Project Phase #1. Type Checking

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

General Information

- **Check "Prj #1" in *Assignment* tab of *Cyber Campus***
  - Skeleton code (**`Prj1.tgz`**) is attached in the post
  - Deadline: **11/22** Wed. 23:59
  - Submission will be accepted in that post, too
  - Late submission deadline: **11/24** Sat. 23:59 **(-20% penalty)**
- **Please read the instructions in this slide carefully**
  - **The specification of the project is quite complex**
  - The slide also contains important submission guidelines
    - If you do not follow the guidelines, you will get penalty

Skeleton Code

- **Copy Prj1.tgz into CSPRO server and decompress it**
  - Don’t decompress-and-copy; copy-and-decompress
  - This course will use cspro**5**.sogang.ac.kr (**don’t miss the "5"**)
- **`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)**

Structure of `src` Directory

- `lexer.mll` : Input file for `OCaml Lex/Flex`
- `parser.mly` : Input file for `OCaml Yacc/Bison`
- `program.ml` : Definition of the AST for program
- `error.ml` : Definition of semantic errors to detect
- `typeCheck.ml` : Type checking (semantic analysis) logic
- `main.ml` : The main driver code
- `Makefile` : The top-level `Makefile` redirects to this one

Where do I have to read and fix?

- **Front-end code (`lexer.mll` and `parser.mly`) is already filled in, and you do not have to care**
- **But you must read `program.ml` and understand the definition of program AST**
- **You should also read `error.ml` and understand what kind errors you must detect**
- **And `typeCheck.ml` is the only file that you have to fix**
  - You will **submit only this file**, and the whole code must compile when I copy your file into the skeleton code

Source Language: mini-C

- As I mentioned in *Chapter 5. Type Checking*, we will use a simplified C (mini-C) as our source language
- In mini-C, a program consists of (1) global variable declarations and (2) function definitions

```
int x;

int y;

int f(void) { ... }

void g(int n) { ... }
```

Key Difference from Original C

- **Cannot declare and initialize variables in one line**
  - Ex) `"int v = 0;"` **: not allowed in our syntax**
- **Declaration of function is not needed**
- **Only `int` / `bool` / `void` types are supported**
- **Only `while` loop is supported**
- **Only basic operators are supported (`+`, `-`, `*`, `/`, …)**
  - Ex) `"x += 1;"`, `"y++;"` **: not allowed in our syntax**

```
int v;

int f(void) { g(); ... }

void g(int n) { v = 0; ... }
```

`g()` is used without declaration

Key Difference from Original C

- **Statement cannot be used as an expression**
  - Ex) `"x = (y = 1) + 1;"` **: not allowed in our syntax**
- **Cannot omit parentheses in `if`, `else`, or `while`**
  - Ex) `"if (b) x = 1;"` **: not allowed in our syntax**
  - Ex) `"while (b) y = y + 1;"` **: not allowed in our syntax**
- **Can only use `bool` type as a condition of `if` or `while`**
  - Ex) `"if (1) { x = 1; }"` **: allowed in syntax, but your type checking should detect this as an error!**

Assumptions on Input Program

- **Following cases are obvious semantic errors, but let's assume our inputs do not contain such cases**
  - Thus, your type checker **does not have to** care about these
- **Redefinition of variable or functions**
  - But note that you can declare local variable that has the same name with a global variable or a function

```
int x;
int x; // Error
void f(int n, int n) { int n; ... } // Error
```

- **Declaring variable as `void` type**

```
void x; // Error
int f(void n) { void l; } // Error
```

Assumptions on Input Program

- **Following case is problematic, but it is hard to precisely detect with semantic analysis**
  - Again, your type checker **does not have to** care about this
  - Even real-world compilers do not catch these errors
- **A function that terminates without encountering any `return` statement (while the return type is not `void`)**

```
int f(int n) {
  int x;
  x = 1;
} // No 'return' statement encountered.
```

Printing Program AST

- **After compiling the project with `make` command, you can print the input program (e.g., `tc-1`) as follow**
  - **`$ ./main.bin print testcase/tc-1`**
  - With this, you can see how the program is parsed into AST

```
Global variables : [int x, int y, int v]
Functions : [
 void f(bool v, int n) : [
    if (v) [
      x = (1 + (2 * n))
    ] else [

    ],
    return
  ],
...
```

Explicitly shows you the precedence of operators

Running Type Checker

- **Next, you can run the type checking as follow**
  - **`$ ./main.bin check testcase/tc-1`**
  - Take a look at **`main.ml`** for details
- **This will print out the errors found in the program**
- **The type checker (`typeCheck.ml`) is not filled in yet**
  - So the command above will print nothing now

Kind of Errors to Detect

- Check `type error` defined in `error.ml` file
- `UndefinedName` : trying to use undefined name
- `AssignMismatch` : mismatch between LHS and RHS
  - Ex) `"int x; x = true;"`
- `ReturnMismatch` : return type mismatch
  - Ex) `"int f(void) { return true; }"`
- `CallingVariable` : using variable name as function
  - Ex) `"int f(void) { int x; x(); }"`
- `UsingFunctionAsVar` : using function name as variable
  - Ex) `"int f(void) { f = 1; }"`

Kind of Errors to Detect

- **`ArgTypeMismatch` : type mismatch in passed argument**

```
int f(bool b) { ... }
void g(void) { f(1); } // Error
```

- **`ArgNumMismatch` : mismatch in argument number**

```
bool f(int n) { ... }
void g(void) { f(1, 2); } // Error
```

- **`OperandMismatch` : type mismatch of operands**
  - Using **`int`** as a condition of **`if`** or **`while`**
  - `+, -, /, *, >, >=, ...` between *non-integers*
  - `&&, ||, !` with *non-Booleans*

What you can and cannot fix

- **In `typeCheck.ml` file, you have to implement the following function**
  - `let run (p: program) : error list = ...`
  - It means **`run`** takes in **`program`** and return a list of **`error`**
  - If you change the type of **`run`**, the project **won't compile!**
- **I already provided some code as a guideline**
  - FYI, my reference solution is about 200 lines
    - So you will have to add about 150 lines of code
  - But this is just my suggestion, and you can choose to delete all the code and write everything from scratch

Order of Error in the List

- **If there are multiple semantic errors, the error found earlier in the program must come first in the list**
- **For the following example program, `run()` must return `[AssignMismatch; ArgNumMismatch]`**

```
bool f(int n) { int x; x = true; }
void g(void) { f(1, 2); }
```

- **For simplification, let's assume that test input programs will only contain one error at maximum per each line**
  - In other words, no multiple errors in a single line

Promise on Tricky Cases

- **This case is `OperandMismatch` (not `AssignMismatch`)**
  - **Reason**: If either LHS or RHS already has an error, let's not check further for the mismatch between them

```
bool f(void) {
  int x;
  x = 1 + true; // Error
}
```

- **This is `ArgNumMismatch` (not `ReturnMismatch`)**
  - **Reason**: If the return value already has an error, let's not check further for the mismatch with the function's return type

```
bool f(void) { return true; }
void g(void) { return f(1, 2); } // Error
```

Self-Grading

- **In `testcase/` directory, `tc-N` (test input) and `ans-N` (expected output of `tc-N`) files are provided**
- **Output of following command must be same to `ans-N`**

```
./main.bin check testcase/tc-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

Tips for OCaml Syntax

- **You may need the following syntax for this phase**
  - But this is just based on my experience; you are not required to use these

```
(* You can append two lists with @ operator. *)
let l = [1; 2; 3] @ [4; 5] (* l is [1; 2; 3; 4; 5] *)

(* Defining two functions that are mutually recursive. *)
let rec f n =
  n + g (n - 1)

and g m =
  if m <= 1 then 1 else m * f (m - 1)
```

Submission Guideline

- **You should submit only one file (be careful not to submit compile by-product files like `*.cmo`)**
  - `typeCheck.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**