PS1: Compilers and Parsing Combinators

Avi Hayoun, Ben Eyal and Lior Zur-Lotan

November 3, 2016

Contents

1 Introduction ................................................................. 1
  1.1 Scheme implementation ........................................... 1
     1.1.1 Regarding Racket ........................................... 1

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

3 Parsing Combinators ................................................... 3
  3.1 A very brief refresher on parsers ................................. 3
  3.2 An incomplete overview of parser types .......................... 3
  3.3 Parsing Combinators ............................................... 4
  3.4 Example: Simple arithmetic expressions ......................... 4
     3.4.1 Plan ......................................................... 4
     3.4.2 The operator parser ........................................ 5
     3.4.3 The digit parsers ............................................ 5
     3.4.4 The natural numbers parser ................................. 6
     3.4.5 The arithmetic expression parser ............................ 6
     3.4.6 The arithmetic expression parser - fixed .................. 7
  3.5 *pack and *pack-with ............................................. 8

1 Introduction

• Course homepage: http://www.cs.bgu.ac.il/~comp171/
  – Useful Resources: http://www.cs.bgu.ac.il/~comp171/Useful_Resources

• Mayer’s course page: http://little-lisper.org/website/comp.html

1.1 Scheme implementation

The version of Scheme used in this course is Chez Scheme. This page contains
instructions for installing Chez Scheme on (nearly) any OS.
1.1.1 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 test your code on Chez Scheme using Linux. If your codes doesn’t work in Chez Scheme on Linux, 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 Linux before you submit anything.

2 Compiler structure

A compiler is a computer program that transforms source code written in a programming language into another computer language. Structure of a compiler:

```
  Scanner
     |
     v
  Parser
     |
     v
Semantic Analyzer
     |
     v
Code Generation
```
• **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 (parse tree) implicit in the input tokens.

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

• **Code Generator** - process of converting some intermediate representation of the source code 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 once it’s up, 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 parsers are implementations of 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).

### 3.2 An incomplete overview of parser types

- **Top-Down parsers** are parsers that try to build the AST from a single starting non-terminal and expand it. Then they expand the new non-terminals created from the derivation and so on.

- **LL parsers** are a variation of Top-Down parsers. They consume the input from left to right and they apply the derivation rules on the leftmost non-terminal in the current stage of the parse tree. The Parsing Combinators package creates LL parsers (actually LL Recursive Descent parsers).

- **Recursive Descent Parsers** are based on (perhaps mutually) recursive procedures where each procedure is usually responsible for a single derivation rule (sometimes it takes a few procedures to implement some rule and sometimes a few rules are implemented by one procedure). This creates a parser which is similar in structure to the CFG which defines the parser, and makes for very a easy-to-modify parser.
The parser you will write in this course (as part of the compiler) will be written using *parsing combinators*.

### 3.3 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 as 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 construct only very basic, simple, production rules explicitly, and define more complex production rules by *iteratively composing* simpler parsers. An example follows.***

### 3.4 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.***

#### 3.4.1 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). Do this using composition rules.
4. Define a parser for an expression.
5. Expand the expression parser into recursive parser for a chained expressions.
6. 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 \(1 + (2 - 3)\).*
3.4.2 The operator parser

\(<\text{op}> \rightarrow '+' | '-'\)

```scheme
(define <op>
  (new ;initialize a parser stack
     (*parser (char #\+)) ; create and push a parser for the '+' operator
     (*parser (char #\-)) ; create and push a parser for the '-' operator
     (*disj 2) ; pop top 2 parsers, and push their disjunction
     (*pack (lambda (op-char) ; transform the output
              (string->symbol (string op-char)))))
  )) ; finalize the stack and return the top-most parser
```

3.4.3 The digit parsers

\(<\text{digit-0-9}> \rightarrow '1' | \ldots | '9'\)

```scheme
(define <digit-0-9>
  (range #\0 #\9))
```

\(<\text{digit-1-9}> \rightarrow '1' | \ldots | '9'\)

```scheme
(define <digit-1-9>
  (range #\1 #\9))
```
3.4.4 The natural numbers parser

<nat> → <digit-1-9><digit-0-9>*

```
(define <nat>
  (new
    (*parser (char #\0))
    (*pack (lambda (_ 0))
      (*parser <digit-1-9>)
      (*parser <digit-0-9>) *star
      (*caten 2)
      (*pack-with
        (lambda (x xs)
          (string->number (list->string `'(,x ,@xs))))))
    (*disj 2)
    done))
```

3.4.5 The arithmetic expression parser

<expr> → <expr><op><nat> | <nat>

```
(define <exper>
  (new
    (*delayed (lambda () <exper>)) ; allow recursion
    (*parser <op>)
    (*parser <nat>)
    (*caten 3)
    (*pack-with (lambda (n op expr)
                  `'(,op ,n ,expr)))
    (*parser <nat>)
    (*disj 2)
    done))
```
**Left recursion:** Immediate left recursion occurs in rules of the form \( A \rightarrow Ao|\beta \) where \( \alpha \) and \( \beta \) are sequences of nonterminals and terminals, and \( \beta \) does not start with \( A \).

The production \(<expr> \rightarrow <expr><op><nat>\) is a left recursive production rule. In \( LL \) parsers (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: \(<expr> \rightarrow <nat><op><expr>\).

3.4.6 The arithmetic expression parser - fixed

\(<expr> \rightarrow <nat><op><expr> | <nat>\)

```plaintext
(define exper
  (new
    (*parser <nat>)
    (*parser <op>)
    (*delayed (lambda () <expr>)))

(*caten 3)

(*pack-with (lambda (n op expr) 
              `((,op ,n ,expr)))

(*parser <nat>)

(*disj 2)

done))
```

7
3.5 *pack and *pack-with

The *pack and *pack-with functions are also parsing combinators, meaning they take a parser as input and composes 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.

*pack-with works similarly, but for *caten. So (*pack-with (lambda (n op expr) '(',op ,n ,expr))) takes the 3 expressions returned from <nat>, <op>, and <expr>, and outputs a scheme expression which corresponds to the matched arithmetic expression: 1 + 2 − 3 → (+ 1 (- 2 3)).