PS1: Compilers and Parsing Combinators

Avi Hayoun, Shahaf Shperberg and Lior Zur-Lotan

November 2, 2017

Contents

1 Introduction ................................................. 1
   1.1 Course communication policy .................. 2
   1.2 Scheme implementation ....................... 2
      1.2.1 Regarding Racket ..................... 2

2 Compiler structure ......................................... 2

3 Parsing Combinators ....................................... 3
   3.1 A very brief refresher on parsers ............ 3
   3.2 Parsing Combinators .......................... 4
   3.3 Example: Simple arithmetic expressions .... 4
      3.3.1 Plan ................................ 4
      3.3.2 The operator parser .................. 5
      3.3.3 The digit parsers .................... 5
      3.3.4 The natural numbers parser .......... 6
      3.3.5 The arithmetic expression parser .... 6
      3.3.6 The arithmetic expression parser - fixed 7
   3.4 *pack and *pack-with ....................... 7

1 Introduction

- Course homepage: http://www.cs.bgu.ac.il/~comp181/
  - Useful Resources: http://www.cs.bgu.ac.il/~comp181/Useful_Resources
- Course email: comp181@cs.bgu.ac.il
1.1 Course communication policy

All course communication will be done only via the course email. This way we can make sure no email will be lost within the staff’s personal mailbox.

1.2 Scheme implementation

The version of Scheme used in this course is Chez Scheme. The Useful Resources page contains instructions for installing Chez Scheme.

1.2.1 Regarding Racket

Racket and Scheme are not the same. Although very similar to Scheme, Racket is a different language, and it behaves differently in certain situations (e.g. some mutable structures and parameter passing order).

We’re going to grade your assignments on Chez Scheme on the Linux image on the departmental lab computers. If your codes doesn’t work in this setting, you will get no points(!) for that work (even if it works perfectly on Windows/Mac or Racket/other languages).

Also note, if you use Dr. Racket (even with the #lang scheme directive), it might behave incorrectly due to implementation differences (parameter passing order for one).

To make it perfectly clear, You are expected to test your code in Chez Scheme on the Linux image of the departmental lab computers before you submit anything.

2 Compiler structure

A compiler is a computer program that generates code base on the given input. More specifically, it transforms source code written in a source programming language into code in a target programming language. The classic structure of a compiler:
• **Scanner** (*lexical analysis*) - converts a sequence of characters into a sequence of tokens.

• **Parser** (*syntactic analysis*) - checks for correct syntax and builds a hierarchical structure (abstract syntax tree).

• **Semantic Analyzer** - adds semantic information to the abstract syntax tree and builds the symbol table. This phase performs semantic checks such as type checking, object binding etc.

• **Code Generator** - process of converting the intermediate representation of the source code, provided by the semantic analyzer, into another computer language.

3 Parsing Combinators

The parsing combinators library and a DIY tutorial for it are up on the course website under Useful Resources. You are strongly encouraged to work through the tutorial, as it includes additional examples and more in-depth explanations on some topics covered in this class.

3.1 A very brief refresher on parsers

We’ll cover parser in greater depth later in the course, but for the time being, just remember that the computational model of parsers is the stack automaton. So, first and foremost, parsers decide whether their input belongs to their Context Free Grammar. Secondly, parsers break up their input into chunks ordered in a hierarchical
structure - the Abstract Syntax Tree (or Parsing Tree).

The parser you will write in this course (as part of the compiler) will be written using parsing combinators.

### 3.2 Parsing Combinators

Parsing Combinators are higher-order functions which take parsers as their input and construct new parsers as their output. Note that a parser is simply a function that accepts tokens (or strings in our case) as input, and returns some hierarchical structure as its output.

**Remark.** What this means in essence is that when creating a parser using parsing combinators, we will start by constructing very small parsers, with trivial grammars (single tokens, or a small set of tokens at most). We will then define more complex parsers by iteratively composing the simpler parsers. An example follows.

### 3.3 Example: Simple arithmetic expressions

In this example, we will construct a parser for a simple grammar using parsing combinators. The grammar will be of chained summation and subtraction of numbers, e.g. \(100 - 5 + 72 - 32 - 1 + 0 + 7\).

**Remark.** As we will construct the parser using parsing combinators, we will want to define only the most basic production rules of the grammar explicitly.

#### 3.3.1 Plan

1. Define a parser for operations (+ or −).
2. Define a parser for digits.
3. Define a parser for numbers (we will stick to natural numbers for simplicity’s sake).
4. Define a parser for expressions.

Along the way, we will make sure to output valid scheme code which will evaluate to the correct result.

**Remark.** For simplicity’s sake, the parser will assume right associativity of the arithmetical expression, so \(1 + 2 - 3\) will evaluate to \(1 + (2 - 3)\).
3.3.2 The operator parser

\[ \langle \text{op} \rangle \rightarrow '+', '-' \]

1 (define \langle \text{op} \rangle
2 (new ; initialize a parser stack
3
4 (*parser (char #\+)) ; create and push a parser for the '+' operator
5 (*parser (char #\-)) ; create and push a parser for the '-' operator
6
7 (*disj 2) ; pop top 2 parsers, and push their disjunction
8
9 (*pack (lambda (op-char) ; transform the output
10 (string->symbol (string op-char)))
11
12 done)) ; finalize the stack and return the top-most parser

3.3.3 The digit parsers

\[ \langle \text{digit-0-9} \rangle \rightarrow '0' | \ldots | '9' \]

1 (define \langle \text{digit-0-9} \rangle
2 (range #\0 #\9))

The range function is a "shortcut" of a process similar to that we used to define the \langle \text{op} \rangle parser.

\[ \langle \text{digit-1-9} \rangle \rightarrow '1' | \ldots | '9' \]

1 (define \langle \text{digit-1-9} \rangle
2 (range #\1 #\9))
3.3.4 The natural numbers parser

\(<\text{nat}>\) \rightarrow \text{'}0\text{'} \mid \text{<digit-1-9>\text{<digit-0-9>}\text{*}}

\begin{verbatim}
1 (define <nat>
2  (new
3  (*parser (char #\0)))
4  (*pack (lambda (_) 0))
5
6  (*parser <digit-1-9>)
7  (*parser <digit-0-9>) *star
8  (*caten 2)
9
10  (*pack-with
11    (lambda (x xs)
12      (string->number (list->string '\(,x ,@xs)))\))
13
14  (*disj 2)
15  done))
\end{verbatim}

3.3.5 The arithmetic expression parser

\(<\text{expr}>\) \rightarrow \text{<expr>\text{<op>\text{<nat>}} \mid \text{<nat>}}

\begin{verbatim}
1 (define <expr>
2  (new
3
4  (*delayed (lambda () <expr>)) ; allow recursion
5  (*parser <op>)
6  (*parser <nat>)
7
8  (*caten 3)
9
10  (*pack-with (lambda (n op expr)
11    '\((,op ,n ,expr)))
12
13  (*parser <nat>)
14
15  (*disj 2)
16  done))
\end{verbatim}

Remark. Note the use of the \textit{*delayed} combinator to allow recursion. Since the incomplete definition of \textit{<expr>} cannot use itself, we must delay the evaluation of the
reference to a later time (after the definition is complete). The *delayed combinator allows us to do exactly that.

**Left recursion:** Immediate left recursion occurs in rules of the form \( A \rightarrow A\alpha | \beta \) where \( \alpha \) and \( \beta \) are sequences of nonterminals and terminals, and \( \beta \) doesn’t start with \( A \).

The production \(<\text{expr}\> \rightarrow <\text{expr}><\text{op}><\text{nat}>\) is a *left recursive* production rule. In parsers that work from left to right (such as the one we are defining in the example above), such production rules cause an infinite loop. We will discuss left recursion in more depth in a future class. For now, it is enough to realize that in the case of our grammar, the result is the same if we make the rule *right recursive* instead: \(<\text{expr}\> \rightarrow <\text{nat}><\text{op}><\text{expr}>\).

### 3.3.6 The arithmetic expression parser - fixed

\(<\text{expr}\> \rightarrow <\text{nat}><\text{op}><\text{expr}> | <\text{nat}>\)

1 (define <expr>
2  (new
3
4  (*parser <nat>)
5  (*parser <op>)
6  (*delayed (lambda () <expr>))
7
8  (*caten 3)
9
10  (*pack-with (lambda (n op expr)
11    ',(op ,n ,expr)))
12
13  (*parser <nat>)
14
15  (*disj 2)
16 done))

### 3.4 *pack and *pack-with

The *pack and *pack-with functions are also parsing combinators, meaning they take a parser as input and compose it with some extra functionality. Unlike the other combinators (such as *caten and *disj), the packing combinators don’t compose multiple parsers together, but rather allow us to transform the output of a parser before returning it.
As an example, consider the code (**pack (lambda (_) 0)) from the <nat> parser. In this case, we are taking whatever is returned from the parser (char #\0) and returning the number 0 instead.

The purpose of **pack and **pack-with are identical, but they provide two subtly different interfaces:

- The function given to **pack must take a single argument.
- The function given to **pack-with may take any number of arguments.

In most cases, it will make sense to use **pack-with with parsers defined using *caten and **pack for most other parsers, but this is not a rule set in stone. A good suggestion would be to write the data transformation function first, and select whichever packer is most appropriate for it.