Assignment 3: Translation

COMP 3010 – Organization of Programming Languages
March 2020

[**Note:** *This assignment is adapted from the original version authored by Prof. Michael Scott.*]

Problem description

Your task in this assignment is to implement a complete (if simplistic) compiler for the extended version of the calculator language, again with `if` and `do`/`check` statements. Your compiler (which we will call the "translator") will be written in OCaml and will generate C code.

We are providing you with a complete parser generator and driver (in the file "`parser.ml`") that build an explicit parse tree. The provided code also includes the skeleton (stubs) of a possible solution (in the file "`translator.ml`") that converts the parse tree to an abstract syntax tree (AST), and then recursively "walks" the AST to produce the translation into C.

The provided parser code has two main entry points:

```
get_parse_table : grammar -> parse_table = ...
parse : parse_table -> string -> parse_tree = ...
```

The first of these functions returns a parse table, in the format expected as the first argument of the second function. The second function returns a parse tree. (Two example parse trees are included as text files in the code directory provided for the assignment.) If the program has syntax errors (according to the grammar), `parse` will print an error message (as a side effect) and return a `PT_error` value (it does not do error recovery).[1]

The grammar takes the form of a list of production sets, each of which is a pair containing the LHS symbol and *k* right-hand sides, each of which is itself a list of symbols. When `get_parse_table` builds the parse table, the grammar is augmented with a start production that mentions an explicit end of file `$$`. Later, `parse` will remove this production from the resulting parse tree.

[1] If the input grammar provided to the parser generator is malformed, it may produce unhelpful run-time errors—it isn't very robust.

The format of the grammar for the extended calculator language looks like this:

```
let ecg : grammar =
    [ ("P",  [["SL"; "$$"]])
    ; ("SL", [["S"; "SL"]; []])
    ; ("S",  [ ["id"; ":="; "E"]; ["read"; "id"]; ["write"; "E"]
             ; ["if"; "R"; "SL"; "fi"]; ["do"; "SL"; "od"]
             ; ["check"; "R"]
             ])
    ; ("R",  [["E"; "ET"]])
    ; ("E",  [["T"; "TT"]])
    ; ("T",  [["F"; "FT"]])
    ; ("F",  [["id"]; ["num"]; ["("; "E"; ")"]])
    ; ("ET", [["ro"; "E"]; []])
    ; ("TT", [["ao"; "T"; "TT"]; []])
    ; ("FT", [["mo"; "F"; "FT"]; []])
    ; ("ro", [["=="]; ["<>"]; ["<"]; [">"]; ["<="]; [">="]])
    ; ("ao", [["+"]; ["-"]])
    ; ("mo", [["*"]; ["/"]])
    ];;
```

A program is just a string:

```
let sum_ave_prog = "
   read a
   read b
   sum := a + b
   write sum
   write sum / 2";;
```

Your work will proceed in two steps:

1. Translate the parse tree into a syntax tree:

   ```
   let rec ast_ize_P (p:parse_tree) : ast_sl = ...
   ```

   where the single argument is a parse tree generated by function `parse`. We have provided a complete description of the `ast_sl` type in "`translator.ml`".

2. Translate the AST into an equivalent program in C:

   ```
   let rec translate (ast:ast_sl) : string * string = ...
   ```

   where the argument is a syntax tree as generated by function `ast_ize_P`, and the return value is a tuple containing a pair of strings. The first string, which will usually be empty, indicates (as a nicely-formatted, human-readable error message) the names of any variables

that are assigned to (or read) but never used in the program. This is as close as we get to static semantic error checking in this very tiny language.

Putting the pieces together, you might write:

```
let (my_warnings, my_C_prog) =
  translate (ast_ize_P (parse ecg_parse_table my_prog));;
```

Note: *Your OCaml program should **not** take advantage of any imperative language features. You may create testing code that uses* `print_string` *and related functions, and you may keep the code that prints an error message if the input program in the extended calculator language contains a syntax error, but the main logic of your syntax tree construction and translation should be purely functional.*

As noted in the previous assignment, the addition of `if` and `do`/`check` to the calculator language gives it significant computing power. If we let `primes_prog` be a string containing the primes-generating program from that assignment (also included in the starter code), and then type:

```
print_string (snd
  (translate (ast_ize_P
    (parse ecg_parse_table primes_prog))));;
```

you should see, on standard output, a C program which, when compiled, run, and fed the input 10, prints:

```
2
3
5
7
11
13
17
19
23
29
```

For the sake of convenience, we recommend that your output program always begin with definitions of two helper functions:

```
#include <stdio.h>
#include <stdlib.h>

int getint() {
  ... // returns an integer from standard input or
      // prints an appropriate error message and dies.
}

void putint(int n) {
  ... // prints an integer and a linefeed to standard output.
}
```

As noted above, your translator is required to give a warning if the input program assigns to any variable that is never used. (You do not have to detect whether the program contains a variable use that will never be executed—only the lack of any use at all.)

The C program you generate is required to catch the following dynamic semantic errors, any of which will cause it to terminate early (with a helpful error message). Note that some of these errors are not caught by default in C. Your program will have to include extra code to catch them.

- **unexpected end of input** (attempt to read when there's nothing there)
- **non-numeric input** (the extended calculator language accepts only integers)
- **use of an uninitialized variable**—one to which a value has not yet been assigned (`read`-ing counts as assignment)
- **divide by zero** (**Note**: C's automatically generated "floating exception" is not very helpful; ideally you want an error message such as "divide by zero at line 23". You should write your code so that *if you were tracking line numbers*, you could easily include the line number in the error message. That is, your generated code should check explicitly to make sure that denominators are not equal to zero (even though your error message does **not** have to include the line number.)

Suggestions

For most of the assignment, it will probably be easiest to use the `ocaml` interpreter. You'll want to keep reloading your source code (`#use "translator.ml"`) as you go along, so you catch syntax and type errors early. On occasion, you may also want to try compiling your program with `ocamlc`, to create a stand-alone executable.

In "`translator.ml`", we have provided code for the sum-and-average and primes-generating calculator programs. You will undoubtedly want to write more calculator programs for purposes of debugging.

We will be grading your assignment using the OCaml interpreter in the Ubuntu VM. You can download your own copy of Ocaml for Windows, MacOS, or Linux, but please be sure to allow ample time to check that your code works correctly on the VM installation.

As a rough guess, you should be able to write a complete implementation of `ast_ize_P`, `translate`, and everything they call in less than 150 lines of code.

Besides the resources mentioned already during class, you may find the following helpful:

- **OCaml system documentation**—Includes the language definition; manuals for the interpreter, compilers, and debugger; standard library references; and other resources: <http://caml.inria.fr/pub/docs/manual-ocaml/>
- **Tutorials** on various aspects of the language: <http://ocaml.org/learn/tutorials/>
- **OCaml for the Skeptical**—An alternative, arguably more accessible introduction to the language from the University of Chicago: <http://www2.lib.uchicago.edu/keith/ocaml-class/home.html>
- **Developing Applications with Objective Caml**—A book-length online introduction to functional programming and everything OCaml: <http://caml.inria.fr/pub/docs/oreilly-book/html/index.html>
- **Real World OCaml, 2nd Edition**—A free online version of a popular print book on OCaml; we recommend Chapter 1 ("A Guided Tour") as a good overall introduction, and the rest of the book is an excellent reference with lots of examples: <http://dev.realworldocaml.org/toc.html>

Division of labor and submission procedure

You may work alone on this project, or in teams of two or three. Be sure your write-up (README file) describes any features of your code that the TA might not immediately notice – for example, how to run your code (e.g., in the interpreter, or creating a stand-alone executable).

If you choose to work in pairs/triples, we strongly encourage you to read each other's code, to make sure you have a full understanding of semantic analysis. The most obvious division of labor for a 2-person team is for one team member to write `ast_ize_P` (and its associated functions), and the other to write `translate` (and its associated functions), but other divisions are fine as well.

To turn in your code, use the following procedure, which will be the same for all assignments this semester:

1. Your code should be in a directory called "`<YourName>_OPL_A3`" (for example, "`TomWilkes_OPL_A3`"). Put your write-up in a `README.txt` or `README.pdf` file in the same directory as your code.
2. In the parent directory of your A3 directory, create a tarball (`.tar.gz`) or Zip file that contains your A3 directory (e.g., "`TomWilkes_OPL_A3.tar.gz`").
3. In the page for Assignment 3 on Blackboard, use the Submit button to upload your tarball or Zip file. When you submit, give the name(s) of your partner(s) (if applicable) in the submission comments field.