CS:3820
Programming Language Concepts

Imperative languages, environment and store, micro-C
Overview

• A Naive imperative language
• C concepts
  - Pointers and pointer arithmetic, 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
Imperative Programming

- Based on *stateful* computation
- A *state* is a mapping from variables to values
- (Terminating) programs start with an initial state and end with a final state
- Programs are sequences of *statements*
- Each statement modifies the current state
- Statements may contain *expressions*, which are evaluated to *values*
- Some expressions have also the *side effect* of modifying the store
A naive-store imperative language

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

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

i = 1;
sum = 0;
while sum < 10000 do begin
    sum = sum + i;
    i = 1 + i;
end;
```
Naive-store statement execution, 1

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

```ml
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; \ldots; 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)

```fsharp
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

```ml
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

```
i:  42
sum: 44
```

```
+---+---+---+---+---+
| 41 | 42 | 43 | 44 | 45 |
|    | 100|    | 5050|    |
+---+---+---+---+---+
```
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

<table>
<thead>
<tr>
<th>Feature</th>
<th>C</th>
<th>C++</th>
<th>F#/ML</th>
<th>Smtalk</th>
<th>Haskell</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garbage collection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exceptions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounds checks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static types</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic types (para. polym.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern matching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reflection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Refl. on type parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anonymous functions (λ)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lazy eval.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```
p:   42
sum: 44
```

```
41 42 43 44 45
...
   44 ...
     5050
```

\[ * (p+1) \]
C variable basics

• What a variable \( x \) refers to depends on its position in a statement

• It can be:
  - *the content of \( x \) ’s storage location* (rvalue), as in \( x + 4 \)
  - *the storage location itself* (lvalue), as in \( x = y + 4 \)
C pointer basics

• The value of a pointer \( p \) is a storage location (address)

• Depending on context, 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 *usual* value, on right-hand side of assignment: 17, true

- Lvalue is *location*, on left-hand side of assignment: x, a[2]

- In assignment e1 = e2, expression e1 must have lvalue

<table>
<thead>
<tr>
<th></th>
<th>Has lvalue</th>
<th>Has rvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>a[2]</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>*p</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>x+2</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>&amp;x</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
C variable declarations

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>int n</td>
<td>n is an integer</td>
</tr>
<tr>
<td>int *p</td>
<td>p is a pointer to integer</td>
</tr>
<tr>
<td>int ia[3]</td>
<td>ia is array of 3 integers</td>
</tr>
<tr>
<td>int *ipa[4]</td>
<td>ipa is array of 4 pointers to integers</td>
</tr>
<tr>
<td>int (*iap)[3]</td>
<td>iap is pointer to array of 3 integers</td>
</tr>
<tr>
<td>int **p</td>
<td>p is a pointer to a pointer to an integer</td>
</tr>
</tbody>
</table>

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

tnt **n[7]
Using pointers for return values

Example ex5.c, computing square(x):

```c
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

```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         1 ...
Storage model for micro-C

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

<table>
<thead>
<tr>
<th>globals</th>
<th>main</th>
<th>fac(3)</th>
<th>fac(2)</th>
<th>fac(1)</th>
<th>fac(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin. data</td>
<td>params+locals</td>
<td>temps</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- An array int a[4] consists of
  - its 4 int elements
  - a pointer to a[0]
Operators &x and *p are inverses

<table>
<thead>
<tr>
<th>x: 41</th>
<th>41</th>
<th>42</th>
<th>43</th>
<th>44</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>y: 45</td>
<td>1</td>
<td>45</td>
<td>7</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>p: 42</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
</tr>
</tbody>
</table>

• 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,          Ex: *(&y) == *45 == 6 == y
     provided e has lvalue
Modifying input parameters

```c
int a = 11;
int b = 22;
swap1(a, b);
swap2(&a, &b);
```

```c
static void swap1(int x, int y) {
    int tmp = x; x = y; y = tmp;
}
```

```c
static void swap2(int *p, int *q) {
    int tmp = *p; *p = *q; *q = tmp;
}
```

```
int a = 11;
int b = 22;
swap1(a, b);
swap2(&a, &b);
```

```
11 41 22 22 11 11
```
Micro-C syntactic concepts

Types
- int
- int *x
- int x[4]

Abstract Syntax
- TypI
- TypP(TypI)
- 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) { ... }


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
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, 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
Interpreter’s functions

run: program -> int list -> store
(run p [a1; ...; an])
Execute program p by initializing global vars and then calling p’s main function with args a1; ...; an, returning final store

exec: stmt -> locEnv -> gloEnv -> store -> store
(exec sta lenv genv sto)
Execute statement sta in local env lenv and global env genv and store sto, returning updated store

stmtordec: stmtordec -> locEnv -> gloEnv -> store -> locEnv * store
(stmtordec sd lenv genv sto)
Execute statement or declaration sd in local env lenv and global env genv and store sto, returning updated local env and store
Interpreter’s functions

**eval**: expr -> locEnv -> gloEnv -> store -> int * store

(eval e lenv genv st)
Evaluate expression e in local env lenv and global env genv and store sto, returning e’s result and updated store

**access**: access -> locEnv -> gloEnv -> store -> address * store

(access a lenv genv st)
Evaluate access expression a in local env lenv and global env genv and store sto, returning an address and updated store

**allocate**: typ * string -> locEnv -> store -> locEnv * store

(allocate t x lenv sto)
Bind var x in local env lenv and allocate space for x in store sto, returning updated updated env and 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

• 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

Value: none
Effect: change x

Value: 6
Effect: change x
Micro-C expression evaluation, 1

• Evaluation of an expression
  - takes local and global env and a store
  - gives a resulting \textit{rvalue} and a new \textit{store}

```plaintext
and eval e locEnv gloEnv store : int * store =
  match e with
  | ... |
  | CstI i -> (i, store) |
  | Prim2(op, e1, e2) ->
    let (i1, store1) = eval e1 locEnv gloEnv store in
    let (i2, store2) = eval e2 locEnv gloEnv store1 in
    let res =
      match op with
      | "*"    -> i1 * i2
      | "+
```
Micro-C expression evaluation, 2

- To evaluate access expression \( x, *p, arr[i] \)
  - find its lvalue, as a location \( \text{loc} \)
  - look up the rvalue in the store, as \( \text{store1}[\text{loc}] \)

- To evaluate \&e
  - just evaluate \( e \) as lvalue
  - return the lvalue

```plaintext
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 \( a[e] \)
  - evaluates \( a \) to address \( k \), then gets \( v = \text{store}[k] \)
  - evaluates \( e \) to rvalue index \( i \)
  - returns address \( v+i \)

```ocaml
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(a, e) ->
    let (k, store1) = access a locEnv gloEnv store
    let v = getSto store1 k
    let (i, store2) = eval e locEnv gloEnv store1
    (v + i, store2)
```
• New: endline comments // blah blah
and delimited comments if (x /* y? */)

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

and EndLineComment = parse
| ['
' ']
| (eof | '') }) |
| _               { EndLineComment lexbuf } |

and Comment = parse
| "/"          { Comment lexbuf; Comment lexbuf }
| "/*/"         { () } |
| ['
' ']
| (eof | '') }) |
| _               { Comment lexbuf }
| (eof | '') }) |
| _               { lexerError lexbuf "Unterminated" } |
Parsing C variable declarations

• Hard, declarations are *mixfix*: `int *a[4]`

• Parser trick: Parse a variable declaration as a type followed by a variable description:

• 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), "a")
```

```
TypI ((fun t -> TypA (TypP t, Some 4)), "a")
```
Micro-C, interpreter and compiler

• So far: Interpretation of micro-C

• Next: Compilation of micro-C