

Assignment 3

Ruofei HUANG(z5141448) Anqi ZHU(z5141541)

May 30, 2018

1 Task 1

1.1 Some useful symbol definitions about manipulating words

A **word**, as mentioned in the assignment specification, is defined as a finite sequence of letters over $L = \{ 'a', \dots, 'z' \}$. The specification already defines the relationship of ' \leq ' between a **word** v and a **word** w (this means v is a prefix of w or w itself if $v \leq w$).

So we would like to further this definition and define a relationship of ' $<$ ' when v is a proper prefix of w which cannot be w itself if $v < w$. That means:

$$v < w \Leftrightarrow v \leq w \wedge v \neq w$$

We also would like to define a symbol $|w|$ that represents the length of a *word* w . We formally define this by:

$$|w| = \begin{cases} 0 & \text{if } w = \epsilon \\ 1 + |w'| & \text{else} \end{cases}$$

$$\text{where } \exists l \in L \{w = w' l\}$$

1.2 Syntactic(Abstract) Data Type *Dict*

Inspired by the program sketch and the assignment statement, we could describe the syntactic data type *Dict* as below (the encapsulated state would be a dictionary word set W).

$$Dict = (W = \phi, \left(\begin{array}{l} \mathbf{proc} \text{ addword}^{Dict}(\mathbf{word} \ w) \cdot b, W : [\mathbf{TRUE}, b = b_0 \wedge W = W_0 \cup \{w\}] \\ \mathbf{func} \text{ checkword}^{Dict}(\mathbf{word} \ w) : \mathbb{B} \cdot \\ \quad \mathbf{var} \ b \cdot b, W : [\mathbf{TRUE}, b = (w \in W) \wedge W = W_0]; \mathbf{return} \ b \\ \mathbf{proc} \text{ delword}^{Dict}(\mathbf{word} \ w) \cdot b, W : [w \in W, b = b_0 \wedge W = W_0 \setminus \{w\}] \end{array} \right))$$

2 Task 2

2.1 Data Type Refinement

Now we would like to refine *Dict* to a second data type *DictA* where we replace *W* with a trie *t*. We would also like to define the domain of a *t* as **dom**(*t*). We shall use this definition later in our refinement.

2.1.1 Inductive Relation Predicate

The correspondence between the two state space *W* and *t* is captured by the inductively defined predicate as follows:

$$r = (W = \{w \in \mathbf{dom}(t) | t(w) = 1\})$$

We can translate this into a relation function that transfers a concrete state space *t* to an abstract state space *W*. The function is as follows:

$$f(t) = \{w \in \mathbf{dom}(t) | t(w) = 1\}$$

With that in mind, we can propose the initialisation predicate and corresponding operations of *DictA*.

2.1.2 Initialisation Predicate

We would like to define the initialisation predicate of *DictA* as follows:

$$\mathit{init}^{DictA} = (t := \{\epsilon \mapsto 0\})$$

2.1.3 Operations

We would like to define the operations of *DictA* as follows:

```
proc addwordDictA(word w) · b, t :  
  [TRUE, b = b0 ∧ t = t0 \ {w ↦ 0} ∪ {w ↦ 1} ∪ {w' < w ∧ w' ∉ dom(t) | w' ↦ 0}]  
func checkwordDictA(word w) : B · var b · b, t :  
  [TRUE, t = t0 ∧ b = (∀ w' ≤ w (w' ∈ dom(t) ∧ t(w) = 1)]; return b  
proc delwordDictA(word w) · b, t :  
  [TRUE, b = b0 ∧ (w ∉ dom(t) ∨ t = (t : w ↦ 0))]
```

2.2 Proof of Refinement

Now we would like to start proving the refinements of the initialization and each operation from *t* to *W*. So that we can prove *DictA* is a data refinement of *Dict*.

2.2.1 Refinement proof for *init*

We start from proving the refinement between $init^{DictA}$ and $init^{Dict}$.

$$\begin{aligned}
& init^{DictA} \Rightarrow init^{Dict}[f(t)/W] \\
\Leftrightarrow & \langle \text{Definition of } init^{DictA} \text{ and } init^{Dict} \rangle \\
& \forall w \in \mathbf{dom}(t) (t(w) = 0) \Rightarrow W = \phi
\end{aligned}$$

Then we move on to prove the refinement of defined operations of *Dict* and *DictA*. We don't need to prove the validity of the condition (3_f) for any operations since their preconditions are always TRUE. We only need to check the validity of the condition (4_f) in all three operations.

2.2.2 Refinement proof for *addword*

$$\begin{aligned}
& pre_{addword}^{Dict}[f(t_0)/W] \wedge post_{addword}^{DictA} \\
\Leftrightarrow & \langle \text{Definition of } addword^{Dict} \text{ and } addword^{DictA} \rangle \\
& TRUE[f(t_0)/W] \wedge b = b_0 \wedge t = t_0 \setminus \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\} \\
\Rightarrow & \langle \text{Definition of } f \rangle \\
& f(t) = f(t_0 \setminus \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}) \\
\Leftrightarrow & \langle \text{Logic, only } w \text{ maps to 1 in } t \text{ (only } w \text{ is newly added into } W) \rangle \\
& f(t) = f(t_0) \cup \{w\} \\
\Leftrightarrow & \langle \text{Definition of } addword^{Dict} \text{ and } addword^{DictA} \rangle \\
& post_{addword}^{Dict}[f(t_0), f(t)/W_0, W]
\end{aligned}$$

2.2.3 Refinement proof for *checkword*

$$\begin{aligned}
& pre_{checkword}^{Dict}[f(t_0)/W] \wedge post_{checkword}^{DictA} \\
\Leftrightarrow & \langle \text{Definition of } checkword^{DictA} \text{ and } checkword^{Dict} \rangle \\
& TRUE[f(t_0)/W] \wedge t = t_0 \wedge b = (\forall w' \leq w (w' \in \mathbf{dom}(t)) \wedge t(w) = 1) \\
\Rightarrow & \langle \text{Definition of } f \rangle \\
& b = (w \in f(t) \wedge t(w) = 1) \wedge t = t_0 \\
\Leftrightarrow & \langle \text{Definition of } checkword^{DictA} \text{ and } checkword^{Dict} \rangle \\
& post_{checkword}^{Dict}[f(t_0), f(t)/W_0, W]
\end{aligned}$$

2.2.4 Refinement proof for *delword*

$$\begin{aligned}
& pre_{delword}^{Dict} [f(t_0)/W] \wedge post_{delword}^{DictA} \\
\Leftrightarrow & \langle \text{Definition of } delword^{DictA} \text{ and } delword^{Dict} \rangle \\
& w \in f(t_0) \wedge b = b_0 \wedge (w \notin \mathbf{dom}(t) \vee t = (t : w \mapsto 0)) \\
\Rightarrow & \langle \text{Definition of } f \rangle \\
& f(t) = f(t_0) \setminus \{w\} \\
\Leftrightarrow & \langle \text{Definition of } delword^{DictA} \text{ and } delword^{Dict} \rangle \\
& post_{delword}^{Dict} [f(t_0), f(t)/W_0, W]
\end{aligned}$$

3 Task 3

3.1 Pre-defined function calls

Before refining the operations in *DictA* into toy language, we would like to first define some useful function calls that helps building our later refinement more close to the real c program constructions.

3.1.1 *POPWORD*

The semantic function *POPWORD* returns a substring w' of the word w starting from the first letter to the *index*'th letter of w . The definition of the function and corresponding function call is as below:

Definition of *POPWORD*

$$POPWORD(w, i) = w' \quad \text{where } w' \leq w \wedge |w'| = i$$

Definition of the function call *popWord*

```

func popWord(value w, value i) : word .
  var w' ·  $\sqcup w' : [i \leq |w|, w' \leq w \wedge |w'| = i]; \lrcorner(P1) \mathbf{return} \ w'$ 

```

Specification of the procedure in *popWord*

$$\begin{aligned}
(P1) \sqsubseteq & \langle |w[0..i-1]| = i \wedge w[0..i-1] \leq w \rangle \\
& w' := w[0..i-1]
\end{aligned}$$

This function helps us to build our refinement as close to the trie construction in c programs as possible. (In c programs, words are splitted into prefixes in increasing length and checked in the trie recursively.)

3.1.2 *doAddword*

Using the previous definition of *POPWORD*, we would like to develop a corresponding function call named *doAddword*(**var** w , **var** $index$) that allows using a variable i to locate which prefix of w (or say which sub-trie of t) the procedure currently is checking at during the entire word-adding operation. This function will be called by the function *addword* of *DictA* to do most of the adding-word job.

All the prefixes of w should exist in t eventually, so if the current prefix *POPWORD*(w, i) does not exist, *doAddword* will add the new prefix into t and continue searching for the next prefix *POPWORD*($w, i + 1$). Considering this and the fact that t always initializes with ϵ included, it is guaranteed that every *doAddword* operation at i level will already have all prefixes of w with length no longer than i exist in t , which satisfies the precondition of the *doAddword* function itself. The definition of the function and the refinement of its linking procedure is as follows.

Definition of *doAddword*

proc *doAddword*^{*DictA*}(**value** w , **value** $index$)
 $\sqsubseteq t : \left[\begin{array}{l} 0 \leq index \leq |w| \wedge \forall w' \leq w \wedge |w'| \leq index (w' \in \mathbf{dom}(t)) , \\ t = t_0 \setminus \{w \mapsto 0\} \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\} \end{array} \right] \neg(A2)$

Specification of the procedure in *doAddword*

$$\begin{aligned}
(A2) &\sqsubseteq \langle \text{if} \rangle \\
&\quad \text{if } index < |w| \\
&\quad \text{then } \perp t : [index < |w| \wedge pre(A2), post(A2)] \neg(A3-1) \\
&\quad \text{else } \perp t : [index \geq |w| \wedge pre(A2), post(A2)] \neg(A3-2) \\
&\quad \text{fi} \\
(A3-1) &\sqsubseteq \langle \text{seq} \rangle \\
&\quad \perp t : \left[\begin{array}{l} index < |w| \wedge pre(A2), \\ index \geq |w| \wedge pre(A2) \wedge POPWORD(w, index + 1) \in \mathbf{dom}(t) \end{array} \right] \neg(A3-3) \\
&\quad \perp t : \left[\begin{array}{l} index \geq |w| \wedge pre(A2) \wedge POPWORD(w, index + 1) \in \mathbf{dom}(t), \\ post(A2) \end{array} \right] \neg(A3-4) \\
(A3-3) &\sqsubseteq \langle \text{if} \rangle \\
&\quad \text{if } (POPWORD(w, index + 1) \in \mathbf{dom}(t)) \\
&\quad \text{then } \perp t : \left[\begin{array}{l} pre(A3-3) \wedge POPWORD(w, index + 1) \in \mathbf{dom}(t), \\ post(A3-3) \end{array} \right] \neg(A3-3-1) \\
&\quad \text{else } \perp t : \left[\begin{array}{l} pre(A3-3) \wedge POPWORD(w, index + 1) \notin \mathbf{dom}(t), \\ post(A3-3) \end{array} \right] \neg(A3-3-2) \\
&\quad \text{fi} \\
(A3-3-1) &\sqsubseteq \langle POPWORD(w, index + 1) \in \mathbf{dom}(t) \Rightarrow post(A3-3) = \text{TRUE} \rangle \\
&\quad skip; \\
(A3-3-2) &\sqsubseteq \langle \text{Ass} \rangle \\
&\quad t := t \cup \{popWord(w, index) \mapsto 0\}; \\
(A3-4) &\sqsubseteq \langle \text{proc} \rangle \\
&\quad doAddword(w, index + 1) \\
(A3-2) &\sqsubseteq \langle index \geq |w| \wedge index \leq |w| \Rightarrow index = |w| \text{ and definition of } post(A2) \rangle \\
&\quad t := t_0 \setminus \{w \mapsto 0\} \cup \{w \mapsto 1\}
\end{aligned}$$

We gather the code for the body of *doAddword*

```

if  $index < |w|$ 
then
  if  $POPWORD(w, index + 1) \in \mathbf{dom}(t)$ 
  then skip;
  else
     $t := t \cup \{popWord(w, index) \mapsto 0\};$ 
  fi
  doAddword( $w, index + 1$ )
else
   $t := t_0 \setminus \{w \mapsto 0\} \cup \{w \mapsto 1\}$ 
fi

```

3.1.3 *doCheckword*

Similarly to *doAddword*, we would also develop a corresponding function call named *doCheckword*(**var** w , **var** $index$) which will search the complete word w in t based on the precondition that all prefixes of w with length no longer than $index$ are guaranteed to exist in t . This function would be called by *checkword* of *DictA* to do most of the checking-word job. The definition of the function and the refinement of its linking procedure is as follows.

Definition of *doCheckword*

```

func doCheckword(value  $w$ , value  $index : \mathbb{N}$ ) :  $\mathbb{B}$ .
  var  $b \cdot \sqsubseteq b :$   $\left[ \begin{array}{l} 0 \leq index \leq |w| \wedge \\ \forall w' \leq w \wedge |w'| \leq index (w' \in \mathbf{dom}(t)), \\ b = (\forall w' \leq w (w' \in \mathbf{dom}(t)) \wedge t(w) = 1) \end{array} \right] \dashv(C2); \mathbf{return} \ b$ 

```

Specification of the procedure in doCheckword

$$\begin{aligned}
(C2) &\sqsubseteq \langle \text{if} \rangle \\
&\quad \text{if } index < |w| \\
&\quad \text{then } \perp b : [index < |w| \wedge pre(C2), post(C2)] \perp (C3-1) \\
&\quad \text{else } \perp b : [index \geq |w| \wedge pre(C2), post(C2)] \perp (C3-2) \\
&\quad \text{fi} \\
(C3-1) &\sqsubseteq \langle \text{if} \rangle \\
&\quad \text{if } (POPWORD(w, index + 1) \in \mathbf{dom}(t)) \\
&\quad \text{then } \perp b : \left[\begin{array}{l} pre(C3-1) \wedge POPWORD(w, index + 1) \in \mathbf{dom}(t), \\ post(C2) \end{array} \right] \perp (C3-1-1) \\
&\quad \text{else } \perp b : \left[\begin{array}{l} pre(C3-1) \wedge POPWORD(w, index + 1) \notin \mathbf{dom}(t), \\ post(C2) \end{array} \right] \perp (C3-1-2) \\
&\quad \text{fi} \\
(C3-1-1) &\sqsubseteq \langle \text{ass, func} \rangle \\
&\quad b := doCheckword(w, index + 1) \\
(C3-1-2) &\sqsubseteq \langle \text{Ass and } popWord(w, index + 1) \notin \mathbf{dom}(t) \Rightarrow b = \text{FALSE} \rangle \\
&\quad b := \text{FALSE} \\
(C3-2) &\sqsubseteq \langle index \geq |w| \wedge index \leq |w| \Rightarrow index = |w| \text{ and definition of } post(C2) \rangle \\
&\quad b := (t(w) = 1)
\end{aligned}$$

We gather the code of *doCheckword* below:

```

if index < |w|
then
  if POPWORD(w, index + 1) ∈ dom(t)
  then skip
  else return doCheckword(w, index + 1)
fi
else b := (t(w) = 1)
fi

```

3.1.4 doDelword

Similarly to *doCheckword*, we would also develop a corresponding function call named *doDelword*(**var** *w*, **var** *index*) which will set the result of *t(w)* as 0. This function would be called by *delword* of *DictA* to do most of the deleting-word job. The definition of the function and the refinement of its linking procedure is as follows.

Definition of doDelword

```

proc doDelwordDictA(value  $w$ , value  $index$ )
   $\sqsubseteq t : \left[ \begin{array}{l} 0 \leq index \leq |w| \wedge \forall w' \leq w \wedge |w'| \leq index (w' \in \mathbf{dom}(t)), \\ (w \notin \mathbf{dom}(t) \vee t = t_0 : w \mapsto 0) \end{array} \right] \neg(D2)$ 

```

Specification of the procedure in doDelword

```

(D2)  $\sqsubseteq$        $\langle \text{if} \rangle$ 
              if  $index < |w|$ 
              then  $\sqsubseteq t : [index < |w| \wedge pre(D2), post(D2)] \neg(D3-1)$ 
              else  $\sqsubseteq t : [index \geq |w| \wedge pre(D2), post(D2)] \neg(C3-2)$ 
              fi
(D3-1)  $\sqsubseteq$        $\langle \text{if} \rangle$ 
              if ( $POPWORD(w, index + 1) \in \mathbf{dom}(t)$ )
              then  $\sqsubseteq t : \left[ \begin{array}{l} pre(D3-1) \wedge POPWORD(w, index + 1) \in \mathbf{dom}(t), \\ post(D2) \end{array} \right] \neg(D3-1-1)$ 
              else  $\sqsubseteq t : \left[ \begin{array}{l} pre(D3-1) \wedge POPWORD(w, index + 1) \notin \mathbf{dom}(t), \\ post(D2) \end{array} \right] \neg(D3-1-2)$ 
              fi
(D3-1-1)  $\sqsubseteq$        $\langle \text{ass, func} \rangle$ 
              doDelword( $w, index + 1$ )
(D3-1-2)  $\sqsubseteq$        $\langle \text{Ass and } POPWORD(w, index + 1) \notin \mathbf{dom}(t) \Rightarrow post(D2) = \text{TRUE} \rangle$ 
              skip
(D3-2)  $\sqsubseteq$        $\langle index \geq |w| \wedge index \leq |w| \Rightarrow index = |w| \text{ and definition of } post(D2) \rangle$ 
               $t := (t : w \mapsto 0)$ 

```

We gather the code of *doDelword* below:

```

if  $index < |w|$ 
then
  if  $POPWORD(w, index + 1) \in \mathbf{dom}(t)$ 
  then doDelword( $w, index + 1$ )
  else skip
fi
else  $t := (t : w \mapsto 0)$ 
fi

```

Now we could specify the initialisation and all operations of *DictA* into toy language, in which above functions will be called to do the corresponding jobs.

3.2 Specification of *init*

From the spec we have:

$$\begin{aligned} & \mathbf{dom}(t) = \{\epsilon\} \wedge f(t) = \phi \\ \sqsubseteq & \quad \langle \text{ass} \rangle \\ & t := \{\epsilon \mapsto 0\} \end{aligned}$$

3.3 Specification of *addword*

From the spec¹ we have:

$$\begin{aligned} & \mathbf{proc} \text{ addword}^{DictA}(\mathbf{value} \ w) \cdot \\ & \quad \sqcup b, t : [\mathbf{TRUE}, b = b_0 \wedge t = t_0 \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}] \text{ } \textcolor{red}{\dashv(A1)} \\ \textcolor{red}{(A1)} \sqsubseteq & \quad \langle \text{c-frame} \rangle \\ & t : [\mathbf{TRUE}, t = t_0 \cup \{w \mapsto 1\} \cup \{w' < w \wedge w' \notin \mathbf{dom}(t) | w' \mapsto 0\}] \\ \sqsubseteq & \quad \langle \mathbf{proc}, 0 \leq |w| \text{ and } \epsilon \in \mathbf{dom}(t) \text{ since } \textit{init}^{DictA} \rangle \\ & doAddword(w, 0) \end{aligned}$$

3.4 Specification of *checkword*

From the spec we have:

$$\begin{aligned} & \mathbf{func} \text{ checkword}^{DictA}(\mathbf{value} \ w) : \mathbb{B} \cdot \\ & \quad \mathbf{var} \ b \cdot \sqcup b, t : [\mathbf{TRUE}, b = (\forall w' \leq w (w' \in \mathbf{dom}(t)) \wedge t(w) = 1)]; \textcolor{red}{\dashv(C1)} \mathbf{return} \ b \\ \textcolor{red}{(C1)} \sqsubseteq & \quad \langle \text{c-frame} \rangle \\ & b : [\mathbf{TRUE}, b = (\forall w' \leq w (w' \in \mathbf{dom}(t)) \wedge t(w) = 1)]; \\ \sqsubseteq & \quad \langle \mathbf{proc}, 0 \leq |w| \text{ and } \epsilon \in \mathbf{dom}(t) \text{ since } \textit{init}^{DictA} \rangle \\ & b := doCheckword(w, 0); \end{aligned}$$

3.5 Specification of *delword*

From the spec we have:

$$\begin{aligned} & \mathbf{proc} \text{ delword}^{DictA}(\mathbf{value} \ w) \\ & \quad \sqcup \cdot b, t : [\mathbf{TRUE}, b = b_0 \wedge (w \notin \mathbf{dom}(t) \vee t := t : w \mapsto 0)] \text{ } \textcolor{red}{\dashv(D1)} \\ \textcolor{red}{(D1)} \sqsubseteq & \quad \langle \text{c-frame} \rangle \\ & t : [\mathbf{TRUE}, (w \notin \mathbf{dom}(t) \vee t = t_0 : w \mapsto 0)] \\ \sqsubseteq & \quad \langle \mathbf{proc}, 0 \leq |w| \text{ and } \epsilon \in \mathbf{dom}(t) \text{ since } \textit{init}^{DictA} \rangle \\ & doDelword(w, 0) \end{aligned}$$

¹Definition of this is in the Assignment 3 requirements of cs2111.

3.6 Task 4

Now we would translate our data refinement to C functions to match the prototypes given in dict.h.

There are some significant differences between the structure of a trie used in c functions and in our data refinement (because the tries in c functions can be operated as nodes but they are operated as sets in our data refinement). The created new word/prefix attaches to the current trie by being linked to a pointer in its sub-trie, while in our data refinement, t absorbs the new word/prefix w by combining a new function subset containing w in its domain. Also, the trie in c functions uses sub-tries to recursively find the next extend prefix/word, while in our data refinement, we uses the extension of a substring of w ($POPWORD$) to represent the next level of sub-trie.

However, it should be noted that the logic used on both sides is the same (they both use recursive call to derive the final result).

```
1  #include "dict.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  void newdict(Dict *dp){
5      // malloc the space of root node.
6      *dp = malloc(sizeof(struct __tnode__));
7  }
8
9  void doAddword(const Dict r, const word w, int index) {
10     if (w[index] != '\0') {
11         // the word is not ended
12         if (r->cvec[w[index+1]-'a']!=NULL) {
13             /* POPWORD(w,index+1) \in dom(t) */
14             // skip
15         }
16         else{
17             // t:= (POPWORD(w,index+1) -> 0)
18             newdict(&(r->cvec[w[index+1]-'a']));
19         }
20         // recursive call
21         doAddword(r->cvec[w[index+1]-'a'],w , index +1);
22     }
23     else{
24         // index = |w|
25         // t:= t_0 \{w -> 0\} U \{w -> 1\}
26         r->eow = TRUE;
27     }
28 }
```

```

29
30 bool doCheckword(const Dict r, const word w, int index){
31     if (w[index]!='\0') {
32         // the word is not ended
33         if (r->cvec[w[index+1]-'a']!=NULL) {
34             /* POPWORD(w,index+1) \in dom(t) */
35             // skip
36         }
37         else{
38             // t:= (POPWORD(w,index+1) -> 0)
39             // there is not exist the word in this dict
40             return FALSE;
41         }
42         // recursive call
43         return doCheckword(r->cvec[w[index+1]-'a'],w , index +1);
44     }
45     else{
46         // index = |w|
47         // return b:= (t(w) = 1)
48         return r->eow;
49     }
50 }
51
52 void doDelword(const Dict r, const word w, int index) {
53     if (w[index]!='\0') {
54         // the word is not ended
55         if (r->cvec[w[index+1]-'a']!=NULL) {
56             /* POPWORD(w,index+1) \in dom(t) */
57             // recursive call
58             doDelword(r->cvec[w[index+1]-'a'],w, index+1);
59         }
60         else{
61             // t:= (POPWORD(w,index+1) -> 0)
62             // there is not exist the word in this dict
63             // nothing to delete
64             // skip;
65             return;
66         }
67     }
68     else{
69         // index = |w|
70         // return t:= t: w-> 0
71         r->eow = FALSE;
72     }

```

```

73 }
74
75 void addword (const Dict r, const word w){
76     doAddword(r, w,0);
77 }
78 bool checkword (const Dict r, const word w){
79     return doCheckword(r, w,0);
80 }
81 void delword (const Dict r, const word w){
82     doDelword(r,w, 0);
83 }
84 void barf(char *s){
85     fprintf(stderr, "%s\n",s);
86     exit(1);
87 }

```