Fundamentals of Computing: Lecture 25

Piyush P Kurur
Office no: 224
Dept. of Comp. Sci. and Engg.
IIT Kanpur

October 5, 2009

Summary of last class

Summary of last class

- A structure is a way of constructing product types.

Summary of last class

- A structure is a way of constructing product types.

```
struct structure-name{
  field declarations
};
```

Summary of last class

- A structure is a way of constructing product types.

```
struct structure-name{
  field declarations
};
```

- If `foo` is of type `strunct Foo` and bar is a field of the structure `Foo` then the `bar` field of `foo` can be accessed via the expression `foo.bar`

Summary of last class

- A structure is a way of constructing product types.

```
struct structure-name{
  field declarations
};
```

- If `foo` is of type `strunct Foo` and bar is a field of the structure `Foo` then the `bar` field of `foo` can be accessed via the expression `foo.bar`
- Fields are both l-values as well as r-values.

Summary of last class

- A structure is a way of constructing product types.

```
struct structure-name{
  field declarations
};
```

- If `foo` is of type `strunct Foo` and bar is a field of the structure `Foo` then the `bar` field of `foo` can be accessed via the expression `foo.bar`
- Fields are both l-values as well as r-values.
- Structures are passed as values to function which is unlike Java.

Examples

Examples

```
struct Vector2D {
  double x;
  double y;
};
```

Examples

```
struct Vector2D {
  double x;
  double y;
};

typedef struct{
  double x;
  double y;
} Vector2D;
```

Examples

```
struct Vector2D {
  double x;
  double y;
};

typedef struct{
  double x;
  double y;
} Vector2D;

struct Vector2D{
  double x;
  double y;
} origin = {0,0};
```

Structures are *not* passed as reference to functions.

Structures are *not* passed as reference to functions.

```
#line 214 "lecture23.lhs"
#include <stdio.h>
void printVector(struct Vector2D);
void shiftByOneUnit(struct Vector2D u);
int main () {
  printf("Before shift: ");printVector(origin);
  shiftByOneUnit(origin);
  printf("After shift: ");printVector(origin);
}
void shiftByOneUnit(struct Vector2D u){
  u.x = u.x + 1;
}
void printVector(struct Vector2D u){
  printf("(%f,%f)\n",u.x,u.y);
}
```

Recursive data types

- List.

Recursive data types

- List. A list is either an empty list or an element followed by a list.

Recursive data types

- List. A list is either an empty list or an element followed by a list.
- Binary trees.

Recursive data types

- List. A list is either an empty list or an element followed by a list.
- Binary trees. A binary tree is either an empty Tree or a root node with two children left subtree and right subtree.

Recursive data types

- List. A list is either an empty list or an element followed by a list.
- Binary trees. A binary tree is either an empty Tree or a root node with two children left subtree and right subtree.
- General trees (some times called Rose trees).

Recursive data types

- List. A list is either an empty list or an element followed by a list.
- Binary trees. A binary tree is either an empty Tree or a root node with two children left subtree and right subtree.
- General trees (some times called Rose trees). Either an empty tree or a node with a Forest of subtrees.

Recursive data types

- List. A list is either an empty list or an element followed by a list.
- Binary trees. A binary tree is either an empty Tree or a root node with two children left subtree and right subtree.
- General trees (some times called Rose trees). Either an empty tree or a node with a Forest of subtrees.

Mathematically

$$
\begin{aligned}
L(a) &= \bot + a \times L(a) \\
BT(a) &= \bot + a \times BT(a) \times BT(a) \\
T(a) &= \bot + a \times F(a) \\
F(a) &= L(T(a))
\end{aligned}
$$

In Haskell this would be

```
data List    a = Empty | Cons a (List a)
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
data Tree    a = Empty | Node a [Tree a]
```

How does one simulate this in C?

How does one simulate this in C?

An ugly dance involving structs and pointers.

How does one simulate this in C?

An ugly dance involving structs and pointers.

```
typedef struct Node Node;
struct Node {
  int datum;
  Node * next;
};
```

How does one simulate this in C?

An ugly dance involving structs and pointers.

```
typedef struct Node Node;
struct Node {
  int datum;
  Node * next;
};

typedef Node *List;
```

The `head` and `tail` function

```
data List     a = Empty | Cons a (List a)
head (Cons x _)  = x
tail (Cons _ xs) = xs
```

The head and tail function

```
data List    a = Empty | Cons a (List a)
head (Cons x _)  = x
tail (Cons _ xs) = xs
```

```
int head(List l)
{
  if( l == NULL) {error("head of an empty list");}
  else return (*l).datum
}
List tail (List l)
{
  if( l == NULL) {error("tail of an empty list");}
  else return (*l).next
}
```

The head and tail function

```
data List    a = Empty | Cons a (List a)
head (Cons x _)  = x
tail (Cons _ xs) = xs
```

```
int head(List l)
{
  if( l == NULL) {error("head of an empty list");}
  else return (*l).datum
}
List tail (List l)
{
  if( l == NULL) {error("tail of an empty list");}
  else return (*l).next
}
```

Since `(*foo).bar` often comes in practice

The head and tail function

```
data List    a = Empty | Cons a (List a)
head (Cons x _)  = x
tail (Cons _ xs) = xs
```

```
int head(List l)
{
  if( l == NULL) {error("head of an empty list");}
  else return l -> datum
}
List tail (List l)
{
  if( l == NULL) {error("tail of an empty list");}
  else return l -> next
}
```

Since `(*foo).bar` often comes in practice