CS131: Programming Languages

Fall 2015
Week #2

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.

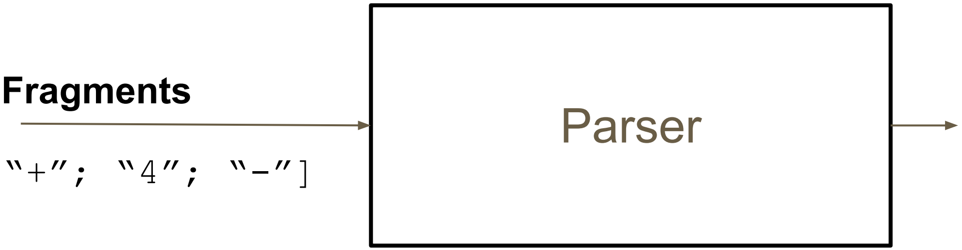
```
let wrong_gram =
  Conversation,
  function
   | ...
   | Sentence -> [N Quiet;
                  N Grunt;
                  N Shout]
   | ...
```

Incorrect!

Parser

Fragments

`["3"; "+"; "4"; "-"]`



Derivation

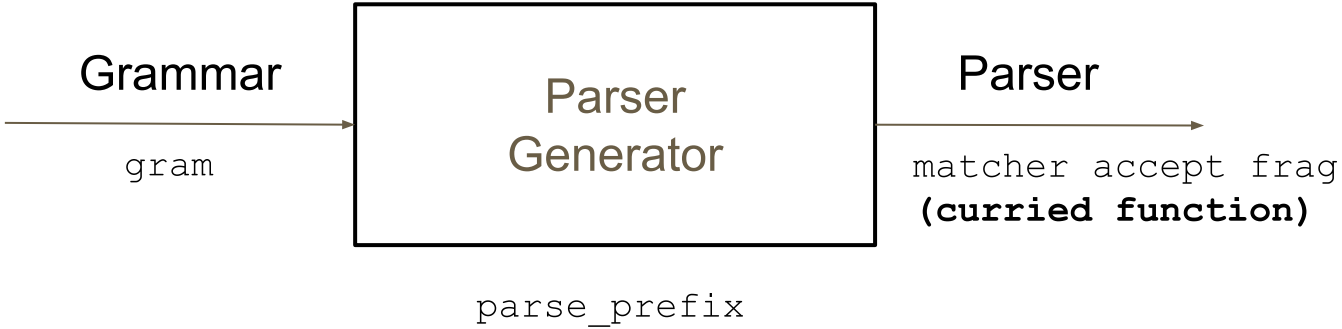
```
Expr   -> Term Binop Expr
Term   -> Num
Num    -> "3"
Binop  -> "+"
Expr   -> Term
Term   -> Num
Num    -> 4
```

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(**rules**) contains '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`

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"]`
  `matcher_binop` `acceptor ["+"]`
  `matcher_expr` `acceptor ["4"]`

Hint code: make_or_matcher

		matcher_combined				
Expr matcher_expr	->	Term matcher_term	⊕	Binop matcher_binop	⊕	Expr matcher_expr

or

Expr matcher_expr	->	Term 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)
```

Grammar rules and Function call

- Grammar rules
  - S → A | B   `(* S: starting symbol *)`
  - A → "a" S | "a"   `(* A, B: nonterminal symbol *)`
  - B → "b" S | "b"   `(* "a", "b": terminal symbol *)`

- Possible sentences from this grammar:
  - "a", "b", "aa", "ab", "ba", "bb", "aaa", "aab", "aba", "abb", … , "bbb" , and so on

Sentence Generator

```
let rec s buf = if (String.length buf) <= 3 then
    begin
        let a buf = begin
            s (buf ^ "a"); (* A -> "a" S *)
        end in
        let b buf = begin
            s (buf ^ "b"); (* B -> "b" S *)
        end in begin
            print_string buf;
            print_string "\n";
            a buf; (* S -> A *)
            b buf; (* S -> B *)
        end
    end;;

s "";;
```

What will be the output?

How to get a derivation (cont'd)

(example code)

```
let produce nt = ( snd grammar ) nt;;
let rec traverse start_symbol depth =
     List.iter (fun lst->begin
          if (* stopping condition *) then
               begin
                    (* print statements *)
                    List.iter ( fun x ->
                    traverse x (* number of terminals *)) lst
               end
          else ();
     end
     ) (produce start_symbol);;

# traverse Expr 1;;
```

(output)

```
expr -> term binop expr
term -> num
num -> 0
num -> 1
...
binop -> +
binop -> -
expr -> term
term -> num
num -> 0
num -> 1
...
expr -> term
term -> num
num -> 0
num -> 1
...
```

This approach will work, but what about the **complexity** of the function traverse??