Project 3: translation to functional form

October 12, 2017

1 Project description

In this project, your task will be to build a small tool to convert a C-like function body into functional form.
We assume you are familiar with the `let .. = .. in ...` *binding* constructs that are used in many functional languages (Racket, Caml, ...). Your goal is to transform a block of code of a C-like language into a functional program using `let .. = .. in ...` constructs.
You first have to identify different sets of variables:

- undefined in the code block. They are then considered arguments of the function.
- defined in the block.
- written in the block, we refer to them as *state variables*. Your function should output these variables.

Then your program should build the function equivalent to the code block. In the functional language, you should use the same expression grammar as the input language and the binding constructs. However you are free to define your own functional language with your own syntax, providing it uses *binding* constructs. The only restriction is that all conditionals should be balanced. To handle *iteration statements* you will also need *recursive binding* constructs, such as the `let rec .. = .. in ..` from Caml.

The ouput should contain:

- the function prototype, with the arguments variables. The name is not important.
- the body of the function, in functional form.

Deliverables At the end of the project, you will be asked to provide the **source code** of your tool. It should be well commented and easy to understand. We also should be able to test your tool on small programs, so it should be compilable on the Teaching Lab's machines.

Additionally, we would like to have **a small report** (2-3 pages), explaining:

- the imperative-to-functional translation process.
- how-to use your tool.
- the syntax of your functional language (you don't need to specify the expression syntax, as it should be the same as the input language).

Notation The input language is quite rich and you might not want to consider all of it when you start. Particularly, handling *iteration-statements* might not be straightforward. 20 points will be awarded if you succeeded in capturing the different set of variables you need to define the function prototype. 50 points for a correct translation to functional form for an input language without iteration statements. If you can also handle the latter, you will get 30 points.

The report is mandatory, because it helps us to understand your approach to solve all the previous problems.

Bonus All the loops in the input language can be translated into recursive functions. However, with some additional analysis, you should be able to identify widely used functional constructs such as *fold*. Try to add this to your tool!

2 Language

For this project your are invited to any language of your choice. It should however be easy for you to perform the program analysis and transformations.

3 Input program grammar

This is a subset of the grammar defined in ISO/IEC 9899:TC3.

We will use the list construction with any syntax object, it refers to an empty list or a comma-separated list of the syntax objects:

⟨*List X*⟩ ::= ⟨*empty*⟩
 | ⟨*NonEmptyList X*⟩

⟨*NonEmptyList X*⟩ ::= ⟨*X*⟩
 | ⟨*X*⟩ ’,’ ⟨*List X*⟩

The syntactic element specified in brackets (ex: {’+’, ’-’} means one of the contained elements can be used to produce the syntax object.

3.1 Expressions

This is inspired from the C99 standard with some restrictions:

- there are no *assignment-expressions*. For example, the following expression is illegal in our subset:

  ```
  char c;
  int i;
  long l;
  l=(c=i);
  ```

- The only pointers are used in array subscripts, therefore we removed address and indirection operators **&** and .

To summarize the features of the expressions of the language: we have function calls, array subscripts, unary expressions and binary expressions.

⟨*primary-expression*⟩ ::= ⟨*identifier*⟩
 | ⟨*constant*⟩
 | ⟨*string-literal*⟩
 | ’(’ ⟨*expression*⟩ ’)’

⟨*postfix-expression*⟩ ::= ⟨*primary-expression*⟩
 | ⟨*postfix-expression*⟩
 | ⟨*postfix-expression*⟩ ’[’ ⟨*expression*⟩ ’]’ // Array access
 | ⟨*postfix-expression*⟩ ’(’⟨*List expression*⟩] ’)’ // Function call (uninterpreted function)

⟨*unary-expression*⟩ ::= ⟨*postfix-expression*⟩
| ’+’ | ’-’ ⟨*unary-expression*⟩

⟨*multiplicative-expression*⟩ ::= ⟨*unary-expression*⟩
| ⟨*multiplicative-expression*⟩ { ’*’ ’/’ ’%’ } ⟨*unary-expression*⟩

⟨*additive-expression*⟩ ::= ⟨*unary-expression*⟩
| ⟨*additive-expression*⟩ {’+’ ’-’ } ⟨*multiplicative-expression*⟩

⟨*shift-expression*⟩ ::= ⟨*additive-expression*⟩
| ⟨*shift-expression*⟩ { ’<<’ ’>>’ } ⟨*additive-expression*⟩

⟨*relational-expression*⟩ ::= ⟨*shift-expression*⟩
| ⟨*relational-expression*⟩ {’<’ ’>’ ’<=’ ’>=’ } ⟨*shift-expression*⟩

⟨*equality-operators*⟩ ::= ⟨*relational-expression*⟩
| ⟨*equality-expression*⟩ { ’==’ ’!=’ } ⟨*relational-expression*⟩

⟨*AND-expression*⟩ ::= ⟨*equality-expression*⟩
| ⟨*AND-expression*⟩ ’&&’ ⟨*equality-expression*⟩

⟨*OR-expression*⟩ ::= ⟨*AND-expression*⟩
| ⟨*OR-expression*⟩ ’||’ ⟨*AND-expression*⟩

⟨*conditional-expression*⟩ ::= ⟨*OR-expression*⟩
| ⟨*OR-expression*⟩ ’?’ ⟨*expression*⟩ ’:’ ⟨*conditional-expression*⟩

⟨*expression*⟩ ::= ⟨*conditional-expression*⟩

3.2 Declarations

⟨*declaration*⟩ ::= ⟨*type-specifier*⟩ ⟨*List init-declarator*⟩

⟨*init-declarator*⟩ ::= ⟨*declarator*⟩
| ⟨*declarator*⟩ ’=’ ⟨*initializer*⟩

⟨*declarator*⟩ ::= ⟨*identifier*⟩
| ’*’ ⟨*declarator*⟩
| ⟨*identifier*⟩ ’[’ ⟨*constant*⟩ ’]’

We use the following type specifiers - or *basic types* :

⟨*type-specifier*⟩ ::= ’void’
| ’int’
| ’float’
| ’_Bool’

3.3 Statements and blocks

⟨*statement*⟩ ::= ⟨*compound-statement*⟩
| ⟨*expression-statement*⟩
| ⟨*selection-statement*⟩
| ⟨*iteration-statement*⟩

⟨*compound-statement*⟩ ::= ’’ ⟨*block-item-list*⟩ ’’

⟨*block-item-list*⟩ ::= ⟨*block-item*⟩
| ⟨*block-item*⟩ ⟨*block-item-list*⟩

⟨*block-item*⟩ ::= ⟨*declaration*⟩ ’;’
| ⟨*statement*⟩

⟨*expression-statement*⟩ ::= ⟨*expression-assign*⟩ ';'

⟨*expression-assign*⟩ ::= ⟨*expression*⟩
  | ⟨*expression*⟩ { '=' '+=' '-=' '&=' '||=' } ⟨*expression*⟩
  | ⟨*expression*⟩ { '++' '–' }
  | { '++' '–' } ⟨*expression*⟩

⟨*selection-statement*⟩ ::= 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
  | 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ 'else' ⟨*statement*⟩

⟨*iteration-statement*⟩ ::= 'while' '(' expression ')' ⟨*statement*⟩
  | 'for' '(' ⟨*expression-statement*⟩ ';' ⟨*expression*⟩ ';' ⟨*expression-statement*⟩ ')' ⟨*statement*⟩

3.4 Program

A program is simply a *block-item-list* in this case.