CMSC 330: Organization of Programming Languages

Parsing & ASTs
Functional Programming with OCaml

Last Lecture
- Learned a little about parsing
  - Recursive descent parser
  - Predictive parsing using FIRST sets
- Rewriting grammars for predicative parsing
  - Left factoring
  - Eliminating left recursion

This Lecture
- Choosing the right parser
  - What are the pros and cons of our approach
- Abstract syntax trees (ASTs)
- Functional programming
  - Features of functional languages
  - OCaml basics

Recursive Descent Parser Review
- Goal
  - Determine if we can produce the string to be parsed from the grammar's start symbol
- Top-down, predictive parser
  - Works with unambiguous, right-recursive grammars
  - Keeps track of the lookahead (the next input symbol)
- Compare lookahead with FIRST sets to choose the production to apply
  - Recursively call match(a) or parse_N functions for the RHS of the selected production
- Parse fails if lookahead does not match FIRST
Tradeoffs with Other Approaches

- Recursive descent parsers are easy to write
  - The formal definition is a little clunky, but if you follow the code then it’s almost what you might have done if you weren’t told about grammars formally
  - They’re unable to handle certain kinds of grammars
- Recursive descent is good for a simple parser
  - Though tools can be fast if you’re familiar with them
  - Can implement top-down predictive parsing as a table-driven parser
    - By maintaining an explicit stack to track progress

More powerful techniques need tool support
- Can take time to learn tools
- Main alternative is bottom-up, shift-reduce parser
  - Replaces RHS of production with LHS (nonterminal)
  - Example grammar
    - $S \rightarrow aA$, $A \rightarrow Bc$, $B \rightarrow b$
  - Example parse
    - $abc \Rightarrow aBc \Rightarrow aA \Rightarrow S$
    - Derivation happens in reverse
    - Something to look forward to in CMSC 430

What’s Wrong With Parse Trees?

- Parse trees contain too much information
  - Example
    - Parentheses
    - Extra nonterminals for precedence
  - This extra stuff is needed for parsing
- But when we want to reason about languages
  - Extra information gets in the way (too much detail)

Abstract Syntax Trees (ASTs)

- An abstract syntax tree is a more compact, abstract representation of a parse tree, with only the essential parts
Abstract Syntax Trees (cont.)

Intuitively, ASTs correspond to the data structure you’d use to represent strings in the language
- Note that grammars describe trees
- So do OCaml datatypes (which we’ll see later)
- \[ E \rightarrow a \mid b \mid c \mid E+E \mid E-E \mid E*E \mid (E) \]

Producing an AST

To produce an AST, we can modify the `parse()` functions to construct the AST along the way
- `match(a)` returns an AST node (leaf) for `a`
- `Parse_A` returns an AST node for `A`
  - AST nodes for RHS of production become children of LHS node

Example

- \[ S \rightarrow aA \]

```java
Node parse_S() {
    Node n1, n2;
    if (lookahead == "a") {
        n1 = match("a");
        n2 = parse_A();
        return new Node(n1, n2);
    }
}
```

The Compilation Process

Now on to functional programming...

Features of ML (a “mostly functional” language)
- Higher-order functions
- Type inference

Objective Caml (OCaml)
- Language basics
- Lists
- Pattern matching

This is not a camel!
Dialects of ML

- ML (Meta Language)
  - Univ. of Edinburgh, 1973
  - Part of a theorem proving system LCF
    - The Logic of Computable Functions

- SML/NJ (Standard ML of New Jersey)
  - Bell Labs and Princeton, 1990
  - Now Yale, AT&T Research, Univ. of Chicago, etc…

- OCaml (Objective CAML)
  - INRIA, 1996
  - French Nat'l Institute for Research in Computer Science

Dialects of ML (cont.)

- Other dialects
  - MoscowML, ML Kit, Concurrent ML, etc…
  - SML/NJ and OCaml are most popular

- Languages all have the same core ideas
  - But small and annoying syntactic differences
  - So you should not buy a book with ML in the title
    - Because it probably won’t cover OCaml

Features of ML

- “Mostly functional”
  - Some assignments

- Higher-order functions
  - Functions can be parameters and return values

- Type inference
  - No need to write types in the source language
    - But the language is statically typed
  - Supports parametric polymorphism
    - Generics in Java, templates in C++

Features of ML (cont.)

- Data types and pattern matching
  - Convenient for certain kinds of data structures

- Exceptions

- Garbage collection
Functional Languages

- In a pure functional language
  - Every program is just an expression evaluation
    ```ocaml
    let add1 x = x + 1;;
    let rec add (x,y) = if x=0 then y else add(x-1, add1(y));;
    add(2,3) = add(1,add1(3)) = add(0,add1(add1(3)))
               = add1(add1(3)) = add1(3+1) = 3+1+1
               = 5
    ```

Functional Languages (cont.)

- OCaml has similar basic behavior
  - Program = expression evaluation

- But has additional features
  - To ease the programming process
  - Features support
    - Less emphasis on data storage
    - More emphasis on function execution

A Small OCaml Program – Things to Notice

- Use `let` to bind variables
- No type declarations
- Need to use correct `print` function (OCaml also has `printf`)
- Use `(*)` for comments (may nest)
- Line breaks, spacing ignored (like C, C++, Java, not like Ruby)

Run, OCaml, Run

- OCaml programs can be compiled using `ocamlc`
  - Produces `.cmo` ("compiled object") and `.cmi` ("compiled interface") files
    - We’ll talk about interface files later
  - By default, also links to produce executable `a.out`
    - Use `-o` to set output file name
    - Use `-c` to compile only to `.cmo/.cmi` and not to link
    - You can use a `Makefile` if you need to compile your files
Run, OCaml, Run (cont.)

• Compiling & running the previous small program

    ocam1.ml
    (* Small OCaml program *)
    let x = 37;;
    let y = x + 5;;
    print_int y;;
    print_string "\n";;

    % ocamlc ocam1.ml
    % .a.out
    42
    %

Run, OCaml, Run (cont.)

Expressions can also be typed and evaluated at the top-level

```
# 3 + 4;;
- : int = 7
# let x = 37;;
val x : int = 37
# x;;
- : int = 37
# let y = 5;;
val y : int = 5
# let z = 5 + x;;
val z : int = 42
# print_int z;;
42- : unit = ()
# print_string "Colorless green ideas sleep furiously";;
Colorless green ideas sleep furiously : unit = ()
```

Basic Types in OCaml

• Read e : t as “expression e has type t”

```
42 : int
true : bool
"hello" : string
‘c’ : char
3.14 : float
() : unit (* don’t care value *)
```

• OCaml has static types to help you avoid errors
  • Note: Sometimes the messages are a bit confusing
    # 1 + true;;
    This expression has type bool but is here used with type int
  • Watch for the underline as a hint to what went wrong
  • But not always reliable
More on the Let Construct

- *let* is more often used for local variables
  - *let x = e1 in e2* means
    - Evaluate *e1*
    - Then evaluate *e2*, with *x* bound to result of evaluating *e1*
    - *x* is not visible outside of *e2*

```plaintext
let pi = 3.14 in pi *. 3.0 *.* 3.0;;
```
- *error*
- *bind pi in body of let*
- *floating point multiplication*

More on the Let Construct (cont.)

- Compare to similar usage in Java/C

```plaintext
let pi = 3.14 in
  pi *. 3.0 *.* 3.0;;
pi;;
```

- In the top-level, omitting *in* means "from now on"

  # let pi = 3.14;;
  (* pi is now bound in the rest of the top-level scope *)

Nested Let

- Uses of *let* can be nested

```plaintext
let pi = 3.14 in
  let r = 3.0 in
    pi *. r *. r;;
(* pi, r no longer in scope *)
```

Defining Functions

- Use *let* to define functions
  - List parameters after function name

```plaintext
let next x = x + 1;;
next 3;;
let plus (x, y) = x + y;;
plus (3, 4);;
```
- No parentheses on function calls
- No return statement
Local Variables

- You can use `let` inside of functions for locals
  ```ocaml
  let area r =
  pi *. r *. r
  ```
- And you can use as many `lets` as you want
  ```ocaml
  let area d =
  let pi = 3.14 in
  let r = d / 2.0 in
  pi *. r *. r
  ```

Function Types

- In OCaml, `->` is the function type constructor
  - The type `t1 -> t2` is a function with argument or domain type `t1` and return or range type `t2`
- Examples
  - `let next x = x + 1 (* type int -> int *)`
  - `let fn x = (float_of_int x) *. 3.14 (* type int -> float *)`
  - `print_string (* type string -> unit *)`
- Type a function name at top level to get its type

Type Annotations

- The syntax `(e : t)` asserts that “`e` has type `t`”
  - This can be added anywhere you like
    ```ocaml
    let (x : int) = 3
    let z = (x : int) + 5
    ```
- Use to give functions parameter and return types
  ```ocaml
  let fn (x:int):float = (float_of_int x) *. 3.14
  ```
  - Note special position for return type
  - Thus `let g x:int = ...` means `g` returns `int`
- Very useful for debugging
  - Especially for more complicated types

;; versus ;

- `;;` ends an expression in the top-level of OCaml
  - Use it to say: “Give me the value of this expression”
  - Not used in the body of a function
  - Not needed after each function definition
    - Though for now it won’t hurt if used there
- `e1; e2` evaluates `e1` and then `e2`, and returns `e2`
  ```ocaml
  let p (s,t) = print_int s; print_int t; "Done!"
  ```
  - Notice no ; at end
  - `;` is a separator, not a terminator
  - `Invoking p (1,2)`
    - Prints “1 2”
    - Returns “Done!”
Lists in OCaml

- The basic data structure in OCaml is the list
  - Lists are written as [e1; e2; ...; en]
  - `# [1;2;3]`
  - `: int list = [1;2;3]`
  - Notice type of list is `int list`
    - Lists must be homogeneous

Lists in OCaml (cont.)

- More on OCaml lists
  - The empty list is `[ ]`
    - `# [ ]`
    - `: 'a list`
  - The `'a` means "a list containing anything"
    - We'll find out more about this later
  - Warning: Don't use a comma instead of a semicolon
    - Means something different (we'll see in a bit)

Consider a Linked List in C

```c
struct list {
    int elt;
    struct list *next;
};
...
struct list *l;...
i = 0;
while (l != NULL) {
    i++;
    l = l->next;
}
```

Lists in OCaml are Linked

- `[1;2;3]` is represented above
  - A nonempty list is a pair (element, rest of list)
  - The element is the head of the list
  - The pointer is the tail or rest of the list
    - which is itself a list!
  - Thus in math a list is either
    - The empty list `[ ]`
    - Or a pair consisting of an element and a list
      - This recursive structure will come in handy shortly
Lists are Linked (cont.)

:: prepends an element to a list
- h::t is the list with h as the element at the beginning and t as the "rest"
- :: is called a constructor, because it builds a list
- Although not emphasized, :: does allocate memory

Examples
3::[] (* The list [3]*)
2::(3::[]) (* The list [2; 3]*)
1::(2::(3::[])) (* The list [1; 2; 3]*)

More Examples

# let y = [1;2;3] ;;
val y : int list = [1; 2; 3]
# let x = 4::y ;;
val x : int list = [4; 1; 2; 3]
# let z = 5::y ;;
val z : int list = [5; 1; 2; 3]
- not modifying existing lists, just creating new lists
# let w = [1;2]::y ;;
This expression has type int list but is here used with type int list list
- The left argument of :: is an element
- Can you construct a list y such that [1;2]:y makes sense?

Lists of Lists

- Lists can be nested arbitrarily
  - Example: [ [9; 10; 11]; [5; 4; 3; 2] ]
    - Type = int list

Practice

- What is the type of
  - [1;2;3] int list
  - [[[; ;]; [1;3;2;4]]] float list list list
  - let func x = x::(0::[]) int -> int list
Pattern Matching

To pull lists apart, use the `match` construct

```latex
match e with p1 -> e1 | ... | pn -> en
```

- p1...pn are patterns made up of
  - [], ::, and pattern variables
- `match` finds the first pk that matches shape of e
  - Then ek is evaluated and returned
  - During evaluation of pk, pattern variables in pk are bound to the corresponding parts of e

Pattern Matching Example

Match syntax

```latex
match e with p1 -> e1 | ... | pn -> en
```

Code 1

```latex
let is_empty l = match l with
    [] -> true
  | (h::t) -> false
```

Outputs

- `is_empty []` (* evaluates to true *)
- `is_empty [1]` (* evaluates to false *)
- `is_empty [1;2]`(* evaluates to false *)

Pattern Matching Example (cont.)

```latex
let hd l = match l with (h::t) -> h
```

Outputs

- `hd [1;2;3]`(* evaluates to 1 *)
- `hd [1;2]` (* evaluates to 1 *)
- `hd [1]` (* evaluates to 1 *)
- `hd []` (* Exception: Match failure *)

Pattern Matching Example (cont.)

```latex
let tl l = match l with (h::t) -> t
```

Outputs

- `tl [1;2;3]`(* evaluates to [2;3] *)
- `tl [1;2]` (* evaluates to [2] *)
- `tl [1]` (* evaluates to [ ] *)
- `tl []` (* Exception: Match failure *')
Pattern Matching – Wildcards

- An underscore \(_\) is a wildcard pattern
  - Matches anything
  - Doesn’t add any bindings
  - Useful when you want to know something matches
    » But don’t care what its value is

- In previous examples
  - Many values of \(h\) or \(t\) ignored
  - Can replace with wildcard \(_\)
  - Code behavior is identical

Pattern Matching – Wildcards (cont.)

- Code using \(_\)
  - let is_empty l = match l with
    - [] -> true
    - (_::_) -> false
  - let hd l = match l with (h::_) -> h
  - let tl l = match l with (_::t) -> t

- Outputs
  - is_empty[1](\(*\) evaluates to false \(*\))
  - is_empty[ ](\(*\) evaluates to true \(*\))
  - hd [1;2;3](\(*\) evaluates to 1 \(*\))
  - tl [1;2;3](\(*\) evaluates to [2;3] \(*\))
  - hd [1](\(*\) evaluates to 1 \(*\))
  - tl [1](\(*\) evaluates to [ ] \(*\))

Pattern Matching – Missing Cases

- When pattern is defined
  - OCaml will warn you about non-exhaustive matches
- When pattern is used
  - Exceptions for inputs that don’t match any pattern
- Example
  ```ocaml
  # let hd l = match l with (h::_) -> h;;
  Warning: this pattern-matching is not exhaustive.
  # hd [ ];;
  Exception: Match_failure ("", 1, 11).
  ```

Pattern Matching – An Abbreviation

- let f p = e, where p is a pattern
  - is shorthand for let f x = match x with p -> e
- Examples
  - let hd (h::_) = h
  - let tl (_::t) = t
  - let f (x::y::_) = x + y
  - let g [x; y] = x + y
- Useful if there’s only one acceptable input
Pattern Matching – Lists of Lists

- Can pattern match on lists of lists as well
- Examples
  - let addFirsts
    \((x::[]): (y::[]): []\) = \(x + y\)
    addFirsts \([\{1;2\};\{4;5\};\{7;8;9\}]\) = 5
  - let addFirstSecond
    \((x::[]): (y::[]): []\) = \(x + y\)
    addFirstSecond \([\{1;2\};\{4;5\};\{7;8;9\}]\) = 6
- Note – you probably won’t do this much or at all
  - You’ll mostly write recursive functions over lists instead