# CS 131 Compilers: Discussion 3: Top-Down Parsers

**杨易为 季杨彪 尤存翰**
{yangyw,jiyb,youch}@shanghaitech.edu.cn

2021 年 3 月 1 日

## 1 Derivations in Top-down parsing

If a string belongs to a particular language, we can find aparse treefor it. A single nonterminal roots the parse tree (e.g ‘prog’ or ‘expr’ ). This top-level nonterminal branches off into constituent nonterminals, eventually breaking down into terminals and symbols at the leaves.Top-down parsingis a method of matching strings via an abstract preorder traversal of a parse tree. We’ ve already seen an example of this: recursive descent! As a recap from lecture:

1. **Leftmost derivation:** a sequence of rules following a preorder traversal of the parse tree.
2. **Rightmost derivation:** see previous definition, but with a right-to-left preorder traversal.

The reverse rightmost derivation is the rightmost derivation in reverse. The reverse rightmost derivation is an instance of bottom-up parsing—the string’ s lowest level de-tails are recognized first, then mid-level details, finally leading up to the start symbol.

## 2 Lambda Expression

### 2.1 Start with halting problems

You must have heard of ”object-oriented”: encapsulation, inheritance, polymorphism, all in all it seems to be very powerful. Functionality sounds like a real weakness.
Functional programming is a completely different programming paradigm, as opposed to ”imperative programming”. It is characterized by: invariants, inertia, high order functions, no side effects, everything is a function.

Have you ever thought of a tool that automatically checks for dead loops in your program? Whether you have ever thought about it or not, I have anyway, but unfortunately the answer is, no!

**Halting problem:** Given any program and its input, determine whether the program can end within a finite number of calculations. Assuming that this algorithm is actually made, you can just give it any function and the input to this function, and it will tell you whether the function will run to the end or not. We describe this in the following pseudo-code:

```
function halting(func, input) {
    return if_func_will_halt_on_input;
}
```

and another pseudo-code:

```
function ni_ma(func) {
    if (halting(func, func)) {
        for(;;) //dead loop }
}
```

When we call **ni_ma(ni_ma)** . We can't deduce whether it is halting or not halting. Thus halting problem is not deciable.

Here comes the **lambda calculus**, Consider the following **lambda calculus** grammar:

## 2.2 What language is accepted by the following DFA?

```
var  : ID ;
expr : var
     .  '('   'l'  var  '.'  expr  ')'
     |  '('  expr expr  ')'  ;
```

Use this grammar to construct leftmost and reverse rightmost derivations of the following strings.

$$(\lambda f.(\lambda x.(f(fx))))$$

Of the three syntaxes defined earlier, the first two are used to generate "functions" and the third is used for function "calls", e.g.: $((\lambda xy.x+y)23)$. For simplicity: let add= $\lambda xy.x+y$ then (add 2 3)

Here's the two axiom of **lambda calculus**:

1. **Permutation axiom:** $\lambda xy.x+y => \lambda ab.a+b$
2. **Substitute axiom:** $(\lambda xy.x+y)ab => a+b$

## 2.3 Function Generator

The $\lambda$ algorithm is equivalent to a function generator

1. let mul= $\lambda$ xy.x*y, we have: mul 3 5 -> 3 * 5
2. let con= $\lambda$ xy.xy, we have: con 'yangyw' 'ISvegetable' -> 'yangywISvegetable'
3. let not = false -> true . true -> false
4. let ext-and = true value -> value . false value -> false . value true -> value . value false -> false
5. let if = $\lambda$ cond tvalue fvalue . (cond and tvalue) or (not cond and fvalue)
6. let fact =$\lambda$ n . if (n==0) 1 (n * fact n-1)

We noticed that the sixth take the function itself as the input for the function. It's not in accordance to the math axiom loop definition and can be parsed by compilers. It will parameterize itself as $self$:

let P = $\lambda$ self n . if (n==0) 1 (n * self(self n-1)) let fact n = P (P n)

Unfortunately this isn't really recursive, it's just an extra argument passed in each time and called repeatedly. So what is our purpose? I want a truly recursive function that is

**Answer:**

## 2.4 Fixed Point $P(fact) = fact$

What is a fixed point? It is a point (in the generalized sense) that is mapped by a function, and the result obtained is still the point. Imagine a map of China on the wall falls on the floor, there must be and only one point on the map with its actual position.

## 2.5 Magical Y

So let's go ahead and assume that there is a magic function Y that finds the immovable point of this pseudo-recursive function, namely:

$$Y(F) = f = F(Y(F)), \text{ where } F(f) = f \text{ and } Y(P) = fact$$

Let's construct the Y Combinator (a starter helper in Silicon Valley.)

1. let Y = $\lambda$ F . G (G), where G = $\lambda$ self. F(self(self))

$$\begin{aligned} \text{Y(P)} &= \text{G(G) where } G = \lambda \text{ self. P(self(self))} \\ &= \text{P(G(G))} \\ &= \lambda n. \text{ if (n==0) 1 (n * G(G) n-1)} \end{aligned} \tag{1}$$

Suppose Y(P) = fact, then Y(P) = fact = $\lambda$ n. if (n==0) 1 (n * fact n-1)

Only if we have Y, we can transform the pseudo-recursive function into the true recursive function we want. Now when we want to define a recursive function, we just add a self parameter, define it in a pseudo-recursive way, and then use the Y-combination subset to make it the true recursion we want.

## 2.6 Turing Equivalence

We have successfully derived the Y-combinator, which is equivalent to deriving a theorem in the **lambda calculus** axiomatic system: It is possible to refer to itself in the process of defining a function This theorem is an important step in proving the Turing equivalence of **lambda calculus**s. What does it mean that the **lambda calculus** is Turing-equivalent?

It means that its computational power is identical to that of our computers. It means that any program can be described by the **lambda calculus**, and that the functions described by the **lambda calculus** must be computable by a computer.

Recall that we just talked about the stopping problem, i.e., the undecidability of whether a Turing machine can stop when given an arbitrary input. The equivalent of this proposition in the **lambda calculus** is: There does not exist an algorithm that can determine whether any two $\lambda$-functions are equivalent, i.e., have f(n) = g(n) for all n.

## 2.7 Functional Programming

Haskell is a purely functional programming language, named in honor of Haskell Curry. Everything in Haskell is a function, not even the concept of variables in imperative programming; all its variables are allowed to be assigned only once and then are immutable, just like the assignment of variables in mathematical derivation.

Haskell also does not have a control flow structure in the usual sense, such as for loops, but instead has recursion. two other important features of Haskell are side-effect free and inertial evaluation. No side effects means that any function given the same input will have the same result every time it is called, and inertial evaluation means that the function will not compute immediately unless needed. Here's Some examples in Haskell:

Before running the Haskell program, first you need to install a GHC compiler, and then run ghci to enter interactive mode. Here are some example

1. let max a b = if a>b then a else b
   (a) max 3 4 = 4
   (b) max 1.0001 1 = 1.0001
   (c) max "BYVoid" "CmYkRgB123" = "CmYkRgB123"
2. list X ::= [] | elem: (list X)
   1 = 1:[]
   1, 2, 3 = 1:2:3:[]
   1,3.. = [1,3,5,...]
3. **Pattern Recognition:** let first (elem:rest) = elem
   (a) first [1,3] = 1, pairing (1,3:[])
4. **List Sum:** accumulate (elem:rest) = elem + accumulate rest
5. **Palindrome:** palindrome [] = True

palindrome [_] = True
palindrome (elem:rest) = (elem == last rest) && (palindrome(init rest))

6. **Lazy Evaluation:** [1,3..] !! 42 = 85
7. **Linear Fibbonacci:** fib = 1:1:zipWith (+) fib (tail fib)

## 3 Recursive Descent Parsers

It is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal. A Predictive Parser is a special case of Recursive Descent Parser, where no Back Tracking is required. By carefully writing a grammar means eliminating left recursion and left factoring from it, the resulting grammar will be a grammar that can be parsed by a recursive descent parser.

Try writing a recursive descent parser for **lambda calculus**. Assume the existence of $LITERAL(C)$, $yynext()$, and $yyerror()$ in flex.

**Answer:**

### 3.1 Ambiguous Grammars

A grammar is ambiguousif it permits multiple distinct parse trees for some string.For example, without the order of operations, 12—8/4 could parse as 1 or as 10. Makethe following grammar unambiguous, and also give precedence to ‘/’ over ‘-’ .

```
e : INT
  | e ‘-’ e
  | e ‘/’ e ;
```

As another example is called dangling else, consider the following grammar:

```
e : 0 | 1 ;

stmt : e ’ ;’
     | if e then stmt
     | if e then stmt else stmt ;
```

Give an example of a string in the language that can produce multiple parse trees.

**Answer:**

How to resolve the problem?

**Answer:**

## 3.2 Syntax-directed Translation

Parser-generators usually support syntax-directed translation, which is a convenientway to execute an action every time a grammar rule is matched. While definingactions, the variable $$ refers to a location into which the semantic value of thecurrent symbol can be stored. The variables $1, ...,$n refer to the semantic values ofthe symbols used to match the current rule. Here’ s an example:

```
p : e ’ ;’         { printf("Result: %d\n", $1); }

e : INT          { $$ = $1; }
  | e  ‘-’  e      { $$ = $1 - $3; }
  | e  ‘/’  e      { $$ = $1 / $3; } ;
```

Write a syntax-directed translator for the first grammar you wrote for this **Simple Caculator**.
**Answer:**

Write for **lambda calculus**.
**Answer:**