Compilation 2012

Abstract Syntax Trees

Asger Feldthaus
Jan Midtgaard
Michael I. Schwartzbach
Aarhus University

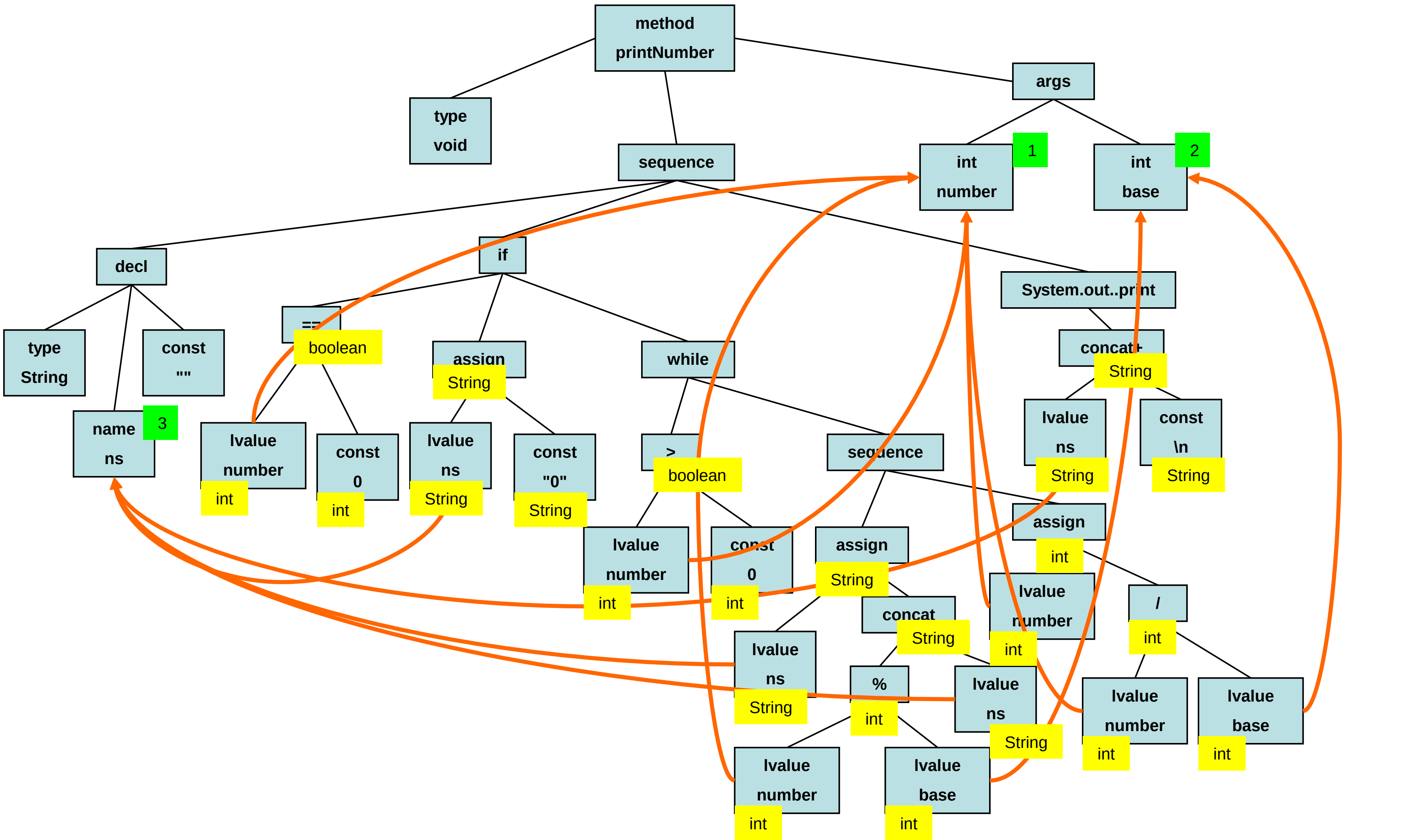
Parsing is not enough

- A parser recognizes whether a program fits to a context-free (or LR(1) or …) grammar
- This is nice but not what we are really after
- We want to *do something* with the program, e.g., generate code
- There are basically two options:
  - Do something during parsing (e.g., a syntax-directed, one-pass compiler)
  - Do something after parsing

Syntax Trees

- Instead of compiling in one pass we can create a syntax tree, i.e., a tree data structure representing the program
- A computation is then defined as a function over these trees
- The translation may be divided into phases
- The different phases may share the same physical tree
  - Advantage: Efficient, only one tree definition
  - Disadvantage: Data dependencies are not explicit, hence harder to understand, debug, optimize, parallelize, …

Syntax Trees Carry Information



menhir doesn’t build anything yet

```
{ open Parser
  let get = Lexing.lexeme
}
(* Helpers *)
 let tab   = '\009'
 let cr    = '\013'
 let lf    = '\010'
 let eol   = cr | lf | cr lf

rule token = parse
 | eol             { token lexbuf }
 | (' ' | tab)     { token lexbuf }
 | eof             { EOF }
 | '+'             { PLUS }
 | '-'             { MINUS }
 | '*'             { STAR }
 | '/'             { SLASH }
 | '('             { LPAR }
 | ')'             { RPAR }
 | ('x'|'y'|'z')  { ID(get lexbuf) }
```

```
%{ %}

%token EOF
%token PLUS MINUS STAR SLASH
%token LPAR RPAR
%token <string>ID

%start <unit> start  /* entry point */
%%
start : expr EOF                 { };

expr   : expr PLUS term          { }
       | expr MINUS term         { }
       | term                    { };

term : term STAR factor          { }
     | term SLASH factor         { }
     | factor                    { };

factor : ID                      { }
       | LPAR expr RPAR          { };
```

First: a datatype for parse trees

cst.ml:

```
type expr =
  | Plusexpr of expr * term
  | Minusexpr of expr * term
  | Termexpr of term

and term =
  | Multterm of term * factor
  | Divterm of term * factor
  | Factorterm of factor

and factor =
  | IDfactor of string
  | Parenfactor of expr
```

```
%{ %}

%token EOF
%token PLUS MINUS STAR SLASH
%token LPAR RPAR
%token <string>ID

%start <unit> start
%%
start : expr EOF    { };

expr
 : expr PLUS term   { }
 | expr MINUS term  { }
 | term             { };

term
 : term STAR factor  { }
 | term SLASH factor { }
 | factor            { };

factor
 : ID                { }
 | LPAR expr RPAR    { };
```

Second: building parse trees

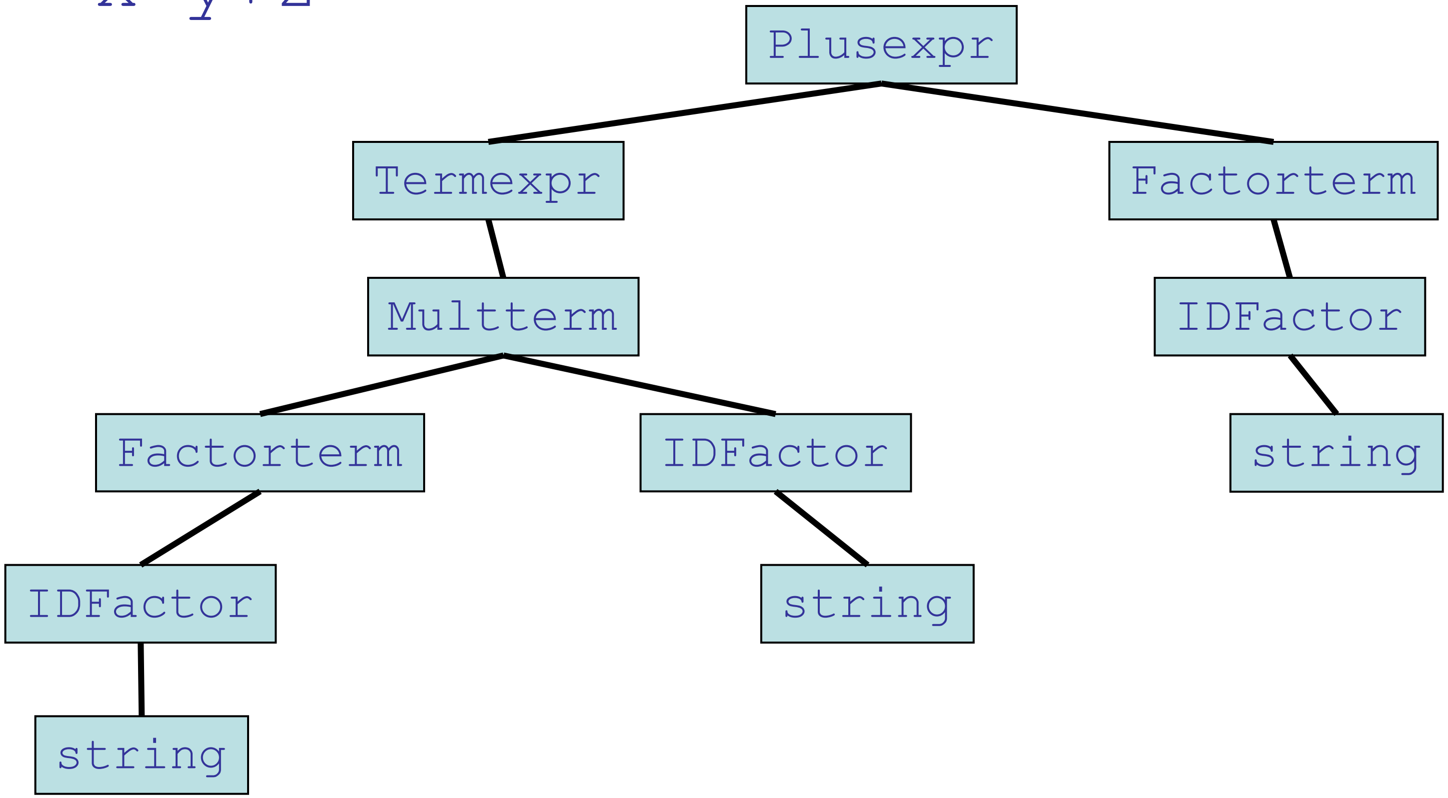
cst.ml:

```
type expr =
   | Plusexpr of expr * term
   | Minusexpr of expr * term
   | Termexpr of term

and term =
   | Multterm of term * factor
   | Divterm of term * factor
   | Factorterm of factor

and factor =
   | IDfactor of string
   | Parenfactor of expr
```

```
%{ %}

%token EOF
%token PLUS MINUS STAR SLASH
%token LPAR RPAR
%token <string>ID

%start <Cst.expr> start
%%
start : expr EOF     { $1 };

expr
 : expr PLUS term    { Cst.Plusexpr ($1,$3) }
 | expr MINUS term   { Cst.Minusexpr ($1,$3) }
 | term              { Cst.Termexpr $1 };

term
 : term STAR factor  { Cst.Multterm($1,$3) }
 | term SLASH factor { Cst.Divterm($1,$3) }
 | factor            { Cst.Factorterm $1 };

factor
 : ID                { Cst.IDfactor $1 }
 | LPAR expr RPAR    { Cst.Parenfactor $2 };
```

Parse trees use all of these nodes

- x*y+z



In OCaml:

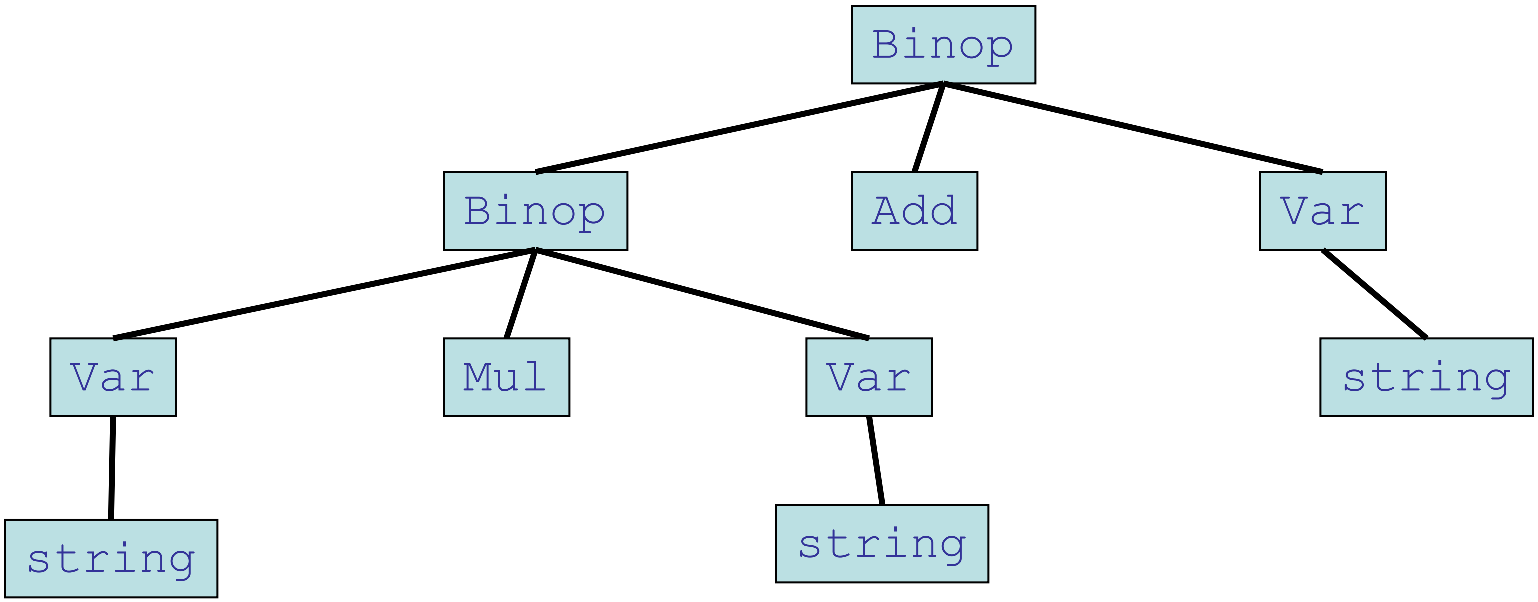
```
Plusexpr(Termexpr(Multterm(Factorterm (Idfactor
  "x"), IDfactor "y")), Factorterm (IDfactor "z"))
```

Irrelevant Details

- LR(1) parse trees have irrelevant details
- There is no semantic distinction between:
  - `expr`
  - `term`
  - `factor`
- The extra structure complicates traversals...

Abstract Syntax Trees

- An AST records only semantically relevant information:



ASTs in menhir

- The AST could be built by traversing the parse tree and building a new one
- We would quickly get tired of this...

- menhir leaves the design space open
- We could use another datatype for the ASTs
- Productions define an inductive mapping
- An AST grammar is just another recursive datatype

Building ASTs (1/3)

```
{ open Parser
  let get = Lexing.lexeme
}
(* Helpers *)
 let tab   = '\009'
 let cr    = '\013'
 let lf    = '\010'
 let eol   = cr | lf | cr lf

rule token = parse
 | eol              { token lexbuf }
 | (' ' | tab)      { token lexbuf }
 | eof              { EOF }
 | '+'              { PLUS }
 | '-'              { MINUS }
 | '*'              { STAR }
 | '/'              { SLASH }
 | '('              { LPAR }
 | ')'              { RPAR }
 | ('x'|'y'|'z')  { ID(get lexbuf) }
```

Building ASTs (2/3)

ast.ml:

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

type exp =
   | Var of string
   | Binop of exp * binop * exp
```

```
%{ %}

%token EOF
%token PLUS MINUS STAR SLASH
%token LPAR RPAR
%token <string>ID

%start <unit> start
%%
start : expr EOF    { };

expr
 : expr PLUS term  { }
 | expr MINUS term { }
 | term            { };

term
 : term STAR factor  { }
 | term SLASH factor { }
 | factor            { };

factor
 : ID                { }
 | LPAR expr RPAR    { };
```

Building ASTs (3/3)

ast.ml:

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

type exp =
  | Var of string
  | Binop of exp * binop * exp
```

```
%{ %}

%token EOF
%token PLUS MINUS STAR SLASH
%token LPAR RPAR
%token <string>ID

%start <Ast.exp> start
%%
start : expr EOF   { $1 };

expr
 : expr PLUS term  { Ast.Binop ($1,Ast.Add,$3) }
 | expr MINUS term { Ast.Binop ($1,Ast.Sub,$3) }
 | term            { $1 };

term
 : term STAR factor  { Ast.Binop($1,Ast.Mul,$3) }
 | term SLASH factor { Ast.Binop($1,Ast.Div,$3) }
 | factor            { $1 };

factor
 : ID                { Ast.Var $1 }
 | LPAR expr RPAR    { $2 };
```

Tree traversals

- Many applications need to traverse the AST
- We can implement traversals as structurally recursive functions over the AST datatype

Pretty Printing

pprint.ml:

```
(*  print_op : Ast.binop -> unit  *)
let print_op op = match op with
  | Ast.Add ->
  | Ast.Sub ->
  | Ast.Mul ->
  | Ast.Div ->

(*  print_exp : Ast.exp -> unit  *)
let rec print_exp exp = match exp with
  | Ast.Var v ->

  | Ast.Binop (exp0, op, exp1) ->
```

Pretty Printing

pprint.ml:

```
(*  print_op : Ast.binop -> unit  *)
let print_op op = match op with
  | Ast.Add -> print_string "+"
  | Ast.Sub -> print_string "-"
  | Ast.Mul -> print_string "*"
  | Ast.Div -> print_string "/"

(*  print_exp : Ast.exp -> unit  *)
let rec print_exp exp = match exp with
  | Ast.Var v ->

  | Ast.Binop (exp0, op, exp1) ->
```

Pretty Printing

pprint.ml:

```
(*  print_op : Ast.binop -> unit  *)
let print_op op = match op with
  | Ast.Add -> print_string "+"
  | Ast.Sub -> print_string "-"
  | Ast.Mul -> print_string "*"
  | Ast.Div -> print_string "/"

(*  print_exp : Ast.exp -> unit  *)
let rec print_exp exp = match exp with
  | Ast.Var v ->
    print_string v
  | Ast.Binop (exp0, op, exp1) ->
```

Pretty Printing

pprint.ml:

```
(*  print_op : Ast.binop -> unit  *)
let print_op op = match op with
  | Ast.Add -> print_string "+"
  | Ast.Sub -> print_string "-"
  | Ast.Mul -> print_string "*"
  | Ast.Div -> print_string "/"

(*  print_exp : Ast.exp -> unit  *)
let rec print_exp exp = match exp with
  | Ast.Var v ->
    print_string v
  | Ast.Binop (exp0, op, exp1) ->
    begin
      print_string "(";
      print_exp exp0;
      print_op op;
      print_exp exp1;
      print_string ")";
    end
```

Evaluating the Expressions

- Prettyprinting was carried out for its sideeffect (`unit`)
- Evaluation on the other hand needs to return a value
- Again we can implement it as a structurally recursive function over the AST datatype

Evaluation (1/2)

eval.ml:

```
exception Unknownvar of string

(*  lookup : string -> (int * int * int) -> int  *)
let lookup var env =
  let (xval, yval, zval) = env in
  match var with
     | "x" -> xval
     | "y" -> yval
     | "z" -> zval
     | _   -> raise (Unknownvar var)
```

Evaluation (2/2)

eval.ml:

```
(*  eval_op : int -> Ast.binop -> int -> int  *)
let eval_op v0 op v1 = match op with
  | Ast.Add -> v0 + v1
  | Ast.Sub -> v0 - v1
  | Ast.Mul -> v0 * v1
  | Ast.Div -> v0 / v1

(*  eval_exp : Ast.exp -> env -> int  *)
let rec eval_exp exp env = match exp with
  | Ast.Var var ->
    lookup var env
  | Ast.Binop (exp0,op,exp1) ->
    let v0 = eval_exp exp0 env in
    let v1 = eval_exp exp1 env in
    eval_op v0 op v1
```

Putting it all together

```
let lexbuf = Lexing.from_channel stdin in
try
  let xval = int_of_string (Sys.argv.(1)) in
  let yval = int_of_string (Sys.argv.(2)) in
  let zval = int_of_string (Sys.argv.(3)) in
  let env  = (xval,yval,zval) in
  let exp  = Parser.start Lexer.token lexbuf in  (* parse input *)
  let ()   = Pprint.print_exp exp in             (* pretty print *)
  let ()   = print_newline () in
  print_int (Eval.eval_exp exp env)              (* evaluate *)
with
  | Invalid_argument _ -> print_endline ("Usage: " ^ Sys.argv.(0) ^ " 3 4 5")
  | Failure msg        -> print_endline ("Failure in " ^ msg)
  | Parser.Error       -> print_endline "Parse error"
  | End_of_file        -> print_endline
                            "Parse error: unexpected end of string"
```

Manipulating ASTs

- Desugaring:
  locally translate constructs into simpler forms
- Weeding:
  reject unwanted ASTs
- Transforming:
  rewrite sub-ASTs

An HTML Subset

HTML → *word**
| **** *HTML* ****
| **** *HTML* ****
| **<i>** *HTML* **</i>**
| **** *HTML* ****

HTML in menhir (1/2)

```
%{ %}

%token EOF
%token STARTA HREF EQ QUOTE GT ENDA
%token STARTB STARTI STARTEM
%token ENDB ENDI ENDEM
%token <string>WORD

%start <unit> main
%%

main : html EOF           { };

html : WORD*                                          { }
     | STARTA HREF EQ QUOTE WORD QUOTE GT html ENDA   { }
     | STARTB html ENDB                               { }
     | STARTI html ENDI                               { }
     | STARTEM html ENDEM                             { };
```

HTML in ocamllex (2/2)

```
{
  open Parser
  let get = Lexing.lexeme
}

(* Helpers *)
let tab   = '\009'
let cr    = '\013'
let lf    = '\010'
let eol   = cr | lf | cr lf
let char  =
 ['a'-'z'] | ['A'-'Z'] | ['0'-'9']
```

```
rule token = parse
  | eol          { token lexbuf }
  | (' ' | tab)  { token lexbuf }
  | eof          { EOF }
  | "<a"         { STARTA }
  | "href"       { HREF }
  | '='          { EQ }
  | '"'          { QUOTE }
  | '>'          { GT }
  | "</a>"       { ENDA }
  | "<b>"        { STARTB }
  | "<i>"        { STARTI }
  | "<em>"       { STARTEM }
  | "</b>"       { ENDB }
  | "</i>"       { ENDI }
  | "</em>"      { ENDEM }
  | char char*   { WORD (get lexbuf) }
```

Desugaring (1/2)

- View `<em>` as syntactic sugar for `<i>`
- The target is a recursive datatype:

ast.ml:

```
type html =
  | Words of string list
  | A of string * html
  | B of html
  | I of html
```

Desugaring (2/2)

- View `<em>` as syntactic sugar for `<i>`
- Just perform the translation during AST building:

parser.mly (high-lights):

```
%start <Ast.html> main
%%

main : html EOF         { $1 };

html
   :  WORD*                                           { Ast.Words $1 }
   |  STARTA HREF EQ QUOTE WORD QUOTE GT html ENDA    { Ast.A ($5,$8) }
   |  STARTB html ENDB                                { Ast.B $2 }
   |  STARTI html ENDI                                { Ast.I $2 }
   |  STARTEM html ENDEM                              { Ast.I $2 };
```

Weeding

- Don't allow nested anchors
- One solution is to rewrite the grammar:

HTML → *word**
  | `<a href="`*word*`">` *HTMLNoAnchor* `</a>`
  | `<b>` *HTML* `</b>`
  | `<i>` *HTML* `</i>`
  | `<em>` *HTML* `</em>`

HTMLNoAnchor → *word**
  | `<b>` *HTMLNoAnchor* `</b>`
  | `<i>` *HTMLNoAnchor* `</i>`
  | `<em>` *HTMLNoAnchor* `</em>`

Combinatorial Explosion

- We just doubled the size of the grammar
- Enforcing 10 constraints like this makes the grammar 2^{10} = 1024 times larger
- And impossible to maintain...

A Weeding Phase

```
(*  weed : Ast.html -> int -> unit  *)
let rec weed html aheight = match html with
  | Ast.Words ws ->

  | Ast.A(link,body) ->







  | Ast.B body ->

  | Ast.I body ->



(*  weed_html : Ast.html -> unit  *)
let weed_html html =
```

A Weeding Phase

```
(*  weed : Ast.html -> int -> unit  *)
let rec weed html aheight = match html with
  | Ast.Words ws ->
     ()
  | Ast.A(link,body) ->







  | Ast.B body ->

  | Ast.I body ->



(*  weed_html : Ast.html -> unit  *)
let weed_html html =
```

A Weeding Phase

```
(*  weed : Ast.html -> int -> unit  *)
let rec weed html aheight = match html with
  | Ast.Words ws ->
     ()
  | Ast.A(link,body) ->
    if aheight > 0
    then
      raise (Failure "Nested anchors")
    else
      weed body (aheight+1)
  | Ast.B body ->

  | Ast.I body ->



(*  weed_html : Ast.html -> unit  *)
let weed_html html =
```

A Weeding Phase

```
(*  weed : Ast.html -> int -> unit  *)
let rec weed html aheight = match html with
  | Ast.Words ws ->
     ()
  | Ast.A(link,body) ->
    if aheight > 0
    then
      raise (Failure "Nested anchors")
    else
      weed body (aheight+1)
  | Ast.B body ->
    weed body aheight
  | Ast.I body ->
    weed body aheight

(*  weed_html : Ast.html -> unit  *)
let weed_html html =
```

A Weeding Phase

```
(*  weed : Ast.html -> int -> unit  *)
let rec weed html aheight = match html with
  | Ast.Words ws ->
    ()
  | Ast.A(link,body) ->
    if aheight > 0
    then
      raise (Failure "Nested anchors")
    else
      weed body (aheight+1)
  | Ast.B body ->
    weed body aheight
  | Ast.I body ->
    weed body aheight

(*  weed_html : Ast.html -> unit  *)
let weed_html html = weed html 0
```

Transformation

- Eliminate nested tags
- Again, one solution is to rewrite the grammar:

HTML → *word**
 | **** *HTML* ****
 | **** *HTMLInsideB* ****
 | **<i>** *HTML* **</i>**
 | **** *HTML* ****

HTMLInsideB → *word**
 | **** *HTMLInsideB* ****
 | **** *HTMLInsideB* ****
 | **<i>** *HTMLInsideB* **</i>**
 | **** *HTMLInsideB* ****

ignore this in the AST

Combinatorial Explosion

- This also doubles the size of the grammar
- Detecting 7 conditions like this makes the grammar 2^7 = 128 times larger
- Combined with the earlier 10 constraints, the grammar is now 131,072 times larger, with nonterminals such as:

  HTMLInsideBNotInsideINoAnchor...

A Transformation Phase

```
(*  transform : Ast.html -> int -> Ast.html  *)
let rec transform html bdepth = match html with
  | Ast.Words ws ->

  | Ast.A(link,body) ->



  | Ast.B body ->





  | Ast.I body ->




(*  transform_html : Ast.html -> Ast.html  *)
let transform_html html =
```

A Transformation Phase

```
(*  transform : Ast.html -> int -> Ast.html  *)
let rec transform html bdepth = match html with
  | Ast.Words ws ->
    html
  | Ast.A(link,body) ->
    let body' = transform body bdepth in
    Ast.A(link,body')
  | Ast.B body ->
    let body' = transform body (bdepth+1) in
    if bdepth>0
    then body'           (* drop nested tag *)
    else Ast.B body'
  | Ast.I body ->
    let body' = transform body bdepth in
    Ast.I body'

(*  transform_html : Ast.html -> Ast.html  *)
let transform_html html = transform html 0
```

An Outline Phase (1/2)

```
(*  indent_string : int -> string  *)
let rec indent_string i = match i with
  | 0 -> ""
  | _ -> " " ^ indent_string (i-1)

(*  outline : Ast.html -> int -> unit  *)
let rec outline html indent = match html with
  | Ast.Words ws ->
    begin
      print_string (indent_string indent);
      List.iter (fun w -> print_string (w ^ " ")) ws;
      print_newline()
    end
  | Ast.A(link,body) ->
    begin
      print_endline (indent_string indent ^ "a " ^ link);
      outline body (indent+1)
    end
```

An Outline Phase (2/2)

```
  | Ast.B body ->
    begin
      print_endline (indent_string indent ^ "b");
      outline body (indent+1)
    end
  | Ast.I body ->
    begin
      print_endline (indent_string indent ^ "i");
      outline body (indent+1)
    end

(*  outline_html : Ast.html -> unit  *)
let outline_html html = outline html 0
```

The Main Application

```
let lexbuf = Lexing.from_channel stdin in
try
  let html  = Parser.html Lexer.token lexbuf in (* parse input *)
  let ()    = Weeding.weed_html html in          (* check nested anchors  *)
  let html' = Transform.transform_html html in  (* eliminate nested b tags *)
              Outline.outline_html html’         (* print an outline *)
with
  | Failure msg  -> print_endline ("Failure --- " ^ msg)
  | Parser.Error -> print_endline "Parse error"
  | End_of_file  -> print_endline "Parse error: unexpected end of string"
```

Beyond ocamllex and menhir

- Most languages come with a lex and yacc variant
- Other parser generators deal with ASTs differently
- SableCC (for Java), for example, will generate classes representing the tree as well as visitor patterns for traversals
- Advantage: no need to write traversal code repeatedly
- Disadvantage: hard to switch AST, one is stuck with the generated traversal, not allowed to declare fields and methods on the nodes
- The previous years we used *Aspect oriented programming* to inject fields and methods

Example: SableCC (1/2)

```
Helpers
  tab   = 9;
  cr    = 13;
  lf    = 10;

Tokens
  eol   = cr | lf | cr lf;
  blank = ' ' | tab;
  star  = '*';
  slash = '/';
  plus  = '+';
  minus = '-';
  l_par = '(';
  r_par = ')';
  id = 'x' | 'y' | 'z';

Ignored Tokens
  blank,eol;
```

Example: SableCC (2/2)

```
Productions
  start {-> exp} =
    {plus} start plus term
           {-> New exp.binop(start.exp, New binop.add(), term.exp)} |
    {minus} start minus term
           {-> New exp.binop(start.exp, New binop.sub(), term.exp)} |
    {term} term {-> term.exp} ;
  term  {-> exp} =
    {mult} term star factor
           {-> New exp.binop(term.exp, New binop.mul(), factor.exp)} |
    {div}  term slash factor
           {-> New exp.binop(term.exp, New binop.div(), factor.exp)} |
    {factor} factor {-> factor.exp};
  factor {-> exp} =
    {id} id {-> New exp.var(id)} |
    {paren} l_par start r_par {-> start.exp};

Abstract Syntax Tree
  exp = {binop} [l]:exp binop [r]:exp | {var} id;
  binop = {add} | {sub} | {mul} | {div};
```

Example: ANTLR

Using the generic tree:

```
expr
   : primary_expr PLUS primary_expr
   ;

primary_expr
   : IDENT
   ;
```

Using a custom AST and semantic actions:

```
expr returns [pNode expr_tree]
  { pNode e1 = NULL;
    pNode e2 = NULL;
  }
  : e1 = primary_expr PLUS e2 = primary_expr
    { expr_tree = factory.build_binary( PLUS, e1, e2 ); }
  ;

primary_expr returns [pNode id_node]
  : IDENT { id_node = factory.make_node( n_id ); }
  ;
```