Lecture 4 — Abstract syntax

- **In this class, you will see some examples of abstract syntax as expressed in OCaml, and write functions on ASTs. Writing recursive functions on ASTs is one of the key skills needed to write compilers.**

  Specifically, we will work with abstract syntax for:

  - **A simple expression language**
  - **A simple expression language with a `let` construct**
  - **MiniJava (a subset of Java)**

Review from Tuesday's class

- **Here is an abstract syntax for simple arithmetic expressions as an OCaml data type:**

```
type expr = Int of int | Plus of expr*expr
          | Times of expr*expr | Negate of expr
```

- **Show the abstract syntax tree for expression** `4+-(7*-8+4)`**:**

- **Give the OCaml expression of type** `expr` **for that tree:**

Exercises using `expr`

- **Write the function** `eval: expr` $\rightarrow$ `int`**, which evaluates its argument, e.g.** `eval (Times(Negate(Int 5), Int 6))` **= -30.**

```
let rec eval e = match e with
  Int i ->

| Plus(e1, e2) ->



| Times(e1, e2) ->



| Negate e ->
```

Exercises using `expr` (cont.)

- **For a little more practice, write `eval` for this slightly different definition of type `expr`:**

```
type expr = Int of int | Binop of bop * expr * expr
                       | Unop of uop * expr
 and bop = Plus | Times
 and uop = Negate

let rec eval e = match e with
  Int i ->

| Binop(op, e1, e2) ->




| Unop(op, e) ->
```

Expressions w/ let

- **If we add let-bound names to arithmetic expressions, we can write expressions like** `let x=3 in let y=x*x in x+y`**. Here's an abstract syntax for this language:**

```
type expr =  Int of int   |  Binop of bop * expr * expr
          |  Var of string  |  Let of def * expr
 and def = string * expr
 and bop = Plus  |  Times
```

- **Write the** `expr` **corresponding to** `let x=3 in let y=x*x in x+y`**.**

Expressions w/ let (cont.)

- **Evaluating expressions with let is harder because expressions can contain variables. Let's start by evaluating expressions that can contain variables but not `let`. The values of the variables are given by a list of type `(string * int) list`, called `st` for "store," which is an argument to `eval`. We need to write a function to look up values in this list:**

```
let rec lookup x st = match st with
```

Expressions w/ let (cont.)

- **Write the** `eval` **function for expressions with variables but not** `let`**.**

```
   type expr =  Int of int  |  Binop of bop * expr * expr
             |  Var of string  |  Let of def * expr
    and def = string * expr
    and bop = Plus  |  Times

let rec eval e st = match e with
   Int i ->

 | Var(s) ->


 | Binop(b, e1, e2) ->
```

Expressions w/ let (cont.)

- **To evaluate lets, we need a way to add variables to the store. But that's easy: to give `x` the value `n`, just cons `(x,n)` to the front of the store.**

- **Write `eval` including `let`. The other clauses are unchanged:**

```
let rec eval e store = match e with
   Int i ->
 | Var(s) ->
 | Binop(b, e1, e2) ->


 | Let((x,e1), e2) ->
```

Abstract syntax of MiniJava

- In the first half of the semester, we will build a compiler for a Java-like language called MiniJava. Over the new few weeks, we will build the "front end" of that compiler, whose primary purpose is to transform source files into abstract syntax trees.

- In MP 2, you will write some functions on the abstract syntax for MiniJava. That abstract syntax is given here; to help you understand what it means, we have shown for some cases the correspondence between abstract and concrete syntax in a box after each constructor declaration.

```
type program = Program of (class_decl list)
```

Program [C_1; C_2; ...] $\Leftrightarrow$ C_1 C_2 ...

```
and class_decl = Class of id * id
                       * (var_decl list) * (method_decl list)
```

Class (c, s, vs, ms) $\Leftrightarrow$ class c extends s { vs ms }

```
and method_decl = Method of exp_type * id * ((exp_type * id) list)
        * (var_decl list) * (statement list) * exp
```

Method (t, m, $args$, $vars$, ss, e) $\Leftrightarrow$ t m ($args$) { $vars$ ss return e; }

```
and var_decl = Var of var_kind * exp_type * id
```

Var (Static, t, x) $\Leftrightarrow$ static t x

Var (NonStatic, t, x) $\Leftrightarrow$ t x

```
and var_kind = Static | NonStatic
```

```
and statement = Block of (statement list)
```

Block [s_1, s_2, ...] $\Leftrightarrow$ s_1 s_2 ...

```
| If of exp * statement * statement
```

If (e, s_1, s_2) $\Leftrightarrow$ if (e) s_1 else s_2

```
| While of exp * statement
```

While (e, s) $\Leftrightarrow$ while (e) s

```
| Println of exp
```

Println (e) $\Leftrightarrow$ System.out.println(e)

```
| Assignment of id * exp
```

Assignment (x, e) $\Leftrightarrow$ x = e;

```
| ArrayAssignment of id * exp * exp
```

ArrayAssignment (x, e_1, e_2) $\Leftrightarrow$ x[e_1] = e_2;

```
| Break
| Continue
```

```
and exp = Operation of exp * binary_operation * exp
    | Subscript of exp * exp
```

> Subscript $(e_1, e_2) \Leftrightarrow e_1[e_2]$

```
    | Integer of int
    | Id of id
    | Length of exp
    | MethodCall of exp * id * (exp list)
```

> MethodCall $(e, f, args) \Leftrightarrow e.f(args)$

```
    | FieldRef of exp * id
    | True
    | False
    | This
    | NewId of id
```

> NewId $(C) \Leftrightarrow$ new $C()$

```
    | NewArray of exp_type * exp
```

$\boxed{\texttt{NewArray } (t\texttt{, } e) \Leftrightarrow \texttt{new } t\texttt{[}e\texttt{]}}$

```
    | Not of exp
    | Null
    | String of string
    | Float of float

and binary_operation = And | Or
    | LessThan | GreaterThan | LessThanEq | GreaterThanEq | Equal
    | Plus | Minus | Multiplication | Division

and exp_type = ArrayType of exp_type
    | BoolType
    | IntType
    | ObjectType of id
    | StringType
    | FloatType

and id = string;
```

Ex: pretty-print expressions

- **Write part of the definition of** pp : exp $\rightarrow$ string**, that produces a parsable string version of its argument. (**pp **stands for "pretty-print".) We repeat the corresponding parts of the abstract syntax for reference.** pp_bop **is an auxiliary function you can use.**

```
and exp = Operation of exp * binary_operation * exp
    | Subscript of exp * exp | Integer of int | Id of id | ...

let pp_bop binop = match binop with And -> "&&" | LessThan -> "<" | ...

let rec pp e = match e with
   Operation(e1, binop, e2) ->



 | Subscript(e1, e2) ->


 | Integer i ->


 | Id id ->
```

Wrap-up

- **Today we discussed:**
  - **Defining ASTs**
  - **Writing functions on ASTs by pattern-matching and tree traversal.**
- **We discussed it because:**
  - **ASTs are the central data structure in a compiler.**
- **In the next two classes, we will:**
  - **Talk about lexing**
  - **Next 3 weeks: goal to convert programs to ASTs (while learning OCaml)**
- **What to do now:**
  - *MP2* — **practice with abstract syntax of MiniJava**
  - **Important: For next class, review DFAs and reg. expr.'s from CS 373**