Design Principles of Programming Languages

Practice

arith, fullsimple, fullref

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong
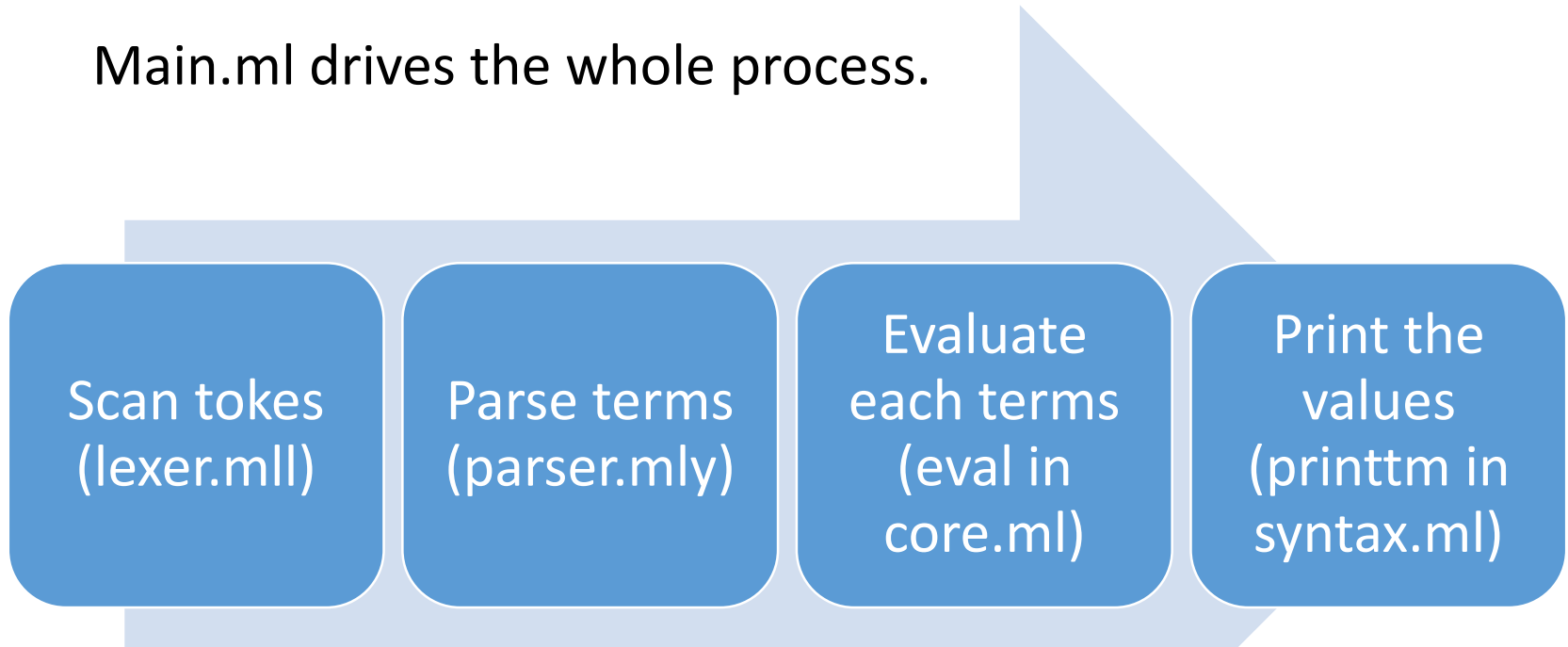
Peking University, Spring Term

Homework

- Please use the associated code to finish the exercises
- If an exercise asks for a program in the defined language, submit the program.
- If an exercise asks for modifying the interpreter
  - Submit all code
  - Your submission should contain file test.f which contains the expressions required by the exercise
  - TA will perform the following two commands to verify your submission:
    - make
    - ./f test.f
- Please submit a compressed file where each problem in a separate folder

Structure of arith

Main.ml drives the whole process.

| Scan tokes (lexer.mll) | Parse terms (parser.mly) | Evaluate each terms (eval in core.ml) | Print the values (printtm in syntax.ml) |

Syntax.ml defines the terms.

Syntax.ml

```
type term =
    TmTrue of info
  | TmFalse of info
  | TmIf of info * term * term * term
  | TmZero of info
  | TmSucc of info * term
  | TmPred of info * term
  | TmIsZero of info * term
```

Info: a data type recording the position of the term in the source file

eval in core.ml

```
let rec eval t =
    try let t' = eval1 t
            in eval t'
    with NoRuleApplies → t
```

eval1: perform a single step reduction

Commands

- Each line of the source file is parsed as a command
  - type command = | Eval of info * term
  - New commands will be added later

- Main routine for each file
```
let process_file f  =
  alreadyImported := f :: !alreadyImported;
  let cmds = parseFile f in
  let g  c =
    open_hvbox 0;
    let results = process_command  c in
    print_flush();
    results
  in
    List.iter g  cmds
```

Exercise arith.simple_use

- Using arith to write the following equation
    - Return five if two is zero, otherwise return nine

    - Hint: read the code in parser.mly

Exercise arith.size

- Make the evaluation computes the size of a term (3.3.2) instead of reducing the term, and test it on the original test.f

- Hint:
    - pr: string->unit prints a string to the screen
    - string_of_int : int->string converts an integer into a string
    - Remember to change both .ml and .mli files

- Some abbreviations
    - UCID = upper case identifier
    - LCID = lower case identifier
    - ty = type
    - tm = term
    - LCURLY = "{"
    - RCURLY = "}"
    - USCORE = "_"

Exercise arith.big-step

- Change the evaluation to use big-step semantics, and compute the following expressions
  - true;
  - if false then true else false;
  - if 0 then 1 else 2;
  - if true then (succ false) else 2;
  - 0;
  - succ (pred 0);
  - iszero (pred (succ (succ 0)));

Big-step vs small-step

- Big-step is usually easier to understand
  - called "natural semantics" in some articles
- Big-step often leads to simpler proof
- Big-step cannot describe computations that do not produce a value
  - Non-terminating computation
  - "Stuck" computation

fullsimple

- Implementing all extensions in Chapter 11
- Allow different types of command:
  - Evaluation: type-checking and reducing a term
  - Bindings
    - Variable binding: a:Int;
    - Type variable binding: T;
    - Term abbreviation binding: t = succ 0;
    - Type abbreviation binding: T = Nat -> Nat;
- Types can be used without declaration (uninterpreted types)
  x:X
  (lambda a:X. a) x

Review: nameless representation

- What is the nameless representation of the following term?
  - $\lambda x. x\ (\lambda y. x\ y)$

- $\lambda.\ 0\ (\lambda.\ 1\ 0)$

fullsimple, terms

```
type term =
   TmVar of info * int * int
 | TmAbs of info * string * ty * term
 | TmApp of info * term * term
 | ..
```

- Using nameless representation of terms
- The second int for TmVar is used for debugging
  - = the number of items in the context
- The "string" in TmAbs is used for printing

Example: printing terms

```
and printtm_ATerm outer ctx t = match t with
 | TmVar(fi,x,n) ->
     if ctxlength ctx = n then
       pr (index2name fi ctx x)
     else
       pr ("[bad index: " ^ ...
| TmAbs(fi,x,tyT1,t2) ->
     (let (ctx',x') = (pickfreshname ctx x) in
       obox(); pr "lambda ";
       pr x'; pr ":"; printty_Type false ctx tyT1; pr "."; ...
       printtm_Term outer ctx' t2; ...
```

Review: context

- What contexts are used in our course?
  - Mapping names to integers in nameless representation
  - Σ: mapping variables to types
- Can be combined into one
- New contexts in the implementation
  - Type variable binding: marking type variables
  - Term abbreviation binding: Mapping variables to terms (and their types)
  - Type abbreviation binding: Mapping type variables to terms
- All can be combined into one

Only used in printing as a placeholder

```
type binding =
  NameBind
  | TyVarBind
  | VarBind of ty
  | TmAbbBind of term * (ty option)
  | TyAbbBind of ty

type context = (string * binding) list
```

Queried by index

Auxiliary functions for nameless representation

- name2index
  - info->context ->string->int
  - return the index of a name
- index2name
  - info->context ->int->string
  - inverse of the above
- pickfreshname
  - context->string ->(context, string)
  - generate a fresh name using the second parameter as hint

```
type binding =
   NameBind
  | TyVarBind
  | VarBind of ty
  | TmAbbBind of term * (ty option)
  | TyAbbBind of ty

type context = (string * binding) list
```

Exercise fullsimple.nameless

- Construct a term t that is evaluated a term t' in fullsimple, where t' is different from t via only alpha-renaming (i.e., no beta-reduction)

Exercise fullsimple.natlist

- Try the following term in fullsimple and explain why it cannot be typed

  NatList = <nil:Unit, cons:{Nat,NatList}>;

  nil = <nil=unit> as NatList;

  cons = lambda n:Nat. lambda l:NatList. <cons={n,l}> as NatList;

Exercise fullsimple.match

- Add pattern matching for tuples, and test on the following expressions
  - let {x, y, z} = {true, 1, {2}} in z;
  - let {x, y, z} = {true, 1, {2}} in (lambda x:Nat. x) y;
  - let {x, y, z} = let x = 1 in {true, x, {2}} in z;
  - lambda x:Nat. let {x, y} = {true, 1} in x;
  - let x = 0 in let {y, z} = {1, 2} in x;
  - let {y, z} = {1, 2} in let y = 3 in y;

- Part of the code is already provided to you in the following two pages

Partial code for fullsimple.match

- Adding the following line to "type term =" in syntax.ml

  - | TmPLet of info * string list * term * term

- Adding the following lines after line 235 in parser.mly

  - | LET Pattern EQ Term IN Term

  - { fun ctx -> TmPLet($1, $2, $4 ctx, $6 (List.fold_left (fun x y -> addname x y) ctx $2)) }

  - Pattern :

  - LCURLY MetaVars RCURLY

  - { $2 }

  - | LCURLY RCURLY

  - { [] }

- Add the following line to tminfo in syntax.ml

  - | TmPLet(fi,_,_,_) -> fi

Partial code for fullsimple.match

- Adding the following lines to "printtm_Term" in syntax.ml
    - | TmPLet(fi, xs, t1, t2) ->
    -     obox0();
    -     pr "let {";
    -     let rec print xs =
    -       match xs with
    -         x::x'::rest -> pr x; pr ","; print (x'::rest);
    -       | x::[] -> pr x;
    -       | [] -> pr ""; in
    -     print xs;
    -     pr "} = ";
    -     printtm_Term false ctx t1;
    -     print_space(); pr "in"; print_space();
    -     let ctx' = List.fold_left (fun ctx x -> addname ctx x) ctx xs in
    -     printtm_Term false ctx' t2;
    -     cbox()

Exercise fullref.rec_no_fix

- Write plus without using fix or letrec in fullref