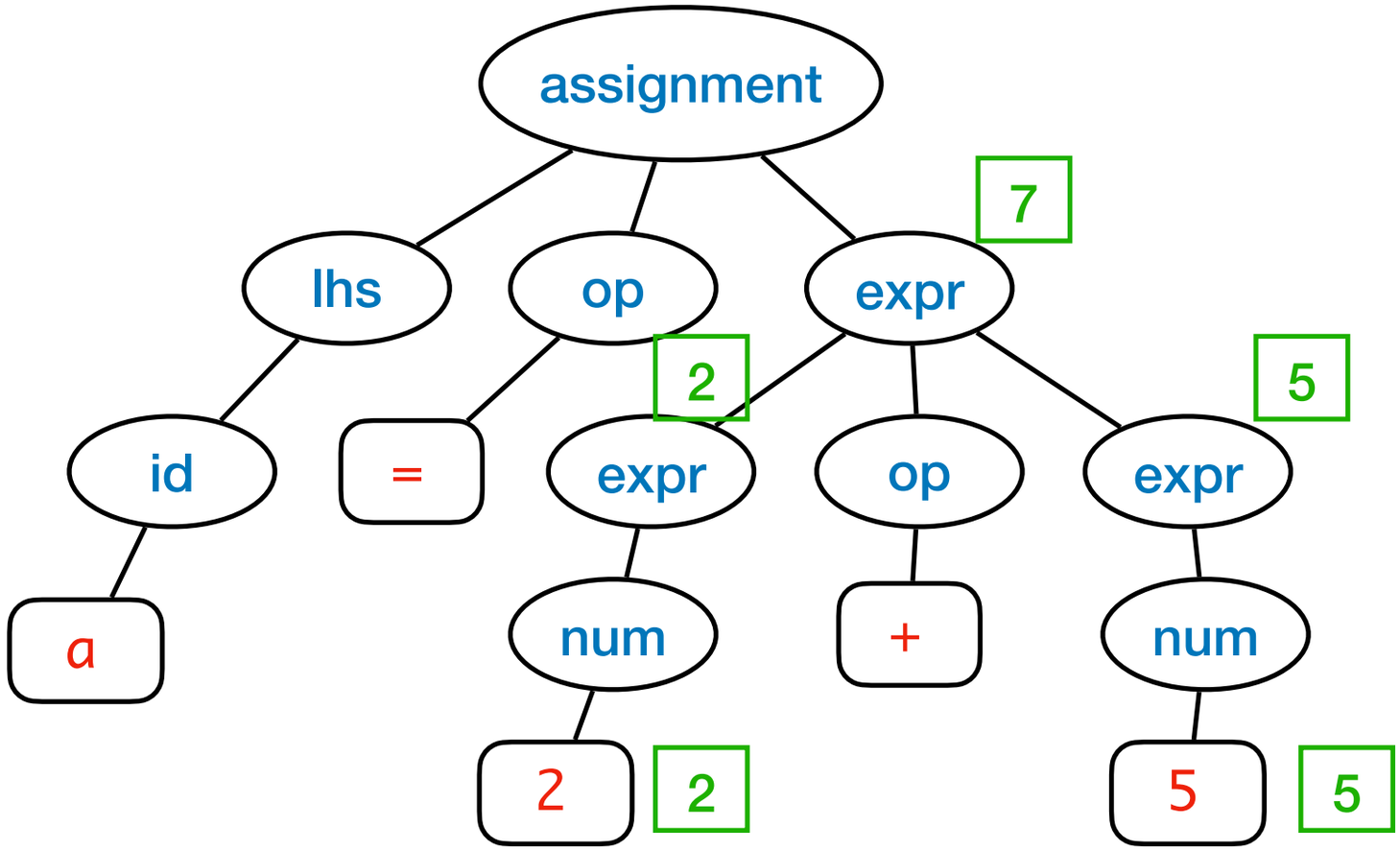
<assignment> → <lhs> <op> <expr>

Semantic Analysis

- Parse tree is “annotated” or “decorated” by semantic analysis.
- The information gathered during this step will be used for code generation.
- Syntax analysis gives nothing about what the code **means**.
- Hence we need to figure out the meaning of code - at least evaluate them.
- Often combined with syntax analysis.

Semantic Analysis

Augmented Parse Tree



It means that 7 will be assigned to a

a ← 7

Intermediate Code Generation

- The final outcome of a compiler is often ***machine dependent***.

- Hence machine code is not a desirable target to apply ***machine independent code optimizations***.

- Instead, a compiler generates intermediate code, which is independent to target machines.

- Then it optimizes code first, and further translate them to instructions for a target machine.

Code Optimization

- A compiler automatically performs code optimizations before it generates an executable program.
  - e.g.) dead code elimination, common subexpression elimination, copy propagation, loop optimization, etc.
  - `a=i*j+k; b=i*j*k;` → `tmp=i*j; a=tmp+k; b=tmp*k;`
- Such optimization might improve your code significantly.
- One of the most important advantage compared to an interpreter.

Code Generation

- Takes intermediate code as an input and produces an equivalent target program.

- Requirements
  - The output code must be correct.
  - The output code should use resource of a target machine effectively.
  - Code generator itself should run efficiently.

Interpreter

- Directly read source code, and execute tasks corresponds to the code.

- Sometimes it implements a virtual computer runs on a real computer, then executes code on the virtual computer.

- It is more easy to implement, occupies less memory than a compiler, but it’s slower.

- e.g.) Python, ML, Scheme, Prolog.

REPL

- ***R**ead-**E**val-**P**rint* **L**oop.

- An interpreter actually repeats the above three tasks.

- Doesn’t require whole program, it simply evaluates what it reads.

- You can write down code at runtime → easy to use.

Summary

- How to implement a PL?
  - Compiler
  - Interpreter