Class Information

- Second homework late submissions ?
- Third homework has been posted; due next Friday.
- Our first programming project will be posted next week, tentatively Wednesday. Due on Friday, March 7.
- Next lecture, Prof. Zheng Zhang

Review: Recursive Descent LL(1) Parsing

Recursive descent LL(1) parsing is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated **parsing procedure** that can recognize any sequence of tokens generated by that non–terminal.
- There is a **main** routine to initialize all globals (e.g.: `token`) and call the start symbol. On return, check whether `token` == **eof**, and whether errors occurred.
- Within a parsing procedure, both non–terminals and terminals can be matched:
  - non–terminal A — call parsing procedure for A
  - token t — compare t with current input token; if match, consume input, otherwise ERROR
- Parsing procedures may contain code that performs some useful “computation” (syntax directed translation).

First Project: tinyL Language

<program> ::= <stmtlist> .

<stmtlist> ::= <stmt> <morestmts>

<morestmts> ::= ; <stmtlist> | ϵ

<stmt> ::= <assign> | <read> | <print>

<assign> ::= <variable> = <expr>

<read> ::= ? <variable>

<print> ::= ! <variable>

<expr> ::= + <expr> <expr> |
- <expr> <expr> |
* <expr> <expr> |
<variable> |
<digit>

<variable> :: = a | b | c | d | e

<digit> :: = 0 | 1 | 2 | 3 | ... | 9

Review: Syntax Directed Translation

Examples:

1. Interpreter
2. Code generator
3. Type checker
4. Performance estimator

Use hand-written recursive descent LL(1) parser

Example: Simple Code Generation

<expr> ::= + <expr> <expr> |

<digit>

<digit> :: = 0 | 1 | 2 | 3 | ... | 9

```
int expr:  // returns target register of operation
  int target_reg, reg1, reg2; // registers
  switch token {
    case +:    token := next_token( );
               target_reg = next_register( ); // ‘‘fresh’’ register
               reg1 = expr( ); reg2 = expr( );
               CodeGen(ADD, reg1, reg2, target_reg);
               return target_reg;
    case 0..9: return digit( );
    ...
  }

int digit:  // returns target register of operation
  int target_reg = next_register( ); // ‘‘fresh’’ register
  switch token {
    case 1:    token := next_token( );
               CodeGen(LOADI, 1, target_reg);
               return target_reg;
    case 2:    token := next_token( );
               CodeGen(LOADI, 2, target_reg);
               return target_reg;
    ...
  }
```

Example: Simple Code Generation

What happens when you parse subprogram

“**+ 2 + 1 2**” ?

Assumption:

$\boxed{\text{first call to } \texttt{next\_register( )} \text{ will return 1}}$

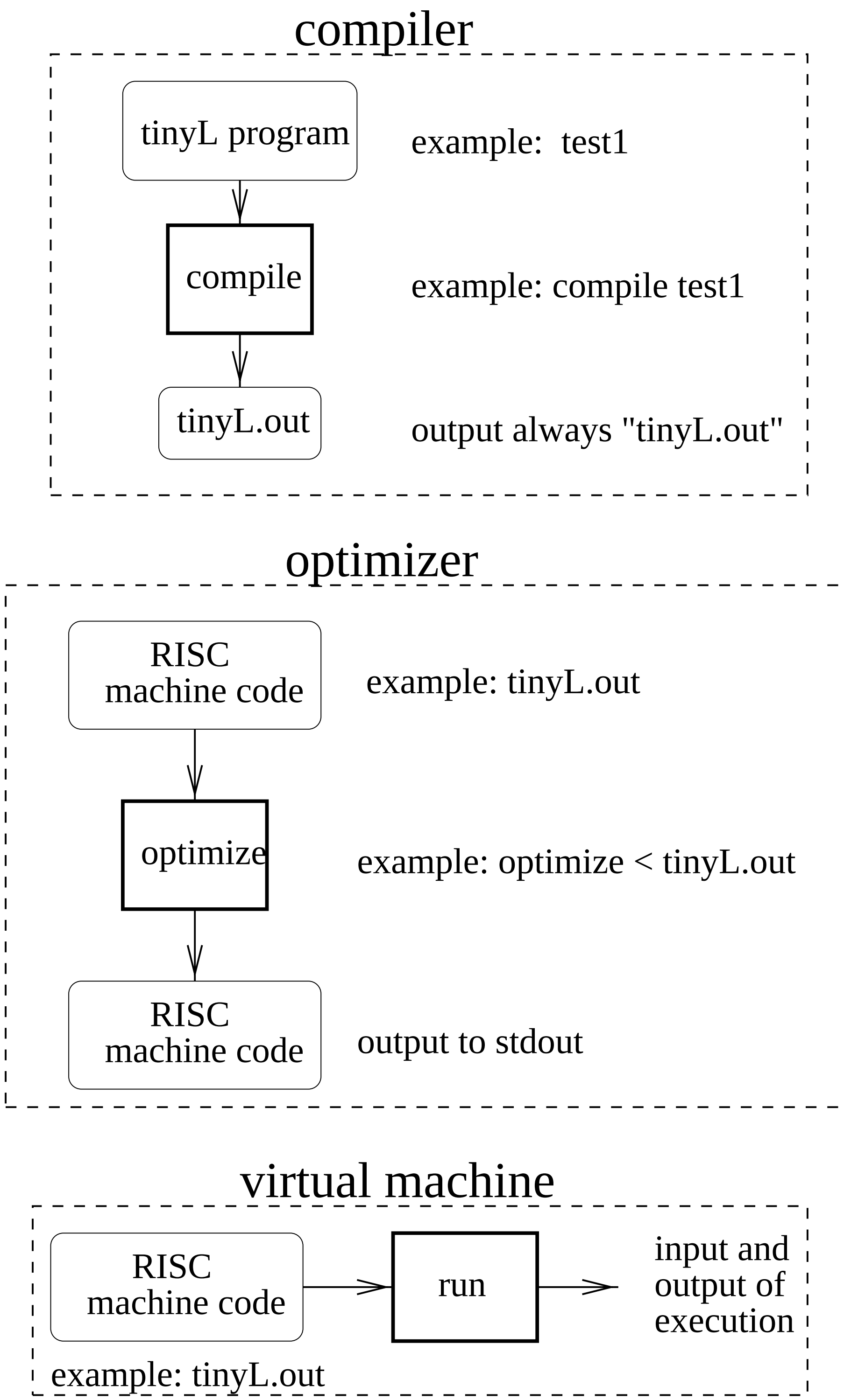
The parsing produces (ILOC code):

```
LOADI 2 => r2
LOADI 1 => r4
LOADI 2 => r5
ADD r4, r5 => r3
ADD r2, r3 => r1
```

The parsing produces (project 1 code):

```
LOADI r2 2
LOADI r4 1
LOADI r5 2
ADD r3 r4 r5
ADD r1 r2 r3
```

Project 1: Overview



Project 1: Dead Code Elimination Optimization

Goal: Identify instructions that do not contribute to the input/output behavior of the program.

These instructions are considered “dead” and can be eliminated.

Example:

```
LOADI r1 5
LOADI r2 7
LOADI r3 2
ADD r4 r1 r2
MUL r5 r1 r2
STORE a r5
WRITE a
```

Are there any “dead” instructions that can be eliminated?

Project 1: Important Things to Note

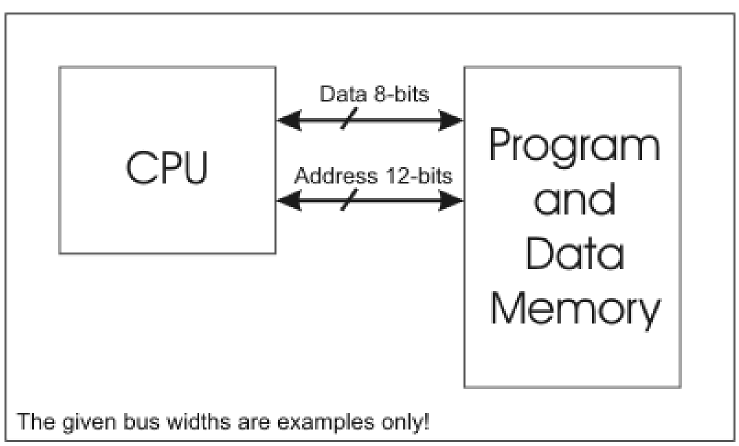
1. Will be posted by next Wednesday.
2. Submission deadline: Friday, March 7, at midnight.
3. You will submit your **source code only**. Your compiler and optimizer has to compile and run on the ilab cluster. **A project that does not compile on the ilab cluster machines will receive no credit!**
4. This is **not a group project**. You may discuss the project with your fellow students in general terms, but are not allowed to share code. **Do not cheat! Read protect your project files.**
5. We will use mainly automatic tools to grade your project. If your compiler and/or optimizer fails to run correctly on a test case, you will not receive any credit for that test case.
6. We will give you a few test cases to evaluate your compiler and optimizer. In addition, you will need to come up with your own test cases. **Do not submit your test cases.**

Imperative Programming Languages

Imperative:

Sequence of state-changing actions.

- Manipulate an abstract machine with:
  1. Variables naming memory locations
  2. Arithmetic and logical operations
  3. Reference, evaluate, assign operations
  4. Explicit control flow statements
- Key operations: *Assignment* and "*Goto*"
- Fits the von Neumann architecture closely

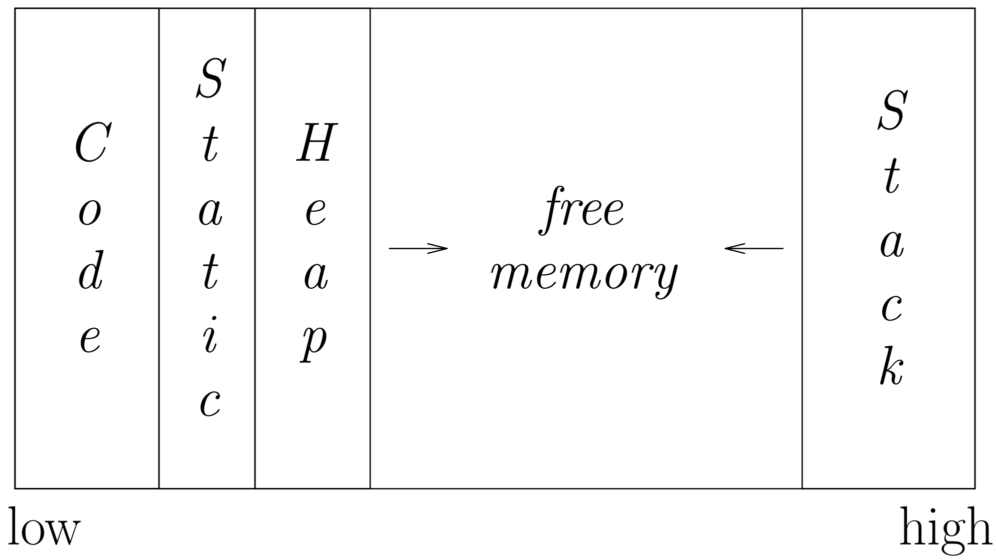


Von Neumann Architecture

Run-time storage organization

Typical memory layout

Logical Address Space



The classical scheme

- allows both stack and heap maximal freedom
- code and static may be separate or intermingled

Will talk about this in more detail in a later lecture!

Next Lecture

Things to do:

Start programming in C. Check out the web for tutorials.

Read Scott: Chap. 3.1 - 3.3; ALSU Chap. 7.1

Next time:

- Prof. Zheng Zhang will teach the class.
- Imperative programming and C; pointers in C; dynamic memory allocation.