Course Project
Phase #2. IR Translation

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

General Information

- **Check "Project #2" in *Assignment* tab of *Cyber Campus***
  - Skeleton code (**`Prj2.tgz`**) is attached in the post
  - Deadline: **12/06** Wed. 23:59
  - Submission will be accepted in that post, too
  - Late submission deadline: **12/08** Fri. 23:59 **(-20% penalty)**
- **Please read the instructions in this slide carefully**
  - **Lots of important information is included**
  - The slide also contains important submission guidelines
    - If you do not follow the guidelines, you will get penalty

Remind: Course Policy

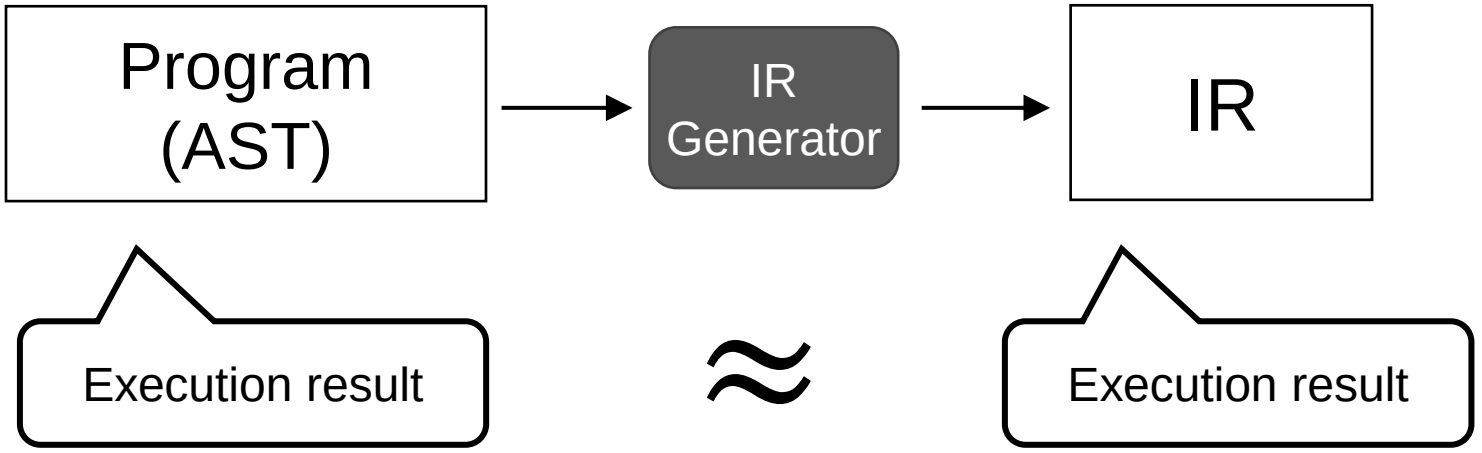
- **Cheating (code copy) is strictly forbidden in this course**
  - Read the orientation slide once more
- **Don’t ask for solutions in the online community**
  - TA will regularly monitor the communities
- **Don’t ask ChatGPT to write your code**
- **Even after the end of the course, please do not upload your code at GitHub or share it with your friends**
  - This makes it hard to manage the course in the following years

Skeleton Code

- **Copy Prj1.tgz into CSPRO server and decompress it**
  - Don’t decompress-and-copy; copy-and-decompress
  - You can use cspro**5**.sogang.ac.kr / cspro.sogang.ac.kr
- **`src/`: Source files you have to work with**
- **`Makefile`: Type `make` to build the whole project**
  - Internally redirects to **`src/Makefile`**
- **`testcase/`: Sample test cases and their answers**
- **`check.py`: Script for self-grading with test cases**
- **`config`: Used by the grading script (you can ignore)**

Outline

- **In this phase, you will write an AST-to-IR translator**
- **Generated IR must have equivalent behavior to the original source program**
  - A function must return the same value
- **How can we test such equivalence?**
  - IR executor (interpreter) is provided in the project code



Changes to the Source Language

- **There are several changes to our source language**
- **Simplified features**
  - Starting from this phase, program will **have only one function**
  - Execution of program means the execution of this function
  - Thus, we don’t have global variables or function calls anymore
- **Newly added feature**
  - Now our programs can **have arrays**: but limited to 1-dimensional
- **To sum up, we will focus on translating just one function**
  - To test the equivalence of function, now the argument and return type of function **must not be void**

Structure of `src` Directory

- Many files are same with the previous phase
- Only important files are summarized below
- `program.ml` : Definition of the AST for program
- `ir.ml` : Definition of IR code (including instructions)
- `translate.ml` : Translation from AST to IR
- `helper.ml` : Provides some functions for translation
- `state.ml, eval.ml, executor.ml` : Responsible for executing the IR code (you will not have to read these)

Where do I have to read and fix?

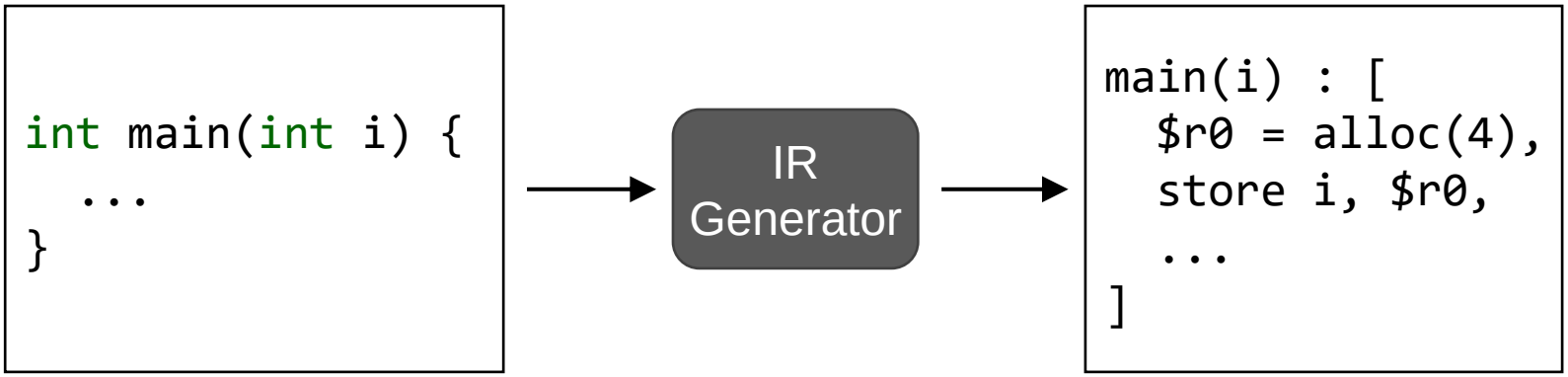
- **First, read `program.ml` and confirm what kind of changes are made to the program AST**
- **Next, read `ir.ml` to see what kind of instructions are available in our IR (read the comments carefully)**
- **And your have to fill in `translate.ml`**
  - You will **submit only this file**, and the whole code must compile when I copy your file into the skeleton code
  - Complete the implementation of the following function:

    ```
    let run (p: program) : ir_code = ...
    ```

  - FYI, my reference solution is about 200 lines
- **During this, functions in `helper.ml` may be useful**

Structure of IR Code

- **Our IR code will also consist of one function**
- **The IR function also has its name and arguments, along with an instruction list**
  - Generating this instruction list is the key challenge of the project
  - What should we do for the arguments? I will explain later



Three Modes in `main.bin`

- **Once you compile the skeleton code and run it, it will print out the usage as follow**
  - There are total three modes supported in this phase
  - The **`print-ast`** mode is the same as before (phase #1)

```
$ make
$ ./main.bin
<Usage>
[*]./main.bin print-ast <source file>
[*]./main.bin print-ir <source file>
[*]./main.bin run-ir <source file> <input file>
```

Printing the Translated IR

- **Using the `print-ir` mode, you can print out the IR code generated by `run()` defined in `translate.ml`**
  - Provided **`translate.ml`** currently generates dummy IR code
  - The dummy IR code simply returns zero

```
$ cat testcase/prog-1
int f(int i, bool b) {
...

$ ./main.bin print-ir testcase/prog-1
f(i, b) : [
  $r0 = 0,
  ret $r0
]
```

Executing the Translated IR

- **Using the `run-ir` mode, you can run the generated IR code and check its execution result (return value)**
  - Note that you also have to provide **input file together**

```
$ cat testcase/inp-1
30 true
7 false
...

$ ./main.bin run-ir testcase/prog-1 testcase/inp-1
0
0
...
```

Each line means **`main(30,true)`** and **`main(7,false)`** respectively

Input file

What about program errors?

- **You might have recognized that the `check` mode is gone**
- **From this phase, we assume that the input program has passed type checking**
  - Therefore, the input program does not have any error that can be detected in the semantic analysis phase
- **But the program may also have other kind of errors that cannot be detected by semantic analysis**
  - Ex) Division-by-zero, out-of-bound array access
  - What should we do for such programs?
  - **Assume that input program does not have such errors, too**
  - Discussion: Is it safe to make such assumption?

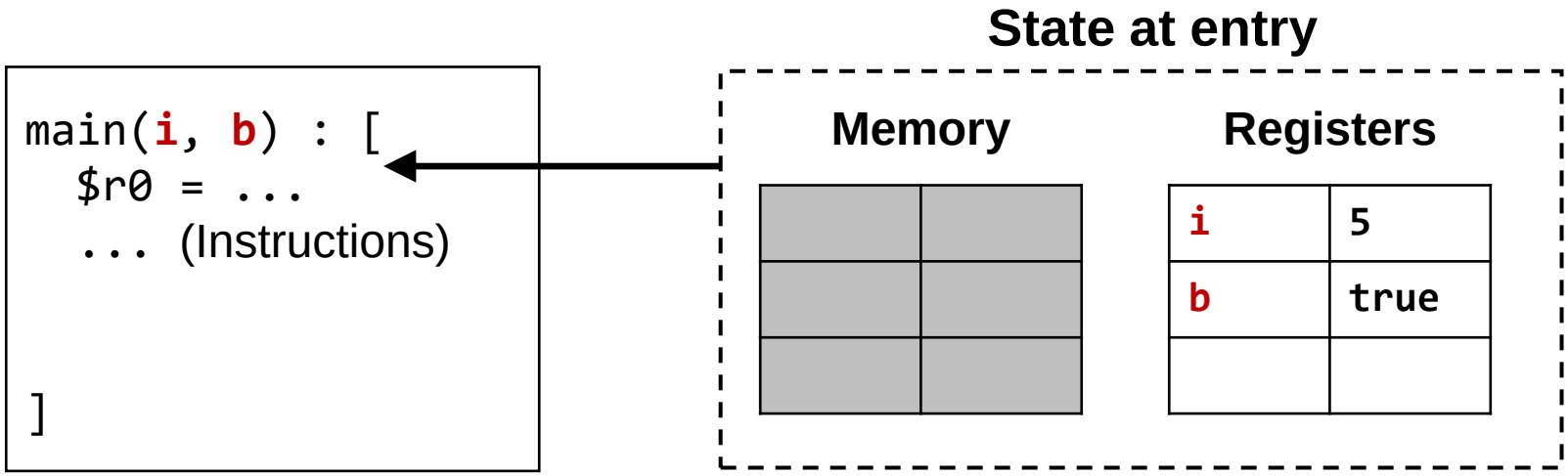
Short-Circuit Code

- **We will assume that our source language defines the semantics of && and || with short-circuit evaluation**
- **Therefore, your translator must generate IR code that implements short-circuit jumps**
- **Consider one of the input program in `testcase/`**
  - This program is error-free, due to the short-circuit semantics
  - Therefore, the IR code generated by your translator must not raise division-by-zero at runtime

```
if (y != 0 && x / y >= 2) {
  return x / y;
}
```

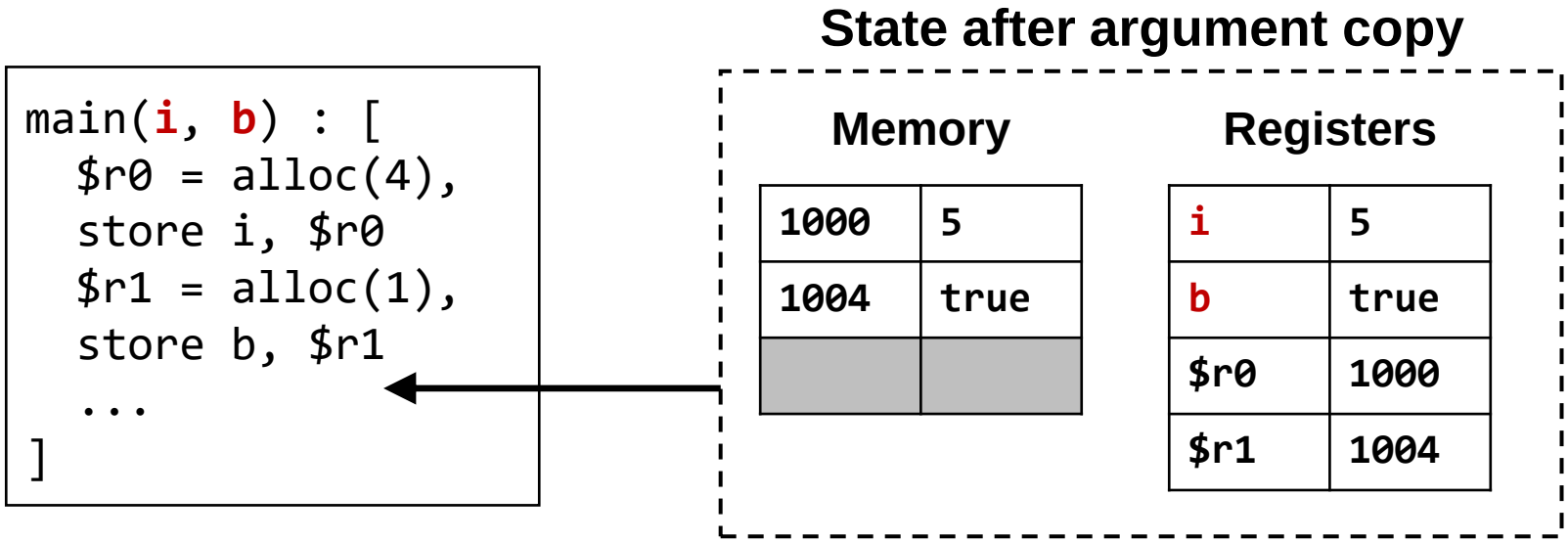
Handling Arguments

- **Function in IR code also takes in arguments**
  - Are these arguments passed through registers or memory?
- **It's a matter of choice: we will follow the LLVM model**
  - Assume that arguments are passed through register
  - At the function entry, argument registers are already defined



IR Translation for Arguments

- **What kind of IR instructions should we generate?**
- **If we copy the argument values into memory, arguments and local variables can be handled in the same way**
  - Example of generated instructions are provided below
  - If you have a better idea, feel free to implement it



Hints (Suggestions)

- **What should be the return type of `trans_exp(`*e*`)`?**
  - Note that the following is just my suggestion (not mandatory)
- **Previously in the lecture note, we defined `trans_exp(`*e*`)` to return a list of instructions**
  - And the last instruction defines a register that contains the result of computing ***e***
- **In the implementation, it would be better to explicitly return a pair of instruction list and register name**
  - Ex) `let (instrs, r) = trans_exp e in ...`
  - Here, `r` is the register that contains the result of computing `e`

Correctness & Performance

- **In this phase, your goal is to generate correct IR code**
  - If the translated IR code returns correct value, you get the point
- **In the next phase (Phase #3. Optimization), I will evaluate the performance of your IR code**
  - You will get higher points if less instructions are executed
- **But of course, you can start early and think about how to generate more efficient IR in your `translate.ml` code**

Self-Grading

- In `testcase/` directory, `prog-N` (program to test), `inp-N` (inputs), and `ans-N` (expected output) are provided
- Output of following command must be same to `ans-N`

```
./main.bin run-ir testcase/prog-N testcase/inp-N
```

- You can also use `check.py` to run all the test cases
  - **Meaning of the result string:** `'O'`: Correct, `'X'`: Incorrect, `'T'`: Timeout, `'E'`: Runtime error, `'C'`: Compile error

Submission Guideline

- **You should submit only one file (be careful not to submit compile by-product files like `*.cmo`)**
  - `translate.ml`
- **Submission format**
  - Upload these files directly to *Cyber Campus* (**do not zip them**)
  - **Do not change the file name** (e.g., adding any prefix or suffix)
  - If your submission format is wrong, you will get **-20% penalty**