Pre-Processing

António Menezes Leitão

April, 2018

1 Introduction

The goal of this project is to develop a language-independent pre-processor that is programmable in Racket. Just like CPP and M4, the pre-processor reads input files and writes output files, operating some transformations on those files. The big difference between this pre-processor and other pre-processors comes, on one hand, from the model used to process the transformations and, on the other hand, from the programming language used to define those transformations.

1.1 Active Tokens

The pre-processor transformations are defined in Racket and have a different flavour than macros common in other pre-processors such as CPP. Here, the "macros" are defined upon the concept of *active token*. An active token is just a sequence of characters that is recognized by the pre-processor and that triggers an action. The action is just a plain Racket function that accepts a string as an argument. It is the function's responsability to read whatever it wants from that string and to return a string containing the transformed contents. Without token definitions, the pre-processor simply returns the same string that was used as input.

Another feature of this pre-processor that distinguishes it from traditional ones is that it re-pre-processes the string as many times as needed until no more token activations occur.

1.2 Examples

Let's start by giving an example of an active token: we will consider the introduction of Racket's comments in other programming languages.

In Racket, the semicolon `;` character causes the reader to ignore all characters until the end of line. C++ and Java already have a similar type of comment (using `//`) but we will consider the introduction of a new one for "private" comments, so that code can be shared without sharing also those comments. To avoid clashes with the usual programming language line terminators or separators, these comments will be indicated by preceeding them with the double semicolon `;;`. Here is an example of a string containing this token:

```
//Another great idea from our beloved client
;;This is stupid but it's what the client wants
for(int i = 0; i < MAX_SIZE; i++) {
```

After, pre-processing, the previous string becomes:

```
//Another great idea from our beloved client
for(int i = 0; i < MAX_SIZE; i++) {
```

To implement this active token, we start by writing a function that takes a string as an argument and returns the rest of the string after the newline character or an empty string if it cannot find it.

```
(define (string-after-newline str)
  (or (for/or ((c (in-string str))
               (i (in-naturals)))
        (and (char=? c #\newline)
             (substring str (+ i 1))))
      ""))
```

Finally, we attach this action to the character sequence `;;` using the `add-active-token` function:

```
(add-active-token ";;" string-after-newline)
```

To simplify programmer's life, the pre-processor must also include a Racket macro `def-active-token` that simplifies active token definitions by merging the previous two steps. Using this macro, the previous example could simply become:

```
(def-active-token ";;" (str)
  (or (for/or ((c (in-string str))
               (i (in-naturals)))
        (and (char=? c #\newline)
             (substring str (+ i 1))))
      ""))
```

Naturally, we can define these active tokens using all the machinery available in Racket. Here is a slightly shorter definition:

```
(def-active-token ";;" (str)
  (match (regexp-match-positions "\n" str)
    ((list (cons start end)) (substring str end))
    (else "")))
```

As a second example, we will present the `//eval` active token that opens the full power of Racket to the pre-processing phase:

```
(define ns (make-base-namespace))

(def-active-token "//eval " (str)
  (call-with-input-string
   str
   (lambda (in)
     (string-append (~a (eval (read in) ns))
                    (port->string in)))))
```

Here is an example of its use in a Java program:

```
if (curYear > //eval (date-year (seconds->date (current-seconds)))) {
  ...
```

After pre-processing the program (and admiting this was done on the year 2018), the result will be:

```
if (curYear > 2018) {
    ...
}
```

2 Active Token Definitions

Besides implementing the pre-processor, you also need to implement the following useful active tokens using the `def-active-token` macro.

2.1 Local Type Inference

Local type inference allows for simpler variable declarations. To better understand this concept, consider the following example in Java:

```
HashMap<String,Integer> x = new HashMap<String,Integer>();
```

Java 10 simplifies this declaration via the `var` keyword, allowing us to write:

```
var x = new HashMap<String,Integer>();
```

and the compiler will infer the correct type for the variable.

However, instead of waiting 22 years for this feature, we could have been using a poor man's approach just by defining `var` as an active token. Given that we don't have access to non-explicit type information, we will restrict the use of the `var` token to the variable declarations that are immediately initialized with a constructor call, i.e., an expression that starts with the `new` keyword, as in the previous example.

2.2 String Interpolation

String interpolation is a useful feature that is available in several programming languages but, unfortunately, is missing in Java. String interpolation allows us to *inject* Java expressions in the middle of a string. Here is an example:

```
static void foo(int a, int b, int c) {
    String str = #"First #{a}, then #{a+b}, finally #{b*c}.";
    System.out.println(str);
}
```

Here, the active token is the caracter `#`, which triggers the processing of the immediately following string, so that the post-processed code becomes:

```
static void foo(int a, int b, int c) {
    String str = "First " + (a) + ", then " + (a+b) + ", finally " + (b*c) + ".";
    System.out.println(str);
}
```

2.3 Type Aliases

The `typedef` operator that exists in the language C and C++ allows the definition of new types that symplify the use of complex types. This is a powerful feature that is frequently requested in Java. For example, the following fragment of code is too difficult to write and read:

```
public static ConcurrentSkipListMap<String,List<Map<String,Object>>>
  mergeCaches(ConcurrentSkipListMap<String,List<Map<String,Object>>> a,
              ConcurrentSkipListMap<String,List<Map<String,Object>>> b) {
    ConcurrentSkipListMap<String,List<Map<String,Object>>> temp =
        new ConcurrentSkipListMap<String,List<Map<String,Object>>>();
    ...
    }
}
```

However, assuming the existence of an `alias` operator, we could write instead:

```
alias Cache = ConcurrentSkipListMap<String,List<Map<String,Object>>>;

public static Cache mergeCaches(Cache a, Cache b) {
    Cache temp = new Cache();
    ...
}
```

Note that the pre-processor must also be capable of combining the two previous active tokens, as in:

```
alias Cache = ConcurrentSkipListMap<String,List<Map<String,Object>>>;

public static Cache mergeCaches(Cache a, Cache b) {
    var temp = new Cache();
    ...
}
```

3 Goals

The main goal of this project is the implementation of the pre-processor in Racket, as well as the definition of the previously described active tokens, namely, for local type inference, for string interpolation, and for type aliasing.

To simplify your task, you will not have to deal with processing the files yourself. Your task is to create a pre-processor that takes a string as an argument and returns a string as a result. More specifically, you need to define the functions:

- `add-active-token`, that takes a string describing an active token and a function that should be triggered when that token is found. This last function takes a string and returns a string. The function `add-active-token` stores the association between the token and the corresponding function.

- `process-string`, that takes a string and returns a string and is responsible for triggering the active token actions, transforming the string until no more active tokens are found. To this end, whenever an active token is found, the associated function is called with the substring that starts immediately after the token, and the result of that function replaces the substring that starts with the token.

Moreover, to simplify the definition of active tokens, you also need to define the macro:

- `def-active-token`, that takes an active token, a parameter list, and a body and that expands into an appropriate use of the function `add-active-token`.

To simplify testing, you should include all the previous definitions (functions, macro, and active tokens) in the same Racket file named `preprocess.rkt`. This file should start with the lines:

```
#lang racket
(provide add-active-token def-active-token process-string)
```

3.1 Extensions

You can extend your project to further increase your grade above 20. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Examples of interesting extensions include:

- Definition of active token for generation of getters and setters for Java.
- Definition of an `#include` active token to support, in other languages, the equivalent of the C language `#include` macro.
- Definition of active tokens that operate similarly to C macros, i.e., they take arguments and produce a string that replaces the macro call.
- Definition of an active token that allows the definition of active tokens in the file that is going to be processed.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections. In order to ensure this behavior, you should implement all your extensions in a different file named `preprocessextra.rkt`.

4 Code

Your implementation must work in Racket 6.7.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

5 Presentation

For this project, a full report is not required. Instead, a public presentation is required. This presentation should be prepared for a 15-minute slot, should be centered in the architectural decisions taken and may include all the details that you consider relevant. You should be able to "sell" your solution to your colleagues and teachers.

6 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named as `project.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- The source code file `preprocess.rkt`
- In case you included extensions, the extensions file `preprocessextra.rkt`

The only accepted format for the presentation slides is PDF. This PDF file must be submitted using the Fénix Portal separately from the ZIP file and should be named `p2.pdf`.

7 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

The public presentation of the project is a compulsory evaluation moment. Absent students during project presentation will be graded zero in the entire project.

8 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

9 Final Notes

Don't forget Murphy's Law.

10 Deadlines

The code must be submitted via Fénix, no later than 19:00 of **May**, **11**. Similarly, the presentation must be submitted via Fénix, no later than 19:00 of **May**, **11**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work and the presentation must not exceed 15 minutes. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may question any member of the group before, during, and after the presentation.