

Lecture 4 — Abstract syntax

- Abstract syntax examples for various languages:
 - A simple expression language
 - OCaml
 - MiniJava (a subset of Java)

Ex: Abstract syntax of simple expressions

- *Abstract syntax* is a tree representation of the basic syntactic structure of a program.
- For a simple example, consider the language of ordinary arithmetic expressions, with numbers, unary and binary operators, and parentheses.
- For example, expression $3 + 4 * 5 + -7$ would be represented by this tree:

Abstract syntax of simple expressions (cont.)

- 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
```

- The tree given above would be written as an OCaml expression of type `expr` like this:

Exercises using `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` (cont.)

- Write the function `countPluses: expr → int`, which counts the number of `Plus` operations in an `expr`:

```
let rec countPluses e = match e with  
  Int i ->  
  
  | Plus(e1, e2) ->  
  
  | Times(e1, e2) ->  
  
  | Negate e ->
```

Exercises using `expr` (cont.)

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

Exercises using `expr` (cont.)

- For a little more practice, consider 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
```

Define `eval` for this definition of `expr`:

Abstract syntax of OCaml

- Here's a (partial) abstract syntax for OCaml:

```
type ocamlexpr = Int of int   | Binop of bop * ocamlexpr * ocamlexpr
                | Var of string | App of ocamlexpr * ocamlexpr
                | Let of def * ocamlexpr | Letrec of def * ocamlexpr
and def = string * string list * ocamlexpr
and bop = Plus | Times
```

- Write the ocamlexpr corresponding to

```
let f a = let x = 0 in a+x in f 7
```

Ex: Abstract syntax of OCaml

`occursin: string  $\rightarrow$  ocamlexpr  $\rightarrow$  bool` checks if a variable or function name is used in an expression (in its scope), e.g.

`a` occurs in `let x = a in ...`

`a` occurs in `let x = 0 in a+x`

`a` does not occur in `let a = 0 in a`

`f` occurs in `let g x = f (x+1) in g 0`

`f` does not occur in `let rec f x = f (x+1) in f 0`

`let rec occursin v e = match e with`

`Int i ->`

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

`| Var(s) ->`

`| App(e1, e2) ->`

`| Let(d, e) ->`

`| Letrec(d, e) ->`

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:

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

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

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

```
* (var_decl list)
* (statement list)
* exp
```

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

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

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

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

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

```
| Println of exp
```

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

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

```
| Break
```

```
| Continue
```

```
| Switch of exp
```

```
    * ((int * (statement list)) list)    (* cases *)
```

```
    * (statement list)    (* default *)
```

```
and exp = Operation of exp * binary_operation * exp
  | Array of exp * exp
  | Length of exp
  | MethodCall of exp * id * (exp list)
  | Integer of int
  | True
  | False
  | Id of id
  | This
  | NewArray of exp_type * exp
  | NewId of id
  | Not of exp
  | Null
  | String of string
  | Float of float

and binary_operation = And
  | Or
  | LessThan
  | 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 `pp : exp → string`, that produces a printed version of its argument, such that, if it were parsed, it would produce the same abstract syntax tree. (pp stands for “pretty-print”.) Use parentheses freely (even though some will be unnecessary).

```
let rec pp_binop binop = match binop with  
  And -> "&&" | LessThan -> "<" | ...
```

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

```
  | Array(e1, e2) ->
```

```
  |
```