CS131: Programming Languages

Fall 2015
Week #2

Today

- OCaml practice
- HW2

OCaml Practice

head_eq

input: element, list

output: If head of the list is equal to the input element: Some(h)
else None

```
let head_eq h list =
  match list with
  | [] -> None
  | h::_ -> Some h
  | _ -> None
```

Do we need all the match cases?

OCaml Practice

head_eq

input: element, list

output: If head of the list is equal to the input element: Some(h)

else None

```
let head_eq h list =
  match list with
  | [] -> None
  | h::_ -> Some h
  | _ -> None
```

Do we need all the match cases?

No. We can remove either the 1st or 3rd match cases. Why?

OCaml Practice

head_eq

input: element, list

output: If head of the list is equal to the input element: Some(h)
else None

```
let head_eq h list =
  match list with
  | h::_ -> Some h
  | _ -> None
```

```
let head_eq h =
  function
  | h::_ -> Some h
  | _ -> None
```

Are these the same? A: Yes

OCaml Practice

head_eq

input: element, list

output: If head of the list is equal to the input element: Some(h)
else None

```
let head_eq h list =
  match list with
  | h::_ -> Some h
  | _ -> None
```

What does

`head_eq 3 [1;2;3]` output?

```
Some 1
```

How do we fix it?

OCaml Practice

head_eq

input: element, list

output: If head of the list is equal to the input element: Some(h)
else None

```
let head_eq h list =
  match list with
  | h::_ -> Some h
  | _ -> None
```

```
let head_eq h list =
  match list with
  | x::_ when x=h -> Some h
  | _ -> None
```

OCaml Practice

drop

input: list, n

output: the same list but with every nth element removed

```
let drop list n =
```

Sample output:

```
drop [] 1 = []
drop [1;2;3] 1 = []
drop [1;2;2;3] 2 = [1;2]
```

OCaml Practice

drop

input: list, n

output: the same list but with every nth element removed

```
let drop list n =
  let rec help i = function
    | [] -> []
    | h :: t -> if i = n
                  then help 1 t
                  else h :: help (i+1) t
  in help 1 list;;
```

HW2: Naive Parsing of CFGs

- A parser generator
- Submission due: Oct 16, 11:55 pm

Converting Grammars

```
let old_grammar =
  Conversation,
  [...
   Sentence, [N Quiet];
   Sentence, [N Grunt];
   Sentence, [N Shout];
   ...]
```

Pair of a nonterminal starting symbol and a list of rules.

```
let new_grammar =
  Conversation,
  function
   | ...
   | Sentence -> [[N Quiet];
                  [N Grunt];
                  [N Shout]]
   | ...
```

Pair of a nonterminal starting symbol and a production function.

The production function is one large pattern match on the nonterminal.

Converting Grammars

Careful: RHS of the new grammar is a **list of lists**

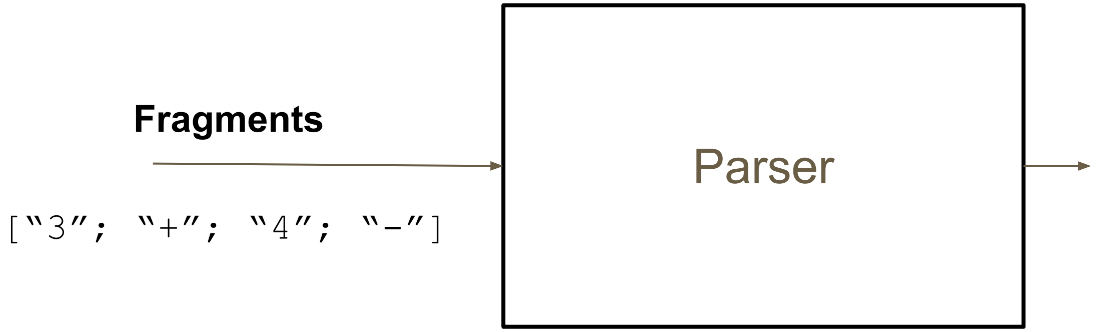
```
let correct_gram =
  Conversation,
  function
   | ...
   | Sentence -> [[N Quiet];
                  [N Grunt];
                  [N Shout]]
   | ...
```

Correct.

```
let wrong_gram =
  Conversation,
  function
   | ...
   | Sentence -> [N Quiet;
                  N Grunt;
                  N Shout]
   | ...
```

Incorrect!

Parser



Fragments

```
[“3”; “+”; “4”; “-”]
```

Parser

Derivation

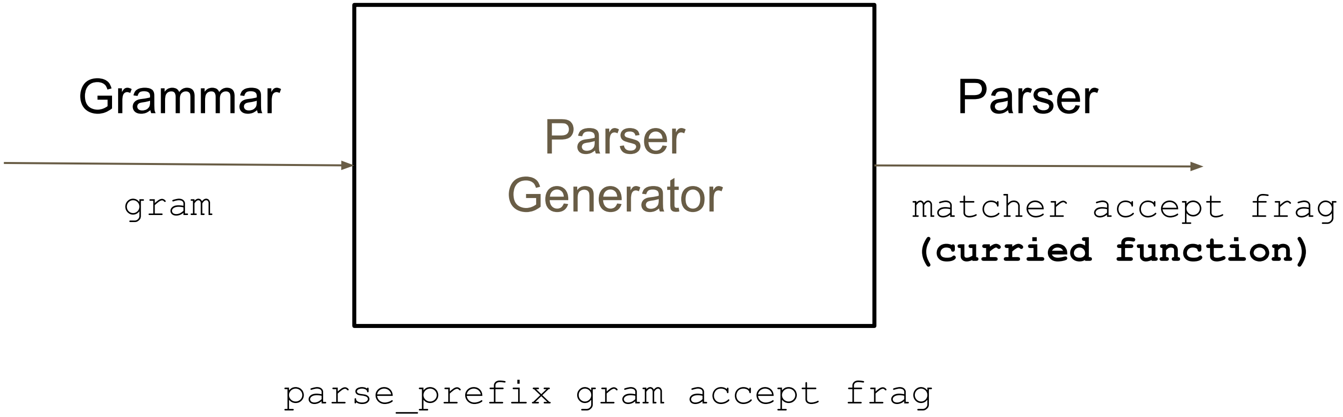
```
Expr   -> Term Binop Expr
Term   -> Num
Num    -> “3”
Binop  -> “+”
Expr   -> Term
Term   -> Num
Num    -> 4
```

Suffixes

```
[“-”]
```

- Input: fragment (i.e. list of tokens)
- Output:
  - the derivation for the given tokens
  - remaining tokens (suffixes)

The Goal: Parser Generator



- Input: Grammar `(starting symbol, production function)`
- Output: A parser which has internalized that grammar

Acceptor

- A function that determines whether the given input is "**acceptable**"

- Input:
  - `rules` (a derivation)
  - `frag` (a list of tokens; suffixes)
- Output:
  - `Some (rules,frag)` if we like the input
  - `None` if we don't like it

Acceptor: examples

-
  ```
  let accept_all rules frag = Some (rules, frag)
  ```

-
  ```
  let accept_empty_suffix rules = function
  | [] -> Some( rules, [] )
  | _ -> None
  ```

-
  ```
  let accept_only_non_lvalues = …
  ```
  - if a derivation’s **rules** contain ‘Lvalue’, it returns **None**

Matcher

- A function that matches **a prefix of a fragment** and checks whether the acceptor passes or not.

- Input: an **acceptor** and a **fragment**
- Output: whatever the acceptor returns
  - `Some (rules, frag) | None`

Basic Matcher

```
let match_num num frag accept =
```

Basic Matcher

```
let match_num num frag accept =
  match frag with
    | [] -> None
    | n::tail -> if n = num
                   then accept tail
                   else None;;
```

Hint code

```
let append_matchers matcher1 matcher2 frag accept =
  matcher1 frag (fun frag1 -> matcher2 frag1 accept)

let match_empty frag accept = accept frag

let make_appended_matchers make_a_matcher ls =
  let rec mams = function
    | [] -> match_empty
    | head::tail -> append_matchers (make_a_matcher
head) (mams tail)
  in mams ls
```

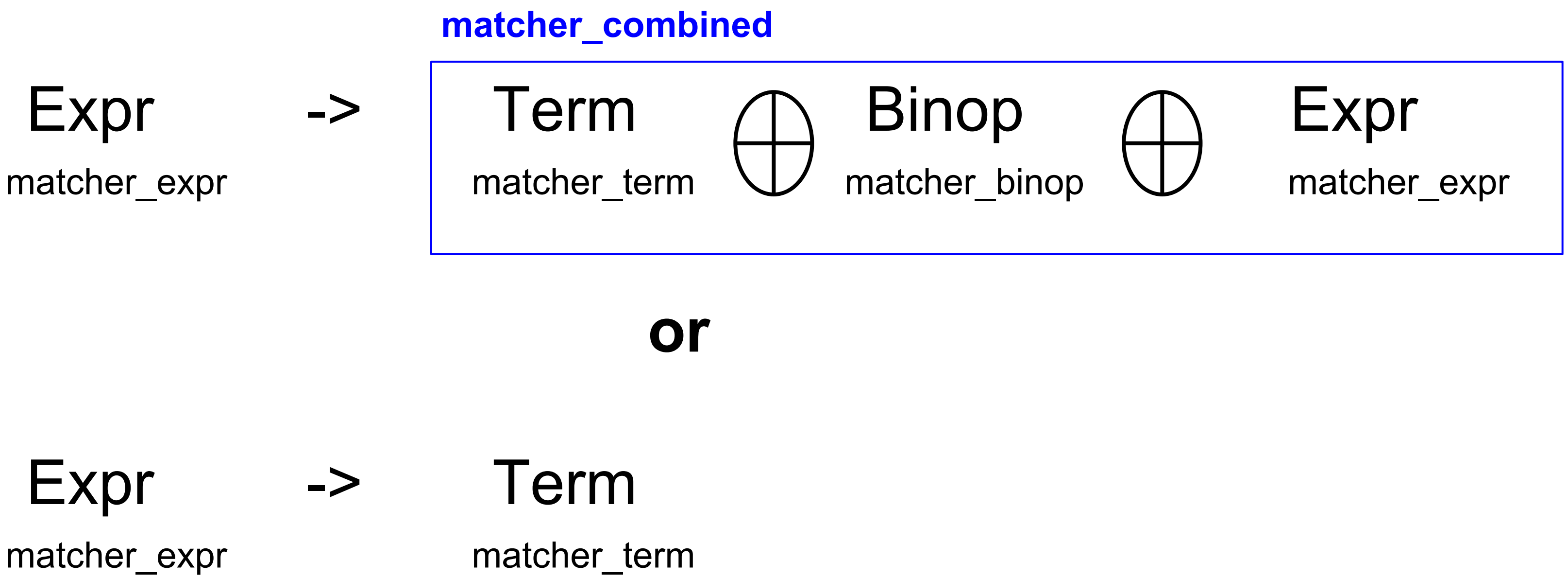
Hint code: make_appended_matchers

Expr -> Term ⊕ Binop ⊕ Expr

matcher_expr matcher_term matcher_binop matcher_expr

- When we define a matcher for the nonterminal symbol **expr**, it can be represented by a combination of three **concatenated** matchers.
-
  ```
  matcher_expr acceptor [“3”; “+”; “4”]
  = matcher_term acceptor [“3”] and
    matcher_binop acceptor [“+”] and
    matcher_expr acceptor [“4”]
  ```

Hint code: make_or_matcher



matcher_combined

Expr -> Term ⊕ Binop ⊕ Expr

matcher_expr matcher_term matcher_binop matcher_expr

or

Expr -> Term

matcher_expr matcher_term

- matcher_expr = **or_matchers**
  [matcher_combined; matcher_term]

Acceptor: Example Purpose

```
Expr -> Term Binop Expr
        | Term
Term -> Num
Binop -> “+”
Num  -> 1
```

- Force a partial derivation:

```
let accept_only_non_binop rules frag =
  if contains_binop rules
  then None
  else Some (rules, frag)
```

- Given this acceptor, only accepted derivation of [“1”; “+”; “1”] would be:

```
[Expr, [Term]; Term, [Num]; Num, [1]]
```

and the remaining fragment would be [“+”; “1”]

Acceptor: Example Purpose

```
Expr -> Term | Num
Term -> Num
Num  -> 1
```

- Ambiguous grammar: two ways to derive “1”
  - What are the ways?
- An acceptor can force one of the derivations to be chosen

```
let rec contains_term = function
  | [] -> false
  | (Term,_)::_ -> true
  | _::rules -> contains_term rules

let accept_only_non_term rules frag =
  if contains_term rules
  then None
  else Some (rules, frag)
```