CS 11 Ocaml track: lecture 6

- Today:
  - Writing a computer language
  - Parser generators
    - lexers (ocamllex)
    - parsers (ocamlyacc)
  - Abstract syntax trees

Problem (1)

- We want to implement a computer language
- Very complex problem
  - however, much similarity between implementations of different languages
  - We can take advantage of this

Problem (2)

- We will implement a (very, very) simplified version of the Scheme programming language
  - hopefully familiar from CS 4
- We call our version of Scheme "bogoscheme"
  - truth in advertising
  - file names end in .bs

Problem (3)

- Here is the Scheme program we want to be able to run:

```
(define factorial
   (lambda (n)
      (if  (= n 0)
           1
           (* n (factorial (- n 1))))))
(print (factorial 10))
```

- Result should be 3628800

Problem (4)

- Two basic ways to implement languages:
  - interpreter
  - compiler
- Difference?

Interpreters and compilers (1)

- A compiler takes a program (as a file or files of text) and generates a machine-language executable file (or files) from it
- An interpreter takes a program (as a file or files of text), converts it to some internal representation, and executes it immediately

Interpreters and compilers (2)

- Some language processors are intermediate between interpreters and compilers
  - *e.g.* Java
  - Compiler converts source code to Java bytecode
  - Interpreter interprets bytecode
  - JIT compiler can compile bytecode to machine language "on the fly"

Building a compiler

- Compilers have many stages
- Start with source code
  - usually in text files
- Go through a number of transformations
- Eventually output machine code
  - which can be executed directly

Stages of a compiler

- Typical compiler stages:
- Lexer: converts input file (considered as a string) into a sequence of tokens
- Parser: converts sequence of tokens into an "abstract syntax tree" (AST)
- [many other stages]
- Code generator: emits machine language code

Stages of a typical interpreter

- Typical interpreter stages:
- Lexer: converts input file (considered as a string) into a sequence of tokens
- Parser: converts sequence of tokens into an "abstract syntax tree" (AST)
- Evaluator: evaluates AST directly

Stages of a Scheme interpreter

- Similar to typical interpreter, with one extra stage
- Lexer: converts input file (considered as a string) into a sequence of tokens
- Parser: converts sequence of tokens into sequence of "S-expressions"
- Convert S-expressions into AST
- Evaluator: evaluates AST directly
- This is subject of labs 5 and 6

Stages of a simple interpreter

- Lexer: converts input file (considered as a string) into a sequence of tokens
- Parser: recognizes sequences of tokens and *executes them directly*
- Only useful for very simple languages *e.g.* calculators
- Will show an example later

Lexer (1)

- A "lexer" ("lexical analyzer") is the first stage of interpretation or compilation
- Takes raw source code and recognizes syntactically meaningful "tokens"
- Also throws out unnecessary stuff
  - comments
  - whitespace

Lexer (2)

- What is a token?
- Simple literal data values
  - integers, booleans, etc.
- Punctuation
  - *e.g.* left, right parentheses
- Identifiers
  - names of functions, variables
- Keywords, operators (if any)
  - we won't need this

Lexer (3)

- Input:

(define x 10) ; set x to 10

- Possible output of lexer

TOK_LPAREN, TOK_ID("define"), TOK_ID("x"), TOK_INT(10), TOK_RPAREN

- NOTE: whitespace and comments are thrown away
- Sequence of tokens will be handed off to parser

Lexer (4)

- How do we recognize tokens?
- Could write long, boring string-recognizing program by hand
- More modern approach: use regular expressions which match tokens of interest
- Each token type gets its own regular expression (also called a regexp)

Regular expressions (1)

- Regexps are a way to identify a class of strings
- Simplest regexps:

  "foo" → recognizes literal string "foo" only
- Or fixed characters:

  '(' → recognizes left parenthesis only
- Or an arbitrary character:

  _ → matches any single character
- Or the end-of-file character:

  EOF → matches end-of-file character

Regular expressions (2)

- Regexps can also match multiples of other regexps:

regexp * → matches zero or more occurrences of *regexp*

"foo" * → matches "", "foo", "foofoo", ...

regexp + → matches one or more occurrences of *regexp*

regexp? → matches zero or one occurrence of *regexp*

Regular expressions (3)

- Regexps can also match a sequence of smaller regexps:

regexp1 regexp2 → matches *regexp1* followed by *regexp2*

- Can also match any character in a set:

['a' 'b' 'c'] → match any of 'a' 'b' 'c'

['a' - 'z'] → match any char from 'a' to 'z'

[^'a' 'b' 'c'] → match any char except 'a', 'b', 'c'

Regular expressions (4)

- "Or patterns":

regexp1 | *regexp2* → matches either *regexp1* or *regexp2*

- Some other regexp patterns as well
- See ocamllex manual for full list
- NOTE: regexp syntax varies between language implementations
  - though you only need to know Ocaml version

Lexer generators (1)

- Writing lexers by hand is boring
  - Also easily automated
- Modern approach:
  - describe lexer in a high-level specification in a special file
  - use a special program to convert this lexer into code for the language needed

Lexer generators (2)

- In Ocaml:
- Write lexer specification in a file ending in *".mll"* (ML Lexer)
  - this is NOT Ocaml code
  - but it's fairly close
- The ocamllex program converts this into Ocaml code (file ending in ".ml")
- ".ml" file compiled normally

Lexer generators (3)

- Will go through details of ocamllex file format when we go through the example

Parsing Scheme (1)

- Most language implementations have a *parser* which
  - takes input from output of lexer (i.e. sequence of tokens)
  - converts into AST (abstract syntax tree)
- We will do it slightly differently
  - Will generate "S-expressions" from tokens
  - Will convert S-expressions into AST

Parsing Scheme (2)

- Advantage of S-expressions
  - *Extremely* easy to write the parser!
  - Almost the simplest possible parser imaginable
  - Can change AST without having to rewrite the parser
- Disadvantage of S-expressions
  - Also have to write the converter from S-expressions to AST

S-expressions (1)

- S-expression stands for "symbolic expression"
- Basically a nested list of symbols
- Simple, regular format
- Very easy to parse

S-expressions (2)

- Definition of S-expression:
- An S-expression is either
  - an atom
  - a list of S-expressions
- Note the recursive definition!
  - S-expressions defined in terms of themselves
  - Similar to recursive data type defs in Ocaml

Atoms

- An "atom" is a single indivisible syntactic entity
- Examples:
  - a boolean
  - an integer
  - an identifier
- NOT the same as a token
  - a "left parenthesis" is not an atom

S-expressions (3)

- Example of S-expression:
- Source code:

```
(define x 10)
```

- S-expression version:

```
LIST[ ATOM["define"] ATOM["x"] ATOM[10] ]
```

S-expressions (4)

- Better S-expression version:

```
Expr_list(Expr_atom(Atom_id("define")),
   Expr_atom(Atom_id("x")),
   Expr_atom(Atom_int(10)))
```

- This version can be written as an Ocaml datatype

S-expressions in Ocaml (1)

- In file sexpr.mli:

```
type atom =
   | Atom_unit
   | Atom_bool of bool
   | Atom_int of int
   | Atom_id of string
```

S-expressions in Ocaml (2)

- In file sexpr.mli:

```
type expr =
   | Expr_atom of atom
   | Expr_list of expr list
```

- That's all there is to S-expressions!
- This is what parser has to generate from a sequence of tokens

S-expr version of our program

```
LIST[ ATOM["define"] ATOM["factorial"]
  LIST[ ATOM["lambda"] LIST[ ATOM["n"] ]
    LIST[ ATOM["if"]
      LIST[ ATOM["="] ATOM["n"] ATOM[0] ]
         ATOM[1]
         LIST[ ATOM["*"] ATOM["n"]
            LIST[ ATOM["factorial"]
              LIST[ ATOM["-"] ATOM["n"] ATOM[1] ] ] ] ] ] ]
LIST[ ATOM["print"]
  LIST[ ATOM["factorial"] ATOM[10] ] ]
```

Parser generators (1)

- Like lexers, can write parser by hand, but it's extremely boring and error-prone
- Instead, have programs called "parser generators" which can do this given a high-level specification of the parser
- Ocaml parser generator is called ocamlyacc
  - yacc originally meant "yet another compiler compiler"

Parser generators (2)

- Parser generator specification includes
  - description of the different token types
    - their names
    - the type of any associated data
  - description of the grammar of the language as a "context free grammar"
  - Sometimes some other stuff
    - *e.g.* operator precedence declarations
    - We won't need this

Parser generators (3)

- Parser generator specification is in a file ending with *".mly"*
- Stands for "ML Yacc" file
- Similar to ocamllex file; has its own format which is different from Ocaml source code
- Will see example later

Context-free grammars (1)

- High-level description of language syntax
- Two elements:
  - terminals -- correspond to tokens
  - nonterminals -- (usually) correspond to some kind of expression in the language
- One kind of "rule" called a "production"
  - describes how each nonterminal corresponds to a sequence of other nonterminals and/or terminals

Context-free grammars (2)

- There is also an entry point, which is a nonterminal which may represent
  - an entire program (typical for compilers)
  - an entire expression (typical for interpreters)
- Our entry point will be a single S-expression, or None if EOF token is encountered
  - type will be Sexpr.expr option

Context-free grammars (3)

- Context-free grammars often very close to Ocaml type definitions
  - which is one reason Ocaml is a nice language to write language interpreters/ compilers in
- Will see example of CFG/parser generator in the example later

Abstract Syntax Trees (1)

- An Abstract Syntax Tree (AST) is the final goal of parsing
- An AST is a representation of the syntax of a program or expression as an Ocaml datatype
- Includes everything relevant to interpreting the program
- Does not include irrelevant stuff
  - whitespace, comments, etc.

Abstract Syntax Trees (2)

- Our AST is defined in ast.mli:

```
type id = string
type expr =
  | Expr_unit
  | Expr_bool      of bool
  | Expr_int       of int
  | Expr_id        of id
  | Expr_define  of id * expr
  | Expr_if        of expr * expr * expr
  | Expr_lambda of id list * expr list
  | Expr_apply    of expr * expr list
```

Abstract Syntax Trees (3)

- The AST defines all valid expressions in the language
- First few cases represent expressions consisting of a single data value:

```
type expr =
  | Expr_unit                    (* unit value *)
  | Expr_bool   of bool          (* boolean value *)
  | Expr_int    of int           (* integer *)
  | ...
```

- N.B. Expr_unit value is like the Ocaml unit value

Abstract Syntax Trees (4)

- Identifiers are just strings:

```
type id = string  (* id is a type alias for "string" *)
type expr =
  | ...
  | Expr_id of id
  | ...
```

- Expr_id expressions consist of a single identifier
  - for instance, the name of a function

Abstract Syntax Trees (5)

- "define" expressions:

```
type expr =
  | ...
  | Expr_define of id * expr
  | ...
```

- id represents the name being defined
- expr represents the thing it's defined to be

Abstract Syntax Trees (6)

- "if" expressions:

```
type expr =
  | ...
  | Expr_if  of expr * expr * expr
  | ...
```

- if expression consists of three subexpressions
  - test case (always evaluated)
  - "then" case (evaluated if test evaluates to true)
  - "else" case (evaluated if test evaluates to false)

Abstract Syntax Trees (7)

- "lambda" expressions

```
type expr =
  | ...
  | Expr_lambda of id list * expr list
  | ...
```

- lambda expressions represent an anonymous function
- id list is the list of formal parameters of the function
- expr list is the body of the function

Abstract Syntax Trees (8)

- "apply" expressions represent function application

```
type expr =
  | ...
  | Expr_apply of expr * expr list
```

- expr represents the function being applied
  - could be an identifier
  - could be a lambda expression
- expr list represents the arguments of the function application

AST version of our program

```
DEFINE["factorial"
  LAMBDA[(ID["n"])
    IF[APPLY[ID["="] ID["n"] INT[0]]
         INT[1]
         APPLY[ID[ "*" ] ID["n"]
           APPLY[ID["factorial"]
             APPLY[ID["-"] ID["n"] INT[1]]]]]]]
APPLY[ID["print"] APPLY[ID["factorial"] INT[10]]]
```

Example -- calculator language

- We will walk through the "calculator" example from the ocamllex/ocamlyacc documentation
- In the process, will see the format of ocamllex/ocamlyacc files
- This example is NOT a typical language
  - we don't generate an AST
  - instead, just execute code directly after parsing
  - nevertheless, principles are the same

parser.mly file, part 1

```
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN  /* left, right parentheses */
%token EOL            /* end of line */
%left PLUS MINUS      /* lowest precedence */
%left TIMES DIV       /* medium precedence */
%nonassoc UMINUS      /* highest precedence */
%start main           /* the entry point */
%type <int> main
%%
```

token definitions

parser.mly file, part 1

```
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN  /* left, right parentheses */
%token EOL            /* end of line */
%left PLUS MINUS      /* lowest precedence */
%left TIMES DIV       /* medium precedence */
%nonassoc UMINUS      /* highest precedence */
%start main           /* the entry point */
%type <int> main
%%
```

operators and precedences

parser.mly file, part 1

```
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN  /* left, right parentheses */
%token EOL            /* end of line */
%left PLUS MINUS      /* lowest precedence */
%left TIMES DIV       /* medium precedence */
%nonassoc UMINUS      /* highest precedence */
%start main           /* the entry point */
%type <int> main
%%
```

entry point

parser.mly file, part 1

```
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN  /* left, right parentheses */
%token EOL            /* end of line */
%left PLUS MINUS      /* lowest precedence */
%left TIMES DIV       /* medium precedence */
%nonassoc UMINUS      /* highest precedence */
%start main           /* the entry point */
%type <int> main
%%
```

type of entry point

parser.mly file, part 1

```
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN  /* left, right parentheses */
%token EOL            /* end of line */
%left PLUS MINUS      /* lowest precedence */
%left TIMES DIV       /* medium precedence */
%nonassoc UMINUS      /* highest precedence */
%start main           /* the entry point */
%type <int> main
%%
```

start of next section

parser.mly file, part 2

```
main:  expr EOL { $1 } ;
```

entry point

```
expr:   INT { $1 }
        | LPAREN expr RPAREN { $2 }
        | expr PLUS expr { $1 + $3 }
        | expr MINUS expr { $1 - $3 }
        | expr TIMES expr { $1 * $3 }
        | expr DIV expr { $1 / $3 }
        | MINUS expr %prec UMINUS { - $2 } ;
```

parser.mly file, part 2

nonterminals

```
main:  expr EOL { $1 } ;
expr:  INT { $1 }
     | LPAREN expr RPAREN { $2 }
     | expr PLUS expr { $1 + $3 }
     | expr MINUS expr { $1 - $3 }
     | expr TIMES expr { $1 * $3 }
     | expr DIV expr { $1 / $3 }
     | MINUS expr %prec UMINUS { - $2 } ;
```

parser.mly file, part 2

terminals

```
main:  expr EOL { $1 } ;
expr:   INT { $1 }
      | LPAREN expr RPAREN { $2 }
      | expr PLUS expr { $1 + $3 }
      | expr MINUS expr { $1 - $3 }
      | expr TIMES expr { $1 * $3 }
      | expr DIV expr { $1 / $3 }
      | MINUS expr %prec UMINUS { - $2 } ;
```

parser.mly file, part 2

productions

```
main:  expr EOL { $1 } ;

expr:   INT { $1 }
      | LPAREN expr RPAREN { $2 }
      | expr PLUS expr { $1 + $3 }
      | expr MINUS expr { $1 - $3 }
      | expr TIMES expr { $1 * $3 }
      | expr DIV expr { $1 / $3 }
      | MINUS expr %prec UMINUS { - $2 } ;
```

parser.mly file, part 2

actions

```
main:  expr EOL { $1 } ;
expr:   INT { $1 }
      | LPAREN expr RPAREN { $2 }
      | expr PLUS expr { $1 + $3 }
      | expr MINUS expr { $1 - $3 }
      | expr TIMES expr { $1 * $3 }
      | expr DIV expr { $1 / $3 }
      | MINUS expr %prec UMINUS { - $2 } ;
```

parser.mly file, part 2

precedence specifier

```
main:  expr EOL { $1 } ;
expr:   INT { $1 }
      | LPAREN expr RPAREN { $2 }
      | expr PLUS expr { $1 + $3 }
      | expr MINUS expr { $1 - $3 }
      | expr TIMES expr { $1 * $3 }
      | expr DIV expr { $1 / $3 }
      | MINUS expr %prec UMINUS { - $2 } ;
```

How parsing works

- Parser calls lexer to get tokens one at a time
- Parser checks to see if the left-hand side of a production (before the colon) can be matched
  - if so, it executes the corresponding action (which is (almost) Ocaml code)
  - if not, it pushes the token onto a stack

Shifting and reducing (1)

- Two fundamental actions of the parser: shifting and reducing
- Shifting:
  - putting a new token onto the stack
- Reducing:
  - popping off all the tokens on the stack corresponding to the RHS of a given production
  - pushing the LHS of the production onto the stack
    - along with its associated value
  - executing the action of that production

Shifting and reducing (2)

- Can have cases where grammar is ambiguous
- Ambiguity → when rules allow
  - more than one different reduction (called a reduce/reduce conflict)
  - either a shift or a reduction (called a shift/reduce conflict)
- Shift/reduce conflicts are resolved by choosing the shift
- Reduce/reduce conflicts are unresolvable
  - means grammar is completely broken

Parsing actions

- Parsing actions are Ocaml code inside curly brackets
- $1, $2, $3 values represent the value associated with the corresponding location in the RHS of the production

  expr PLUS expr { $1 + $3 }
- Here, $1 represents the value of the first expr
- $3 represents the value of the second expr
- $2 would represent value of PLUS token
  - meaningless (PLUS token has no value)

Example: 2 + 2 = 4

- Input: 2 + 2 <return>
- Tokens: INT(2) PLUS INT(2) EOL
- Parser
  - shifts INT(2)
  - reduces INT(2) to expr with value 2
  - shifts PLUS
  - shifts INT(2)
  - reduces INT(2) to expr with value 2
  - reduces expr PLUS expr to expr with value 4
  - shifts EOL
  - reduces expr EOL to main with value 4

Note

- In more realistic language (like in lab 5) would *not* compute results inside parser actions
- Instead, would generate AST
- For our example, might have *e.g.*

  | expr PLUS expr { Add_expr($1, $3) }
- (Assuming that Add_expr is one constructor for calculator AST)
- Could evaluate AST later
- Complex languages need AST to be interpreted correctly

lexer.mll, part 1

```
{
    open Parser
    exception Eof
}
```

"Header"

```
(* Continued on next slide *)
```

- Header is just code that gets copied to the front of the .ml file that gets generated
- Usually just some declarations (can be empty)

lexer.mll, part 2

Name of lexer

```
rule token = parse
    [' ' '\t'] { token lexbuf } (* skip blanks *)
  | ['\n' ]  { EOL }
  | ['0'-'9']+ as lxm { INT(int_of_string lxm) }
  | '+' { PLUS }
  | '-'  { MINUS }
  | '*'  { TIMES }
  | '/'  { DIV }
  | '('  { LPAREN }
  | ')'  { RPAREN }
  | eof { raise Eof }
```

lexer.mll, part 2

regular expressions

```
rule token = parse
    [' ' '\t'] { token lexbuf } (* skip blanks *)
  | ['\n' ]  { EOL }
  | ['0'-'9']+ as lxm { INT(int_of_string lxm) }
  | '+' { PLUS }
  | '-'  { MINUS }
  | '*'  { TIMES }
  | '/'  { DIV }
  | '('  { LPAREN }
  | ')'  { RPAREN }
  | eof { raise Eof }
```

lexer.mll, part 2

actions

```
rule token = parse
    [' ' '\t'] { token lexbuf }  (* skip blanks *)
  | ['\n' ]  { EOL }
  | ['0'-'9']+ as lxm { INT(int_of_string lxm) }
  | '+' { PLUS }
  | '-'  { MINUS }
  | '*'  { TIMES }
  | '/'  { DIV }
  | '('  { LPAREN }
  | ')'  { RPAREN }
  | eof { raise Eof }
```

Strategy for writing lab 5 (1)

- Don't worry about S-expression to AST conversion at first
- Write parser with dummy actions
  - make sure ocamlyacc doesn't give any errors
- Then write lexer (should be easy)
  - make sure ocamllex doesn't give any errors
- Compile lexer_test and parser_test programs
- Test them on factorial.bs input file

Strategy for writing lab 5 (2)

- Once parser is working, work on Sexpr to AST conversion (in file ast.ml)
- This is the hardest part of the lab
  - have to handle lots of different cases
- USE PATTERN MATCHING!
- Code should work on ANY valid input, not just on factorial.bs file

Next time

- We'll go over lab 6, which is the rest of the mini-Scheme language implementation