CS131: Programming Languages

Fall 2015
Week #1

Today

- Supplements for HW #1
- Basic things for upcoming HW #2 (has not been posted yet)

HW #1: Submission

Due: Today (Oct 2) 11:55 pm

- CCLE
- email to TAs if you are not enrolled

HW #1: Before submission

- You have to define

```
type ('a, 'b) symbol = N of 'a | T of 'b
```

in the beginning of the source code file.

- Please do not implement the functions in the test file hw1test.ml

Review: Determining blind alleys

- A rule is blind-alley if it is **NOT productive**
  - If it can be reduced to a terminal string (all terminal symbols)

- Aspects that don't matter:
  - **Reachability**: if a rule will ever be applied in a derivation
  - **Start symbol**: relates to which terminal strings can be produced
    - Is independent from individual rules

Review: Function currying

- Currying -- to translate the evaluation of a function that takes multiple arguments into evaluating a sequence of functions (with a single argument)
- `f: (X * Y) -> Z  /  curry f: X -> (Y -> Z)`

- Why is it useful?
  - `my_equal x y` where the type is `'a -> 'a -> bool = <fun>`
  - `List.map (my_equal 2) [1;2;3]` the type of `(my_equal 2)` is `int -> bool = <fun>`

    `- : bool list = [false; true; false]`
  - Otherwise, we have to define a helper function:
    - `fun x -> my_equal_non_currying (x,2)` `'a * 'a -> bool = <fun>`
  - It will be heavily used in the homework #2.

Definitions

- Substitution rules (“alternative list” in spec)
  - A list of right hand sides for a given nonterminal symbol

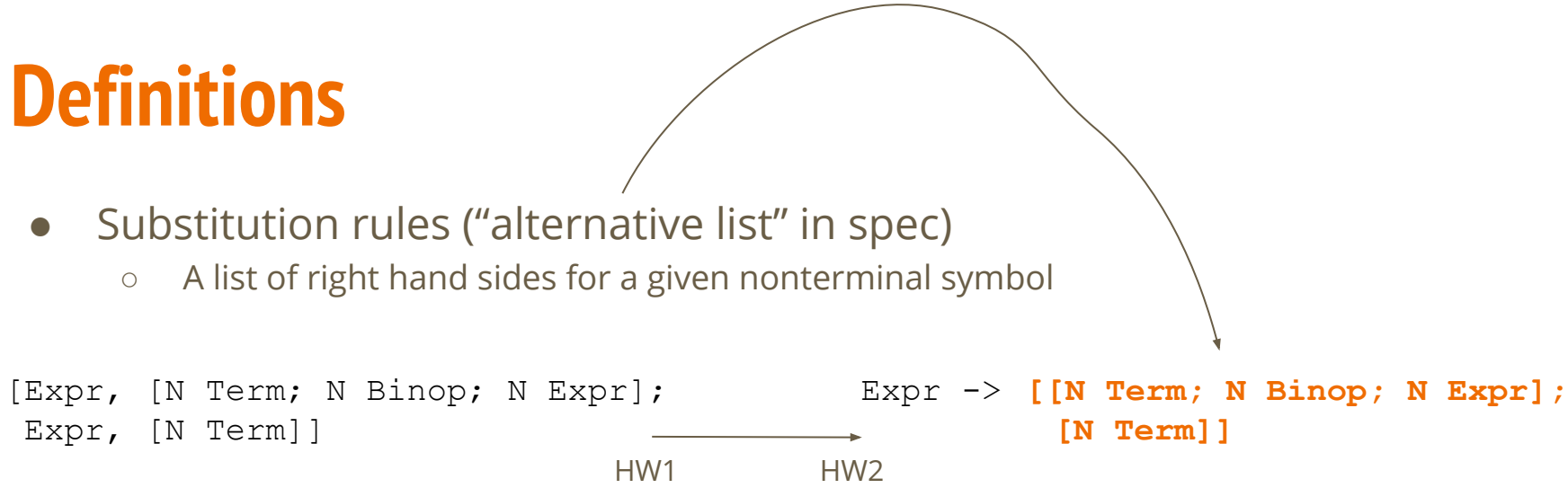
```
[Expr, [N Term; N Binop; N Expr];          Expr -> [[N Term; N Binop; N Expr];
 Expr, [N Term]]                                    [N Term]]
```

HW1 → HW2

- Production function
  - It takes a nonterminal value as an argument and returns an alternative list
  - `f ( Expr ) = [ [ N Term; N Binop; N Expr ]; [N Term] ]`

Definitions

- Substitution rules (“alternative list” in spec)
  - A list of right hand sides for a given nonterminal symbol



- Production function
  - It takes a nonterminal value as an argument and returns an alternative list
  - `f ( Expr ) = [ [ N Term; N Binop; N Expr ]; [N Term] ]`

Grammar Representation...

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

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


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

   | ...
```

a pair of a nonterminal starting symbol and a production function

The grammar in the example

```
Expr → Term Binop Expr  |  Term
Term → Num  |  Lvalue  |  Incrop Lvalue  |  Lvalue Incrop  |  "(" Expr ")"
Lvalue → "$" Expr
Incrop → "++" | "--"
Binop → "+" | "-"
Num → "0" | "1" | "2" | … | "9"
```

A derivation for the phrase "3" "+" "4" is:

Expr -> Term Binop Expr	Term Binop Expr → leftmost derivation
Term -> Num	Num Binop Expr
Num -> "3"	"3" Binop Expr
Binop -> "+"	"3" "+" Expr
Expr -> Term	"3" "+" Term
Term -> Num	"3" "+" Num
Num -> "4"	"3" "+" "4"

Leftmost Derivation

Grammar (CFG)

* Nonterminal symbols: { S, A, B, C },
* Terminal symbols: { a, b, c },
* Starting symbol: S
* Rules: { S -> ABC, A -> a, B -> b, C -> c }

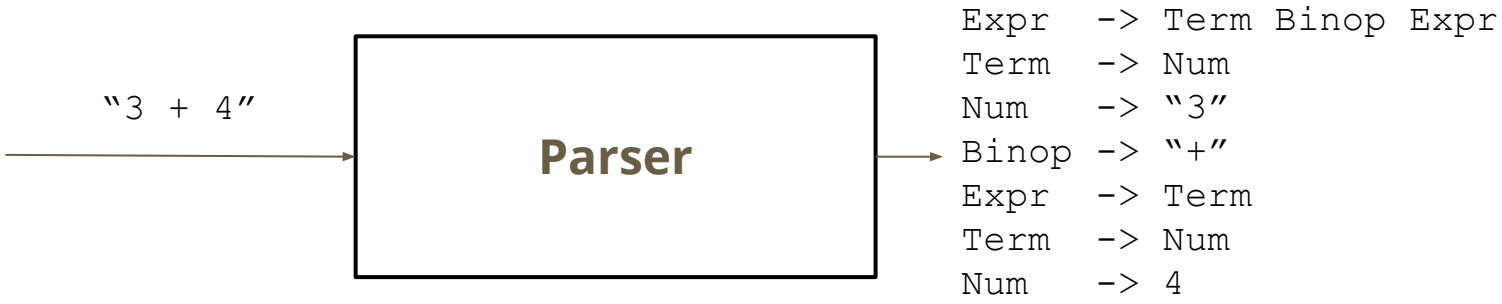
Leftmost derivation

S -> ABC -> aBC -> abC -> abc

Rightmost derivation

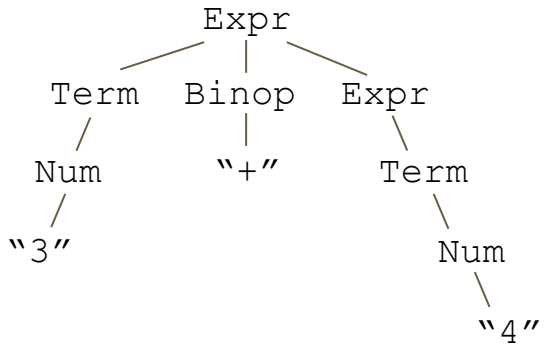
S -> ABC -> ABc -> Abc -> abc

Parser



- Input: terminal string/phrase/code
- Output: A derivation from the start symbol to the phrase
  - Information about the structure of the program
- Internally knows the grammar it's using to parse

Parser



HW2 →

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

- Output of a parser
  - Many times structure of program is represented as a tree
  - For HW2, output will be a list of rules

Parser generator

- Input: Grammar
- Output: a parser which has internalized that grammar

How to get a derivation

(example)

```
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;;
```

Option

- A standard type that can be either **None** (undefined) or **Some x** (where x can be any value)

```
type 'a Option = None | Some of 'a
```

- Useful replacement for NULL pointer values in C++
  - Fits into the OCaml type system

Hint code (in 2006)

- A pattern matcher generator that works only on DNA patterns
  http://web.cs.ucla.edu/classes/fall06/cs131/hw/hw2.html
- A DNA pattern is defined as…

```
type nucleotide = A | C | G | T
type fragment = nucleotide list          (ex) [ A; G; T ]
type acceptor = fragment -> fragment option
type matcher = fragment -> acceptor -> fragment option

type pattern =
  | Frag of fragment
  | List of pattern list
  | Or of pattern list
  | Junk of int
  | Closure of pattern
```

We want to detect the following patterns:

```
A  G  T
A  ?  G  T
```

```
# let p = List [ Frag [A]; Junk 1; Frag [G; T] ];;
val p : pattern = List [Frag [A]; Junk 1; Frag [G; T]]
```

Acceptor

A function returns the ‘wrapped’ fragment , if accepted
‘undefined’ , if rejected

- `type acceptor = fragment -> fragment option`

For example,

- (fun x -> Some x) → it always succeeds the fragment x.
- (fun x -> None) → it always rejects the fragment x.
- (function | (A::tail) -> Some tail | _ -> None) →it accepts if the first nucleotide is A…

Matcher

- A function that inspects a given fragment to find a match for a prefix that corresponds to a pattern.

- val p : pattern = List [Frag [A]; Junk 1; Frag [G; T]]
    - How can we create a matcher for this pattern?

A ‘matcher’ generator

- Similar to the ‘parser generator’ in the previous slide.

```
let rec make_matcher = function
  | Frag frag -> make_appended_matchers match_nucleotide frag
  | List pats -> make_appended_matchers make_matcher pats
  | ...
  | Junk k -> match_junk k (* function currying *)
  | ...

let rec match_junk k frag accept =
  match accept frag with
    | None ->
        (if k = 0 then None
         else match frag with
          | [] -> None
          | _::tail -> match_junk (k - 1) tail accept)
    | ok -> ok
```

Acceptor

```
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)
```