CS:3820
Programming Language Concepts

Imperative languages, environment and store, micro-C

Overview

- A Naive imperative language
- C concepts
  - Pointers and pointer arithmetics, arrays
  - Lvalue and rvalue
  - Parameter passing by value and by reference
  - Expression statements
- Micro-C, a subset of C
  - abstract syntax
  - lexing and parsing
  - interpretation

A naive-store imperative language

- **Naive** store model:
  - a variable name maps to an integer value
  - so store is just a runtime environment

```
sum = 0;
for i = 0 to 100 do
  sum = sum + i;
```

i	100
sum	5050

```
i = 1;
sum = 0;
while sum < 10000 do begin
  sum = sum + i;
  i = 1 + i;
end;
```

i	142
sum	10011

Naive-store statement execution, 1

- Executing a statement gives a new store
- Assignment **`x = e`** updates the store
- Expressions do not affect the store

```
let rec exec stmt (store : naivestore) : naivestore =
    match stmt with
      | Asgn(x, e) ->
        setSto store (x, eval e store)
      | If(e1, stmt1, stmt2) ->
        if eval e1 store <> 0 then exec stmt1 store
                              else exec stmt2 store
      | ...
```

Update store at x with value of e

Naive-store statement execution, 2

- A block `{`s_1`; …; `s_n`}` executes s_1 then s_2 …
- Example:

exec (Block [s_1; s_2]) store // F# interpreter

= loop [s_1; s_2] store

= exec s2 (exec s1 store)

```
let rec exec stmt (store : naivestore) : naivestore =
    match stmt with
      | Block stmts ->
        let rec loop ss sto =
                match ss with
                  | []     -> sto
                  | s1::sr -> loop sr (exec s1 sto)
        loop stmts store
      | ...
```

Naive-store statement execution, 3

- **`for`** and **`while`** update the store sequentially

```
let rec exec stmt (store : naivestore) : naivestore =
    match stmt with
    | ...
    | For(x, estart, estop, stmt) -> ...
    | While(e, stmt) ->
      let rec loop sto =
              if eval e sto = 0 then sto
                                else loop (exec stmt sto)
      loop store
```

Environment and store, micro-C

- The naive model cannot describe *pointers* and *variable aliasing*
- A more realistic store model:
  - *Environment* maps a variable name to an address
  - *Store* maps address to value

The essence of C: Pointers

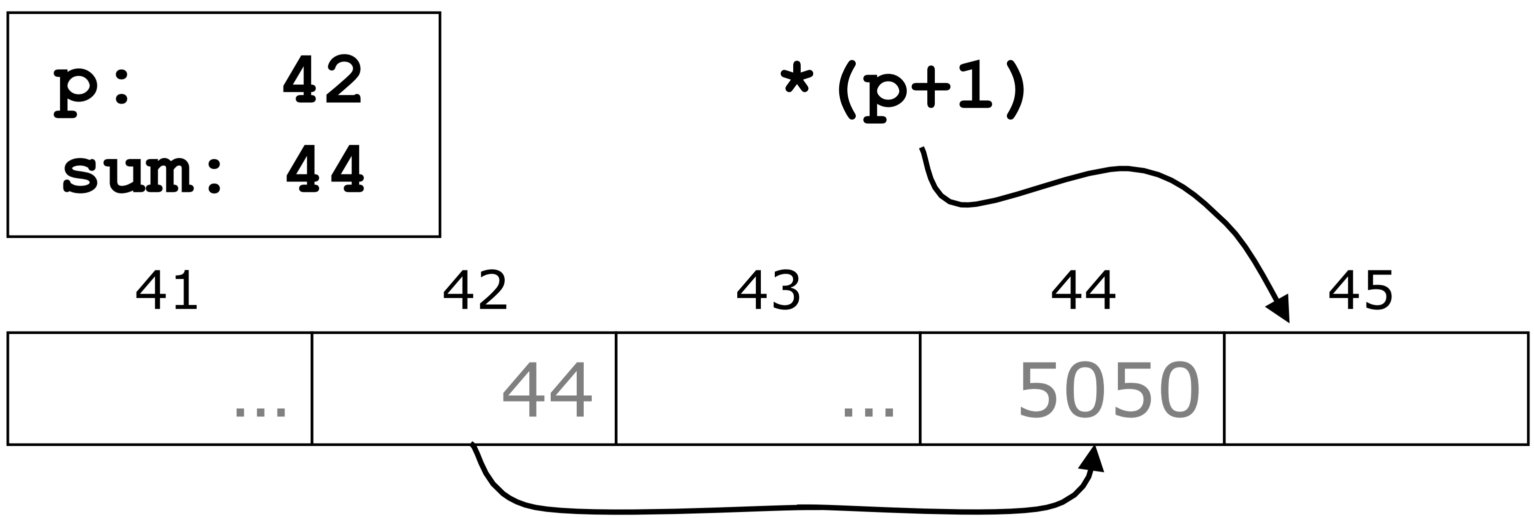
- Main innovations of C (1972) over Algol 60:
  - Structs, as in COBOL and Pascal
  - Pointers, pointer arithmetic, pointer types, array indexing as pointer indexing
  - Syntax: { } for blocks, as in C++, Java, C#
- Very different from Java and C#, which have no pointer arithmetic, but garbage collection

Desirable language features

	C	C++	F#/ML	Smtalk	Haskell	Java	C#
Garbage collection							
Exceptions							
Bounds checks							
Static types							
Generic types (para. polym.)							
Pattern matching							
Reflection							
Refl. on type parameters							
Anonymous functions (λ)							
Streams							
Lazy eval.							

C variable basics

- A variable **`x`** refers to an address (storage location)
- Addresses are mapped to values in the store
- *Pointers* are variables whose values is an address



C pointer basics

- The value of a pointer **`p`** is a storage location (address)

- The dereference expression **`*p`** means:
  - *the content of the location* (rvalue) as in **`*p + 4`**
  - *the storage location itself* (lvalue), as in **`*p = x+4`**

C pointer basics

- The pointer that points to **`x`** is **`&x`**
- Pointer arithmetic:
  `*(p+1)` is the content of the loc just after **`p`**
- If **`p`** equals **`&a[0]`**
  then **`*(p+i)`** equals **`p[i]`** equals **`a[i]`**,
  so an array is a pointer
- **`a[0]`** equals **`*a`**

Lvalue and rvalue of an expression

- Rvalue is "normal" value, right-hand side of assignment: **`17`**, **`true`**
- Lvalue is "location", left-hand side of assignment: **`x`**, **`a[2]`**
- In assignment **`e1 = e2`**, expression **`e1`** must have lvalue

	Has lvalue	**Has rvalue**
`x`	yes	yes
`a[2]`	yes	yes
`*p`	yes	yes
`x+2`	no	yes
`&x`	no	yes

C variable declarations

Declaration	Meaning
int n	n is an integer
int *p	p is a pointer to integer
int ia[3]	ia is array of 3 integers
int *ipa[4]	ipa is array of 4 pointers to integers
int (*iap)[3]	iap is pointer to array of 3 integers
int *(*ipap)[4]	ipap is pointer to array of 4 pointers to ints

Unix program **`cdecl`** or www.cdecl.org may help:

```
cdecl> explain int *(*ipap)[4]
declare ipap as pointer to array 4 of pointer to int
cdecl> declare n as array 7 of pointer to pointer to int
int **n[7]
```

Using pointers for return values

- Example ex5.c, computing square(x):

```
void main(int n) {
   ...
   int r;
   square(n, &r);
   print r;
}

void square(int i, int *rp) {
   *rp = i * i;
}
```

for input

for return value: a pointer to where to put the result

Recursion and return values

- Computing factorial with micro-C/ex9.c

```
void main(int i) {
  int r;
  fac(i, &r);
  print r;
}

void fac(int n, int *res) {
  if (n == 0)
    *res = 1;
  else {
    int tmp;
    fac(n-1, &tmp);
    *res = tmp * n;
  }
}
```

- **`n`** is input parameter
- **`res`** is output parameter: a pointer to where to put the result
- **`tmp`** holds the result of the recursive call
- **`&tmp`** gets a pointer to **`tmp`**

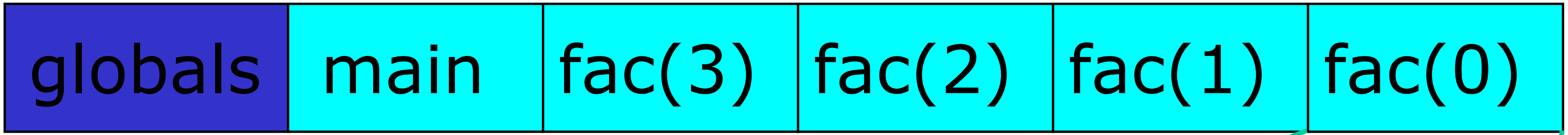
Possible evaluation of main(3)

main(3):
    fac(3, 117):   **&r** is 117
        fac(2, 118):   **&tmp** is 118
            fac(1, 119):   **&tmp** is 119
                fac(0, 120):   **&tmp** is 120
                    *120 = 1
                *119 = 1 * 1   **n** is 1
            *118 = 1 * 2   **n** is 2
        *117 = 2 * 3   **n** is 3
    print 6

...	117	118	119	120	121
...	6	2	1	1	...

Storage model for micro-C

- The store is an indexable stack
  - Bottom: global variables at fixed addresses
  - Plus, a stack of activation records



- An *activation record* is an executing function
  - return address and other administrative data
  - parameters and local variables
  - temporary results

Call-by-value and call-by-reference, C#

```
int a = 11;
int b = 22;
swapV(a, b);
swapR(ref a, ref b);
```

```
a: 41
b: 42
```

addresses

by value

```
static void swapV(int x, int y) {
  int tmp = x; x = y; y = tmp;
}
```

```
x:   43
y:   44
tmp: 45
```

by reference

```
static void swapR(ref int x, ref int y) {
  int tmp = x; x = y; y = tmp;
}
```

```
x:   41
y:   42
tmp: 43
```

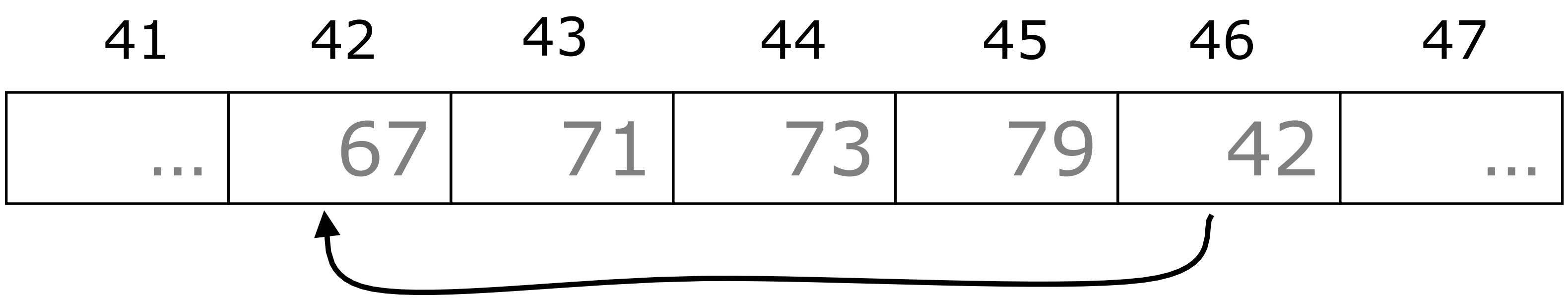
store

41	42	43	44	45
11	22	22	11	11

micro-C array layout

- An array **`int arr[4]`** consists of
  - its 4 int elements
  - a pointer to arr[0]

`arr: 46`



- This is the uniform array representation of B
- Actual C treats array parameters and local arrays differently; complicates compiler

micro-C syntactic concepts

• Types	Abstract Syntax
`int`	`TypI`
`int *x`	`TypP(TypI)`
`int x[4]`	`TypA(TypI, Some 4)`

- Expressions

```
(*p + 1) * 12
```

- Statements

```
if (x != 0) y = 1/x;
```

- Declarations
  - of global or local variables

    ```
    int x;
    ```

  - of global functions

    ```
    void swap(int *x, int *y) { ... }
    ```

micro-C abstract syntax

```
type typ =
  | TypI                                (* Type int                    *)
  | TypC                                (* Type char                   *)
  | TypA of typ * int option            (* Array type                  *)
  | TypP of typ                         (* Pointer type                *)
and expr =
  | Access of access                    (* x    or  *p    or  a[e]     *)
  | Assign of access * expr             (* x=e  or  *p=e  or  a[e]=e   *)
  | Addr of access                      (* &x   or  &*p   or  &a[e]    *)
  | CstI of int                         (* Constant                    *)
  | Prim1 of string * expr              (* Unary primitive operator    *)
  | Prim2 of string * expr * expr       (* Binary primitive operator   *)
  | Andalso of expr * expr              (* Sequential and              *)
  | Orelse of expr * expr               (* Sequential or               *)
  | Call of string * expr list          (* Function call f(...)        *)
and access =
  | AccVar of string                    (* Variable access        x    *)
  | AccDeref of expr                    (* Pointer dereferencing  *p   *)
  | AccIndex of access * expr           (* Array indexing         a[e] *)
and stmt =
  | If of expr * stmt * stmt            (* Conditional                 *)
  | While of expr * stmt                (* While loop                  *)
  | Expr of expr                        (* Expression statement   e;   *)
  | Return of expr option               (* Return from method          *)
  | Block of stmtordec list             (* Block: grouping and scope   *)
and stmtordec =
  | Dec of typ * string                 (* Local variable declaration  *)
  | Stmt of stmt                        (* A statement                 *)
and topdec =
  | Fundec of typ option * string * (typ * string) list * stmt
  | Vardec of typ * string
and program =
  | Prog of topdec list
```

Types

Expressions

rvalue

lvalue

Statements

Declarations

Lexer specification for micro-C

- New: endline comments `// blah blah`
  and delimited comments `if (x /* y? */)`

```
rule Token = parse
  | ...
  | "//"            { EndLineComment lexbuf; Token lexbuf }
  | "/*"            { Comment lexbuf; Token lexbuf }
```

```
and EndLineComment = parse
   | ['\n' '\r']       { () }
   | (eof | '\026')   { () }
   | _                 { EndLineComment lexbuf }
```

```
and Comment = parse
   | "/*"              { Comment lexbuf; Comment lexbuf }
   | "*/"              { () }
   | ['\n' '\r']       { Comment lexbuf }
   | (eof | '\026')   { lexerError lexbuf "Unterminated" }
   | _                 { Comment lexbuf }
```

Parsing C variable declarations

- Hard, declarations are *mixfix*: **`int *x[4]`**
- Parser trick: Parse a variable declaration as a type followed by a variable description:

type info

`int` **`*x[4]`**

`TypI`   **`((fun t -> TypA (TypP t, Some 4)), "x")`**

- Parse var description to get pair **`(f,x)`** of type function **`f`**, and variable name **`x`**
- Apply **`f`** to the declared type to get type of **`x`**

`Vardec(TypA(TypP TypI,Some 4), "x")`

Interpreting micro-C in F#

- Interpreter data:
  - locEnv, *environment* mapping local variable names to store addresses
  - gloEnv, *environment* mapping global variable names to store addresses, and global function names to (parameter list, body statement)
  - *store*, mapping addresses to (integer) values

- Main interpreter functions:

```
exec: stmt -> locEnv -> gloEnv -> store -> store
eval: expr -> locEnv -> gloEnv -> store -> int * store
access: access -> locEnv -> gloEnv -> store ->
        address * store
```

micro-C statement execution

- As with the naive language, but two envs:

```
let rec exec stmt locEnv gloEnv store : store =
    match stmt with

    | If(e, stmt1, stmt2) ->
      let (v, store1) = eval e locEnv gloEnv store
      if v<>0 then exec stmt1 locEnv gloEnv store1
              else exec stmt2 locEnv gloEnv store1

    | While(e, body) ->
      let rec loop store1 =
          let (v, store2) = eval e locEnv gloEnv store1
          if v<>0 then loop (exec body locEnv gloEnv store2)
                  else store2
      loop store

    | ...
```

Expression statements in C, C++, Java and C#

- The “assignment statement”

  `x = 2+4;`

  is really an expression

  `x = 2+4`

  followed by a semicolon

Value: none
Effect: change x

Value: 6
Effect: change x

- The semicolon means: ignore value

```
let rec exec stmt locEnv gloEnv store : store =
    match stmt with
      | ...
      | Expr e ->
        let (_, store1) = eval e locEnv gloEnv store
        store1
```

Evaluate expression then ignore its value

micro-C expression evaluation, 1

- Evaluation of an expression
  - takes local and global env and a store
  - gives a resulting *rvalue* and a *new store*

```
and eval e locEnv gloEnv store : int * store =
    match e with
      | ...
      | CstI i -> (i, store)

      | Prim2(ope, e1, e2) ->
        let (i1, store1) = eval e1 locEnv gloEnv store
        let (i2, store2) = eval e2 locEnv gloEnv store1
        let res =
            match ope with
              | "*"  -> i1 * i2
              | "+"  -> i1 + i2
              | ...
        (res, store2)
```

micro-C expression evaluation, 2

- To evaluate access expression **`x`**, **`*p`**, **`arr[i]`**
  - find its lvalue, as a location **`loc`**
  - look up the rvalue in the store, as **`store1[loc]`**
- To evaluate **`&e`**
  - just evaluate **`e`** as lvalue
  - return the lvalue

rvalue

```
eval e locEnv gloEnv store : int * store =
  match e with
    | Access acc ->
      let (loc, store1) = access acc locEnv gloEnv store
      (getSto store1 loc, store1)

    | Addr acc -> access acc locEnv gloEnv store
    | ...
```

micro-C access evaluation, to *lvalue*

- A variable **`x`** is looked up in environment
- A dereferencing **`*e`** just evaluates **`e`** to an address
- An array indexing **`arr[idx]`**
  - evaluates **`arr`** to address a, then gets aval=store[a]
  - evaluates **`idx`** to rvalue index i
  - returns address (aval+i)

lvalue

```
access acc locEnv gloEnv store : int * store =
  match acc with
  | AccVar x             -> (lookup (fst locEnv) x, store)

  | AccDeref e           -> eval e locEnv gloEnv store

  | AccIndex(arr, idx) ->
    let (a, store1) = access arr locEnv gloEnv store
    let aval = getSto store1 a
    let (i, store2) = eval idx locEnv gloEnv store1
    (aval + i, store2)
```

Operators &x and *p are inverses

- The address-of operator `&` in `&e`
  - evaluates `e` to its lvalue (address) and returns it as an rvalue

  Ex: `&x == 41, &p == 42`
- The dereferencing operator `*` in `*e`
  - evaluates `e` to its rvalue and returns as an lvalue

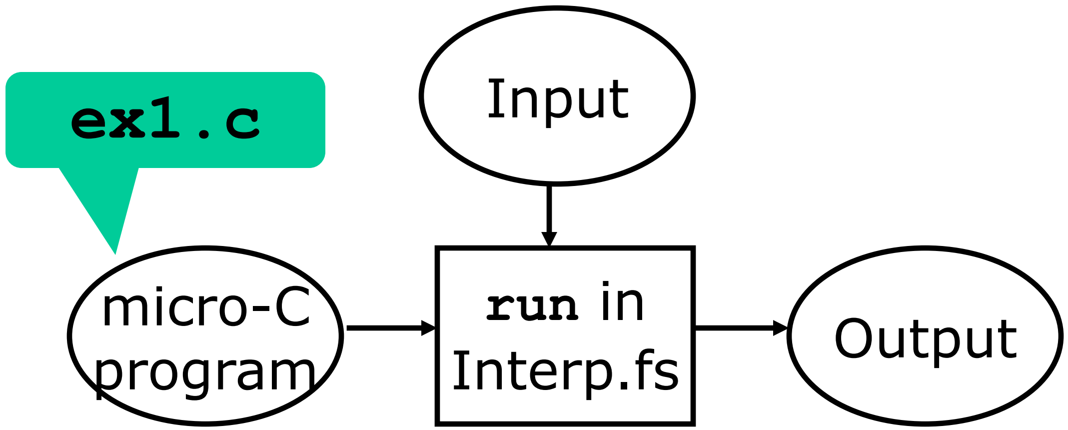
  Ex: `*p` is effectively the same as `y`
- It follows that
  1. `&(*e)` equals `e` — Ex: `&(*p) == &y == 45 == p`
  2. `*(&e)` equals `e`, provided `e` has lvalue — Ex: `*(&y) == *45 == 6 == y`

`x:`	41
`y:`	45
`p:`	42

41	42	43	44	45
1	45	7	8	6

micro-C, interpreter and compiler

- So far: Interpretation of micro-C



- Next: Compilation of micro-C