CS 131 Programming Languages

October 1, 2009

OCaml Syntax

This grammar:

- `type gnt = | Grunt | Conversation | Sentence | Snore | Shout | Quiet ;;`
- `let giant grammar = (Conversation, function | Conversation -> [ [N __; T __]; […]; ] | Grunt -> [ [N __; T __]; […]; ] );;`

requires this type declaration:

- `type ‘a, ‘b symbol = | N of ‘a | T of ‘b;;`
- `type ‘a option = None | Some of ‘a;;`

Check whether a variable is terminal or nonterminal:

- `let is_terminal s = match s with | N _ -> false | T _ -> true ;;`
- `let is_terminal = fun s -> match s with | N _ -> false | T _ -> true;;`
- `let is_terminal = function | N _ -> false | T _ -> true;;`

The underscore (_) repreents a throw-away variable that will never be used. However, this function is a very bad example because one should never need it – use *match* instead.

The intent is for some other function to call *is_terminaal*:

- `if is_terminal s`
    - `then match_terminal (arg s)`
- `else match_nonterminal (arg s);;`

But a proper way is to use *match* (pattern matching) to find the type of the variable:

- `match s with | N a -> match_terminal a | T v -> match_nonterminal v;;`

Ways to define a function:

- `let idf x = x`
- `let idf = fun x -> x`

Examples

A function to reverse the list:

- `let rec rev = function | [] -> [] | a :: d -> (rev d) @ [a];;`
    - `rev: ‘a list -> ‘a list = <fun>`

However, the above takes O(n^2) to run – it iterates over successive smaller portions of the array. Let’s generalize the function to solve the problem more efficiently:

- `let rec reva a = function | [] -> a | hd::tl -> reva (hd::a) tl;;`
- `let rev x = reva [] x;;`
- Another notation: `let rec reva = fun a -> fun l -> match l with | [] -> a | hd:tl -> reva (hd::a) l;;`

The above takes O(n) to run and is tail recursive. For lists, the “::” operator takes O(1) to run while “@” operator takes O(n) because it makes a copy of an entire first list and then appends it to the second list.

The keyword *function* is for pattern matching while the keyword *fun* is for currying.

Let’s try to write a bad function against Eggert’s advice. The function will extract value from a terminal symbol.

- `let extract_value = function | T x -> x;;`
    - `warning: patterns do not exhaust alternatives. here’s an example of an alternative that does not match: N y`
    - `extract_value: (‘a, ‘b) symbol -> ‘b = <fun>`

Let’s try to pass a non-terminal symbol to the function:

- `extract_value (N Grunt)`
    - `runtime error: pattern match failure`

Let’s try to make the compiler happy:

- `let extract_value = function | T x -> x | N y -> y;;`
    - `extract_value: (‘a, ‘a) symbol -> ‘a = <fun>`
- `let extract_value = function | T x -> x | _ -> 0;;`
    - `extract_value: (‘a, int) symbol -> int;;`

Let’s write a function that takes a tuple and returns its first member.

- `let fst (a, _) = a`
- Alternative notation: `let fst = fun x -> match x with (a, _) -> a;;`
    - `fst: ‘a * ‘b -> ‘a`

Let’s try to write the same function but for a list:

- `let fstl a::_ = a;`
    - this will generate a warning because passing an empty list will generate a runtime error.

A function that returns a minimum value. General rule: `minval a@b = min(minval a, minval b).`

- `let minval lt inf = function`
    - `| [] -> inf`
    - `| a::d -> let mt = minval d inf`
        - `in if lt a mt`
        - `then a else mt;;`
- `lt = the function to compare values (less than)`
- `inf = predefined value that signifies infinity (largest)`

General Syntax for Grammars

There is an ISO standard for EBNF. They wanted to be as formal as possible so they had to come up with formal syntax for how to write down grammars. They used their own formal notation to define the syntax of their formal notation.

1. `syntax = syntax rule, {syntax rule};`

Comma (,) is concatenation. Inside curly braces ({ and }) means zero or more objects, concatenated. Identifiers can have spaces. Nonterminal symbols are identifiers (syntax rule) and terminal symbols are inside single quote marks (‘=’, ‘;’).

2. `syntax rule = meta id, ‘=’, defns list, ‘;’;`
    a. This rule is just simply awesome because it describes itself.
3. `defns list = defn, {‘|’, defn};`
4. `defn = term, {‘,’, term};`

The standard can be found at http://www.cl.cam.ac.uk/~mgk25/iso_ebnf.html

Problems with Grammars

1. *Ambiguity*.
    a. Example of an ambiguous grammar: `E -> E + E | E -> E * E | E -> ID | E -> NUM | E -> (E)`
    b. When one comes across an ambiguous grammar, first reaction should be to fix it so that it is not ambiguous anymore but it still describes the language that we want.
    c. Expression 1+2*3 can be parsed in two ways given the above grammar.

d. We look at the example of the ambiguity and look at the wrong (buggy) parse tree and modify tree so that the wrong parse tree is impossible. In the case above, the rule `E -> E * E` allows parses that we don't want to allow. Modified expression: `E -> E * E; E′ = E′ * E | ID | NUM | (E); and E -> E′ * E`
e. Expression 1*2*3 can be parsed in two ways given the above grammar.
f. We can fix that by modifying expression `E -> E′ + E` to `E -> E′ + E′′` and defining `E′′ -> ID | NUM | E`.
g. There errors propagate so we create a new symbol to avoid repeating ourselves. `F -> ID | NUM | (E)`. Now, `E′′′ -> E′ * F′ | F`.
h. Final grammar: `F -> ID | NUM | (E)` and `E -> E + T | T` and `T -> T * F | F`.

New grammar for C

Statement *stmt* is defined as:

```
while (expr) stmt
do stmt while (expr);
if (expr) stmt;
if (expr) stmt else stmt;
expr;
;
return expr;
{ stmt_list }
```

and so on…

The grammar is ambiguous – there is the dangling else problem:

```
if (a) if (b) c; else d; // the else part could belong to either if statement, both parses are valid
```

Fixing the grammar:

stmt:

```
if (expr) st else stmt
st
```

2. *Rules*.
    a. Suppose you define a non-terminal symbol but no rules use it. The time spent writing that non-terminal is wasted.
    b. Or, suppose you have rules using the non-terminal but it's never defined. Therefore you can never construct a parse three that uses that non-terminal and the rule that uses that non-terminal is useless.
    c. Suppose you have a grammar containing `E -> E`. The parse tree will not be useful and the grammar is therefore useless.