Lab 3
Compiler front-end 2/2 : Typing + Symbol Table

Objective

The objective of this session is the construction of the Symbol Table. The Symbol Table will contain all useful information to manage names during the compilation process. In particular, for each name in a given context (main program, other function), we will associate a type, which will be:

- either a basic type : boolean, integer, char, ...
- or a complex type : array of 8 integers, pointer on boolean, ...
- or, a user-defined type, such as structs.

In this session, we will compute all the necessary type information, before constructing one Symbol Table per function. First download the archive (`http://laure.gonnord.org/pro/teaching/capM1.html`) and compile (ignore warnings due to an incomplete management of types).

3.1 API for Type Construction

The files `Type.h/.cc` provide an API for constructing the intermediate representation of such types, and also printing them. The files `SymbolTable.h/.cc` provide a definition of symbol tables.

EXERCISE #1 ▶ **Construction and registration of Types**

- Create a subdirectory `tests` and write syntactically correct C files with type *declarations*. Try to be as exhaustive as possible (do not forget `structs` and pointers). Also write non valid type definitions (non-ending recursivity, ...). Test all these files with your favorite C compiler. [1]
- In the `main` file, write the code to create and print-out the type `char[8]`. Test.
- In `digcc.ypp`, what is the attribute of the non-terminal `type`? Complete the rules of `type` to build correctly the types.
- In `digcc.ypp`, what is the purpose of `add_type($3,$2)` (after line 219 - rule for `type_def`)? Where is this function implemented?
- Add `print_symbols(cout)` to print-out the registered types in the symbols table (Note: Instead of doing it in the `main`, do it inside the rule `prog`, before normalizing the types). Test on your examples. Be aware that this method may not terminate, thus comment out your calls after this exercise.
- The class `Type` owns a method `print_dot` which prints out the `dotty` representation (graph) of the current type (to get a pdf file: `dot -Tpdf test.dot > test.pdf`). You can also use the `show_dot` method. Experiment with 3 or 4 different (compound) types (in the main).

3.2 Type properties

EXERCISE #2 ▶ **Normalization and well-foundedness**

Normalization of types consists in replacing all the identifiers in types by their definitions. This process happens after the last reduction of `type_def_list` (last line of `parser.ypp`).

- Observe the code of `normalize_type` and `normalize_types` in `SymbolTable.cpp`. What is the representation of types after normalization? Check it using pretty-print at adequate program points.
- Give the representation after normalization of the type `list_t`. Check that it is correct using `print_dot` or `show_dot`.
- Explain how the normalization of `list_t` is done in `normalize_type` :

```
typedef struct {
  int element;
```

[1] Be careful with array declarations, the parser accepts `int[10] t;` and rejects `int t[10];`

```
  list_t* next;
} list_t;
```

- Give an example of type rejected by `is_well_formed`. Why do we want to reject those types?

EXERCISE #3 ▶ **Type equivalence**

- When is type equivalence needed in the compilation process?
- Sketch the algorithm to decide equivalence of two types based on their representation after normalization.
- Open `Type.cpp` (line 142), and implement the type equivalence (*bisimulation*).
- Test!

3.3 Function types : construction/check

Each time a function is declared (`parser.ypp` line 478), its signature is added to the symbol table. A call to `add_function()` creates a new (signature of the) current function. Then, `add_argument_type()` adds the types of the arguments. `add_argument()` declares an argument and `add_local_var` declares a new local variable. Then, these informations are used to type the expressions inside the function body.

EXERCISE #4 ▶ **Function types**
In `parser.ypp`:

- Inspect and explain the rules of `function`, `declare_args`, `declare_local_vars`.
- Inspect and explain the rules for `stmt`. How do we manage the assigment polymorphism?
- Explain the rule for the `return`.
- How are procedure handled (return type, type check, ...) ?
- Complete the rules of the non-terminal `e_expr` to control the types. *Use the* `type_check` *function implemented in* `SymbolTable.cpp`.
- Carefully test types for expressions and functions.

3.4 Symbol Table

EXERCISE #5 ▶ **Symbol Table**
The Symbol Table gives all the useful information of a given function, or the whole program. Modify the given code to print all the program information on the standard output. Expected output for a function:

```
********************
Function declaration : toto
Types
mytab_t: (int)[256]

Arguments
plouf (of type my_tab_t) --> r0

Local variables
tmp (of type char) --> r0
********************
```