F28PL1 Programming Languages

Lecture 11: Standard ML 1

Imperative languages

digital computers are concrete realisations of *von Neumann machines*

stored program

memory

- associations between addresses and values

instructions change memory

i.e. change address/value association

order of evaluation is fundamental

changing instruction order changes program meaning

Imperative languages

imperative languages

e.g. Java, C

abstractions from von Neumann machines

variables

associations between names and values

statements/commands change variables

i.e. changing name/value associations

order of evaluation is fundamental

changing statement order changes program meaning

Imperative languages

program parts communicate by modifying/accessing common variables

order of modification/access determines final result

e.g. swap `x and y`

```
int x=3,y=2;{(x,3),(y,2)}

t=x;    {(x,3),(y,2),(t,3)}

x=y;    {(x,2),(y,2),(t,3)}

y=t;    {(x,2),(y,3),(t,3)}
```

Imperative languages

exchange first two statements

```
int x=3,y=2;{(x,3),(y,2)}
x=y;        {(x,2),(y,2)}
t=x;        {(x,2),(y,2),(t,2)}
y=t;        {(x,2),(y,2),(t,2)}
```

```
exchange second two statements
int x=3,y=2;{(x,3),(y,2)}
t=x;        {(x,3),(y,2),(t,3)}
y=t;        {(x,3),(y,3),(t,3)}
x=y;        {(x,3),(y,3),(t,3)}
```

Side effects

change to non-local variables

typically function in expression changes shared variables

e.g. decrement variable and test for 0

```
int inc(int * x)

{  return ++(*x);  }
```

`inc  changes *x as a side effect`

Side effects

```
int inc(int * x)
{  return ++(*x);  }
```

order of use of inc in expresisons with x crucial

```
int i = 0;
printf("%d\n",inc(&i)+i); ==> 2
printf("%d\n",i+inc(&i)); ==> 1
```

Declarative languages

declarative languages

"describe what is to be done, not how to do it"

logic languages

e.g. Prolog

abstractions from predicate calculus

functional languages

e.g. Standard ML, Haskell

abstractions from lambda calculus/recursive function theory

Declarative languages

variables

associations between names and values

in pure declarative languages, *variables cannot be modified*

i.e. no assignment

program parts cannot interact by modifying shared variables

no side effects

Declarative languages

Church-Rosser property

evaluation order independence

evaluate program in any order

if program terminates then result always the same

makes proof of declarative program properties easier than for imperative programs

do not need to build idea of time/order into axioms and rules for declarative languages

declarative languages good basis for parallelism

easier to identify potential loci of parallelism

Declarative languages

declarative languages are more abstract than imperative languages relative to von Neumann machines

harder to generate code for declarative languages

declarative language implementations may be less efficient than for imperative languages

assignment absence requires data structure updating by *copying with changes* instead of direct structure modification

less efficient with large structures

Declarative languages

we will study:

functional language

- Standard ML

declarative language

- Prolog

strong similarities

case definitions, pattern matching, lists, recursion

but fundamental differences:

types; evaluation order; variable binding

Standard ML

modern functional language

originally *meta-language* for LCF Theorem Prover

i.e. language in which theorems expressed

now widely used for:

teaching

research

high-integrity tools/applications

basis of Microsoft’s F# - part of .NET

SML resources

NJSML for Linux from:

http://www.smlnj.org/

obtain other free SML for Windows and Linux via:

http://www.smlnj.org/links.html

G. Michaelson, Elementary Standard ML, UCL Press, 1995

free Postscript from: http://www.macs.hw.ac.uk/~greg/books

NB discusses older SML version

Standard ML

both interpreters and compilers available

SML is really an imperative language with a pure functional subset

useful for evolutionary prototyping:

develop prototype using pure functional SML

once happy with prototype, introduce imperative constructs

Running SML

SML program consists of:

sequence of function and type definitions

expressions using functions and types for evaluation

will use New Jersey SML interpreter

to run SML on Linux:

```
linux00% sml
Standard ML of New Jersey v110.72 [built: Fri May  6 13:06:25 2011]
-
```

`-` is SML prompt

Running SML

enter an expression or command or definition

followed by a `;`

`SML system`

`evaluates expressions`

`displays values and types of expression results`

```
- expression;
> value : type
```

`>  precedes system output`

Running SML

NB some SML systems display:

```
- expression;
```

```
> val it = value : type
```

`it - value of most recently evaluated expression`

`we will not use this form`

Running SML

system commands are expressions based on function calls

SML system remembers definitions

displays types of definitions

system will automatically replace old definitions with new ones

system does not support interactive editing

Running SML

suggested program development cycle:

prepare program in file in one window

run SML system in another window

while *program not perfect* do

- *load program file into SML system*
- if *errors* then
  - *change program in file & save file*
- else
  - *test program*
  - if *errors* then
    - *change program in file & save file*

Running SML

to load file:

`- use “`*`file name`*`”;`

`file name`` is any valid file path enclosed in string quotes “...”`

`file name convention`

`SML file names end with .sml`

`to leave SML`

`- ^D`

Standard ML summary

strong types

i.e. can’t change type associated with variable

static typing

i.e. types checked at compile time

parametric polymorphism

i.e. type variables e.g. Java generics

strict parameter passing

i.e. parameters evaluated before function entry

left to right evaluation

Integers

`int` - integer *type constructor*

positive and negative integer values

```
- 42;
> 42 : int
- ~42;
> ~42 : int
```

NB ~ - integer unary minus/negation function

Integers

binary infix operators

```
+      - add two integers to give integer
-    - subtract 2nd integer from 1st to give integer
*      - multiply two integers to give integer
div  - divide 1st integer by 2nd to give integer
mod  - integer remainder after dividing 1st integer by
 2nd
```

```
 group operations with (...)
```

Integers

precedence:

```
(...) before

~ before

* and div and mod before

+ and -

left to right evaluation order
```

Integers

can use SML interpreter like a desk calculator

```
- 3*14;

> 42 : int

- 6*21 div 3;

> 42 : int

- (9+12) div 3 *(12-6);

> 42 : int

- 3*61 mod 47;

> 42 : int
```

Reals

```
real  - real type constructor

positive/negative floating point and decimal
fractions

- 1.234;

> 1.234 : real

- 5.67E4

> 56700.0 : real

-  891.0E~3;

> 0.891 : real

- ~1.01112;

> ~1.01112 : real
```

Reals

numberEinteger == _number_ * 10_integer_

NB ~ - real unary minus/negation function

binary infix operators

+   - add two reals to give real

- - subtract 2nd reals from 1st to give real

* - multiply two reals to give real

/   - divide 1st real by 2nd to give real

group operations with (...)

Reals

precedence:

```
(...) before

~ before

* and / before

+ and -
```

Reals

left to right evaluation order

```
- 1.4*3E1;

> 42.0 : real

-  4.79*10.0-5.9;

> 42.0 : real
```

Overloaded operators and mixed mode arithmetic

```
~, +, -, and *  are overloaded for integers and real

+, -, and * may be applied to two integers only or
two reals only

cannot mix integer and real arithmetic directly

must convert

prefix unary functions:

real    - converts integer to real

floor   - converts real to integer

        - rounds down
```

Function calls

functions are prefix:

function argument_expression

strict evaluation

i.e. evaluate *argument_expression* to get actual parameter value before entering function

```
- floor 6.789;
> 6 : int
- real 456;
> 456.0 : real
```

Function calls

precedence:

```
(...) before

function call before

operators


- floor 12.3+4;

> 16 : int

- 1.2*real 3+4.5;

> 8.1 real
```

Function calls

only bracket actual parameter if it is another expression

```
- floor (6.7+8.9);
> 15 : int
```

```
bracket nested function calls
```

```
- real (floor 11.12);
> 11.0 : real
```

Strings

```
string - string type constructor

any sequence of characters within "..."

- "hello there";

> "hello there" : string

escape sequence for non-printing characters

\n  - newline

\t  - tab
```

Strings

size - function to return number of characters in string

```
- size “hello”;
> 5 : int
```

^ - binary infix operator to join two strings

```
- “milk”^”shake”;
> “milkshake” : string
```

Characters

```
char - character type constructor
single character string preceded by #
- #”a”;
> #”a” : char
NB string is not a list of char
```

Characters

`ord` - function to convert character to integer ASCII value

`chr` - function to convert integer ASCII value to character

```
- ord #”a”;

> 97 : int

- chr 48;

> #”0” : char
```

Booleans

```
bool - boolean type constructor

true, false - boolean values

- true;

> true : bool

not - unary boolean negation function

- not false;

> true : bool

binary infix operators

andalso  - boolean conjunction

orelse   - boolean disjunction
```

Booleans

group operations with `(...)`

`precedence:`

```
(...) before

not before

andalso before

orelse


- true orelse false andalso true;

> true : bool
```

Booleans

sequential implementation of `andalso and orelse`

`X`	`Y`	`X andalso Y`
`false`	`false`	**`false`**
`false`	`true`	**`false`**
`true`	`false`	`false`
`true`	`true`	`true`

`X andalso Y is false if X is false`

`no need to evaluate Y if X is false`

Booleans

sequential implementation of `andalso and orelse`

`X`	`Y`	`X orelse Y`
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	**`true`**
`true`	`true`	**`true`**

`X orelse Y is true if X is true`

`no need to evaluate Y if X is true`

Booleans

can’t so easily have sequential operators in imperative languages

`X and Y might be function calls`

`evaluating Y might change a global variable`

`side effect`

`so if Y not evaluated than variable not changed...`

Tuples

fixed size collection of values of varying type

like the fields of a Java object

`(`*`exp1`*`,`*`exp2`*`,...,`*`expN`*`)`

```
- (1,1.0,”one”);
> (1,1.0,”one”) : int * real * string
```

Tuples

if:

exp1 : *type1*,... *expN* : *typeN*

then:

(*exp1*,...*expN*) : *type1*...* *typeN*

product type

size of tuple domain is:

size of *type1* domain * ... *
size of *typeN* domain

Tuples

to select element from tuple:

```
#index tuple


- #1 (1,1.0,”one”);
> 1 : int
can nest tuples
- ((“Bianca”,”Castafiore”),”singer);
> ((“Bianca”,”Castafiore”),”singer”) :
   (string * string) * string
```

Tuples

- select from nested tuples with nested selection

```
- #2 (#1 ((“Cuthbert”,”Calculus”),
          ”inventor”)));
> “Calculus” : string
```

Comparison

equality type

any type which allows equality testing

all types except real, functions and streams

binary infix operators

```
=    - equality
<>   - inequality
both operands must be same type
return a boolean
```

Comparison

```
- “banana” <> “banana”;
> false : bool
- (1,”one”) = (1,”won”);
> false : bool
- ((“Captain”,”Haddock”),”sailor”) =
  ((“Captain”,”Haddock”),”sailor”);
> true : bool
```

Comparison

binary infix order operators

`< - less than`

`<=   - less than or equal`

`> - greater than`

`>=   - greater than or equal`

`overloaded for integer, real, string and character`

`both operands must be same type`

`return a boolean`

Comparison

precedence:

```
(...) before

function call before

arithmetic operator before

comparison before

boolean operator
```

```
- 3*4>5*6;

> false : bool
```

Comparison

strings compared for alphabetic order

```
- “ant”<”bee”;
> true : bool
- “anthill”>”ant”;
> true : bool
```

Comparison

NB `not is a function so must bracket a comparison to negate it`

```
- not 1<2;
```

> *`type error - can’t apply not to integer 1`*

```
- not (1<2);
> false : bool
```