ASTs, Objective CAML, and Ocamlyacc

Stephen A. Edwards

Columbia University

Fall 2008

Parsing and Syntax Trees

Parsing decides if the program is part of the language.

Not that useful: we want more than a yes/no answer.

Like most, parsers generated by *ocamlyacc* can include *actions*: pieces of code that run when a rule is matched.

Top-down parsers: actions executed during parsing rules.

Bottom-up parsers: actions executed when rule is "reduced." (ocamlyacc)

Actions

Simple languages can be interpreted just with parser actions.

```
%token PLUS MINUS TIMES DIVIDE
%token EOF
%token <int> LIT

%left PLUS MINUS
%left TIMES DIVIDE

%start top
%type <int> top

%%

top : expr EOF { $1 }

expr
  : expr PLUS expr   { $1 + $3 }
  | expr MINUS expr  { $1 - $3 }
  | expr TIMES expr  { $1 * $3 }
  | expr DIVIDE expr { $1 / $3 }
  | LIT              { $1 }
```

```
{ open Calc_parse }

rule token = parse
  [' ' '\n'] { token lexbuf }
| '+'        { PLUS }
| '-'        { MINUS }
| '*'        { TIMES }
| '/'        { DIVIDE }
| ['0'-'9']+ as s
     { LIT(int_of_string s) }
| eof        { EOF }
```

```
let _ =
  let lexbuf =
    Lexing.from_channel stdin in
  print_int (Calc_parse.top
              Calc_scan.token
              lexbuf);
  print_newline ()
```

Actions

Even in a parser for an interpreter, actions usually build a data structure that represents the program.

Separates parsing from translation.

Makes modification easier by minimizing interactions.

Allows parts of the program to be analyzed in different orders.

Bottom-up parsers can only build bottom-up data structures.

Children known first, parents later.

Context of an object only established later.

What To Build?

Typically, an Abstract Syntax Tree that represents the program.

Represents the syntax of the program almost exactly, but easier for later passes to deal with.

Punctuation, whitespace, other irrelevant details omitted.

Abstract vs. Concrete Trees

Like scanning and parsing, objective is to discard irrelevant details.

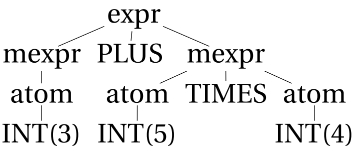
E.g., comma-separated lists are nice syntactically, but later stages probably just want lists.

AST structure almost a direct translation of the grammar.

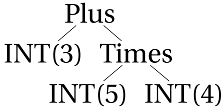
Abstract vs. Concrete Trees

```
expr   : mexpr PLUS mexpr { ... };
mexpr  : atom TIMES atom  { ... };
atom   : INT { ... };
```

```
3 + 5 * 4
```



Concrete Parse Tree



Abstract Syntax Tree

Part I

Designing ASTs

Designing an AST Structure

Sequences of things

Removing unnecessary punctuation

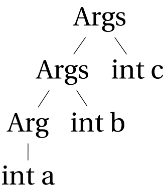
Additional grouping

How to factor

One Way to Handle Comma-Separated Lists

```
type args = Arg of arg | Args of args * arg
type arg = ...

args    : LPAREN arglist RPAREN { $2 };
arglist : arglist COMMA arg     { Args($1, $3) }
        | arg                   { Arg($1) };
```

```
int gcd(int a, int b, int c)
```



Drawbacks:

Many unnecessary nodes

Branching suggests recursion

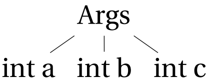
Harder for later routines to get the data they want

A Better Way to Handle Comma-Separated Lists

Better to choose a simpler structure for the tree: use lists

```
type args = Args of arg list
type arg = ...

args     : LPAREN arglist RPAREN { Args(List.rev $2) };
arglist : arglist COMMA arg      { $3::$1 }
        | arg                    { [$1] }
```

```
int gcd(int a, int b, int c)
```



Removing Unnecessary Punctuation

Punctuation makes the syntax readable, unambiguous.

Information represented by structure of the AST

Things typically omitted from an AST

- Parentheses
  Grouping and precedence/associativity overrides
- Separators (commas, semicolons)
  Mark divisions between phrases. Probably want a list of items in the AST.
- Extra keywords
  while-do, if-then-else. Just want a "While" constructor with two children.

How to factor

Two possible ways to represent binary operators:

```
type expr =
  Plus of expr * expr
| Minus of expr * expr
| Times of expr * expr
| ...
```

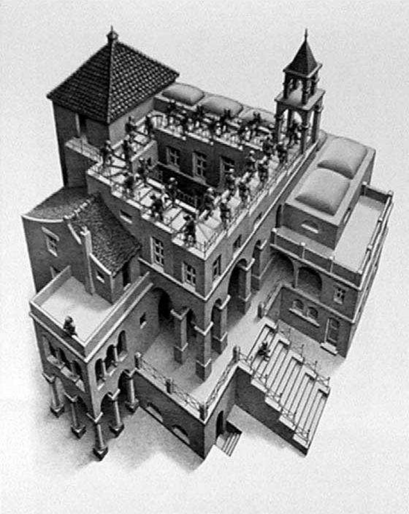
```
type binop = Plus | Minus | Times | ...

type expr =
  Binop of expr * binop * expr
| ...
```

Each has advantages and disadvantages

Main question is how nice the later code looks. Is each operator a special case, or can you handle them all at once?

Part II

Walking ASTs



M. C. Escher, *Ascending and Descending*, 1960

It's easy in O'Caml

```
type operator = Add | Sub | Mul | Div

type expr =
    Binop of expr * operator * expr
  | Lit of int

let rec eval = function
    Lit(x) -> x
  | Binop(e1, op, e2) ->
      let v1 = eval e1 and v2 = eval e2 in
      match op with
        Add -> v1 + v2
      | Sub -> v1 - v2
      | Mul -> v1 * v2
      | Div -> v1 / v2
```

Comments on ASTs

Two ways to handle optional clauses:

```
type stmt =
   If of expr * stmt * stmt option
 | ...

let rec eval = function
    If(e, s1, None) -> ...
  | If(e, s1, Some(s2)) -> ...

(* or *)

let rec eval = function
  If(e, s1, s2) -> ...
     match s2 with
       None -> ...
     | Some(s) -> ...
| ...
```

```
type stmt =
    If of expr * stmt
  | IfElse of expr * stmt * stmt
  | ...

let rec eval = function
    If(e, s) -> ...
  | IfElse(e, s1, s2) -> ...
```