Θέματα Σχεδιασμού Γλωσσών Προγραμματισμού

Katsushika Hokusai, *In the Hollow of a Wave off the Coast at Kanagawa, 1827*

Κωστής Σαγώνας <kostis@cs.ntua.gr>

Εν αρχή ην... η χακεριά

- Έστω ότι θέλουμε να βρούμε κάποια πληροφορία για τους χρήστες ενός μηχανήματος Unix από το αρχείο /etc/passwd το οποίο έχει εγγραφές (πιθανώς κάποιες εγγραφές είναι ανενεργές - σχόλια #) της μορφής:

mailman:*:78:78:Mailman user:/var/empty:/usr/bin/false

Βασικά πρέπει να κάνουμε τα εξής

1. Να κοιτάξουμε μόνο τις γραμμές που δεν είναι σχόλια
2. Από αυτές να απομονώσουμε το όνομα χρήστη (username), το home directory, και το shell του κάθε χρήστη
3. Για να βρούμε ποιοι όντως είναι χρήστες, κοιτάμε κατά πόσο το shell τους υπάρχει ως (εκτελέσιμο) πρόγραμμα

Χακεριά στη C++

- Κύριο πρόγραμμα
  - Δηλώσεις αρχείων επικεφαλίδων
  - Δηλώσεις μεταβλητών

```
#include <fstream>
#include <iostream>
#include <string>

using namespace std;
int main() {
  ifstream infile;
  char newrecord[256];
```


Χακεριά στη C++

- Άνοιγμα αρχείου
  - Έλεγχος σφάλματος

```
  infile.open("/etc/passwd");
  if (!infile) {
    cout << "Error opening /etc/passwd.\n";
    exit(-1);
  }
```


Χακεριά στη C++

- Κύρια επανάληψη

```
  infile.getline(newrecord,256);
  while (!infile.fail()) {
    // -- Make array into a String
    String urecord(newrecord);
    if (urecord.find("#") == string::npos) {
       // Process an entry
       ...
    }
    infile.getline(newrecord,256);
  }
  return 0; // Done
}
```


Χακεριά στη C++

- Κύριο σώμα επανάληψης

```
string::size_type name_index = record.find(":");
string::size_type shell_index = record.find_last_of(":");
string::size_type home_index = record.find_last_of(":",
shell_index);
string name = record.substr(0,name_index);
string home = record.substr(home_index+1, shell_index-1);
string shell = record.substr(shell_index + 1);
// For Part 3, check to see if ushell is a valid file
// and if so, execute the next line; otherwise, do nothing.
cout << name << "\t" << home << "\t" << shell << endl;
```

Χακεριά στη Java

- Είναι παρόμοια με τη C++
  - Αλλά η γλώσσα έρχεται με μια καλή βιβλιοθήκη (StringTokenizer)

```
StringTokenizer st = new
StringTokenizer(record, ":");
String username = st.nextToken();
st.nextToken(); // Don't care about these
st.nextToken();
st.nextToken();
st.nextToken();
String home = st.nextToken();
String shell = st.nextToken();
```


Χακεριά σε Perl

```
#!/usr/bin/perl

open PASSWD, "<", "/etc/passwd" ||
      die "Could not open /etc/passwd: $!";
while (<PASSWD>) {
  if (m/^[^#]/) { # Skip comments
    my @fields = split(/:/, $_); # Split on :
    if (-x $fields[6]) {
      print STDOUT "$fields[0]\t$fields[5]\t$fields[6]\n";
    }
  }
}
close (PASSWD);;
```


Χακεριά σε Awk (με χρήση cat)

```
% cat passwd | awk –F: '/[^#]/{ print "$1\t$6\t$7"}'
```


Κάποια ερωτήματα σχεδιασμού γλωσσών

- Τι θέλουμε να είναι ενσωματωμένο (built-in) στη γλώσσα;
- Πόση προσπάθεια χρειάζεται για να γράψουμε ένα τυπικό πρόγραμμα;
- Πως είναι ο κύκλος ανάπτυξης του προγράμματος;
- Ποια είναι τα ενδεχόμενα σφάλματα λογισμικού;
- Πόσο εύκολη είναι η ανάγνωση/κατανόηση του προγράμματος;
- Πόσο εύκολη είναι η συνεργασία μ' ένα άλλο πρόγραμμα;
- Τι υποστήριξη υπάρχει από πλευράς βιβλιοθηκών;

Επίδοση σε ταχύτητα

- Το πρόγραμμα σε C++ είναι περίπου τρεις φορές πιο γρήγορο από ότι σε Perl
  - Όμως η ταχύτητα και των δύο προγραμμάτων είναι ικανοποιητική (για την προβλεπόμενη χρήση τους)
  - Σε τελική ανάλυση, η ταχύτητα εκτέλεσης έχει περισσότερο να κάνει με το χρόνο που χρειάζεται να διαβάσουμε το αρχείο (από το δίσκο) παρά με τη γλώσσα
- Είναι αυτός λόγος για να γράψουμε X φορές περισσότερο κώδικα;

Επιτυχής σχεδιασμός γλωσσών

- Παίρνει υπόψη του τα χαρακτηριστικά των εφαρμογών
  - **C:**
    - προγραμματισμός συστήματος
    - δυνατότητα παρέμβασης σε πολύ χαμηλό επίπεδο
  - **Lisp, Prolog:** συμβολικός υπολογισμός (symbolic computation)
  - **Erlang:**
    - εφαρμογές ταυτοχρονισμού με απαιτήσεις για αδιάκοπη λειτουργία
  - **Perl, Python:** επεξεργασία αρχείων χαρακτήρων
  - **Java, C#:** εφαρμογές διαδικτύου
  - **Javascript:** light-weight client-side προγραμματισμός
  - **SQL:** εφαρμογές βάσεων δεδομένων

Στοιχεία σχεδιασμού γλωσσών

- **Το κλειδί της επιτυχίας:**
  ευκολία επίλυσης κάποιου συνόλου εφαρμογών
- Οι συγκεκριμένες εφαρμογές βοηθούν τους σχεδιαστές να επικεντρώσουν την προσοχή τους σε συγκεκριμένα χαρακτηριστικά και να έχουν σαφώς προσδιορισμένα κριτήρια για τις αποφάσεις τους
- Ένα από τα βασικότερα συστατικά σχεδιασμού γλωσσών και συγχρόνως μια από τις πιο δύσκολες αποφάσεις είναι το **ποια στοιχεία θα μείνουν εκτός της γλώσσας**!

Στοιχεία σχεδιασμού γλωσσών

- Αφηρημένο υπολογιστικό μοντέλο / μηχανή όπως αυτό παρουσιάζεται στον προγραμματιστή
  - **Fortran:** Πίνακες, αριθμοί κινητής υποδιαστολής, κ.λπ.
  - **C:** Μοντέλο μηχανής υπολογιστή, διευθυνσιοδότηση κατά bytes
  - **Lisp:** Λίστες, συναρτήσεις, αυτόματη διαχείριση μνήμης
  - **Smalltalk:** Αντικείμενα και μέθοδοι, επικοινωνία με μηνύματα
  - **Java:** Αντικείμενα, ενδοσκόπηση (reflection), ασφάλεια, JVM
  - Άλλες; Ιστοσελίδες, βάσεις δεδομένων
- Θεωρητική θεμελίωση
  - Τυπικές γλώσσες, λ-λογισμός, Θεωρία Τύπων, Σημασιολογία

Θέματα Σχεδιασμού Γλωσσών
Σύνταξη και Σημασιολογία

Σύνταξη και Σημασιολογία

- **Σύνταξη γλωσσών προγραμματισμού:** πως δείχνουν τα προγράμματα στο χρήστη, τι μορφή και τι δομή έχουν
  - Η σύνταξη συνήθως ορίζεται με χρήση κάποιας τυπικής γραμματικής
- **Σημασιολογία γλωσσών προγραμματισμού:** τι κάνουν τα προγράμματα, ποια (ακριβώς) είναι η συμπεριφορά τους
  - Η σημασιολογία είναι πιο δύσκολη να ορισθεί από τη σύνταξη
  - Υπάρχουν διάφοροι τρόποι ορισμού της σημασιολογίας

Σύνταξη και Σημασιολογία: Παραδείγματα

- Φράση λεκτικά λάθος
  οπα πάς οπα χύςέ φαγ επα χιάφα κή
- Φράση λεκτικά ορθή αλλά συντακτικά λάθος
  ο παπάς ο φακή έφαγε παχιά παχύς
- Φράση συντακτικά (και λεκτικά) ορθή αλλά σημασιολογικά λάθος
  ο παπάς ο παχιά έφαγε παχύς φακή
- Φράση συντακτικά και σημασιολογικά ορθή
  ο παπάς ο παχύς έφαγε παχιά φακή

Λεκτική ανάλυση γλωσσών προγραμματισμού

- Η λεκτική ανάλυση δεν είναι τετριμμένο πρόβλημα γιατί οι γλώσσες προγραμματισμού συνήθως είναι πιο περίπλοκες λεκτικά από τα Ελληνικά

```
*p->f++ = -.12345e-6
```

- Άλλο παράδειγμα

```
float x, y, z;
float * p = &z;

x = y/*p;
```

- Τι συμβαίνει σε αυτή την περίπτωση;
  `/*` στη C είναι ο συνδυασμός με τον οποίο αρχίζει ένα σχόλιο

Λεκτικές συμβάσεις στις γλώσσες

- Η C είναι γλώσσα *ελεύθερης μορφής* (*free-form*) όπου τα κενά απλώς διαχωρίζουν τις λεκτικές μονάδες (tokens). Ποια από τα παρακάτω είναι τα ίδια?

```
1+2
1 + 2
```

```
foo  bar
foobar
```

```
return  this
returnthis
```

- Τα κενά θεωρούνται σημαντικά σε μερικές γλώσσες. Για παράδειγμα, η γλώσσα Python χρησιμοποιεί στοίχιση (indentation) για ομαδοποίηση, οπότε τα παρακάτω είναι διαφορετικά:

```
if x < 3:
  y = 2
  z = 1
```

```
if x < 3:
  y = 2
z = 1
```


Δήλωση λεκτικών μονάδων

- Πως δηλώνονται οι λεκτικές μονάδες;
  - Λέξεις κλειδιά (keywords) - μέσω συμβολοσειρών (strings)
  - Πως ορίζονται τα ονόματα των μεταβλητών (identifiers);
  - Πως ορίζονται οι αριθμοί κινητής υποδιαστολής;
- Κανονικές εκφράσεις (regular expressions)
  - Ένας εύχρηστος τρόπος να ορίσουμε σειρές από χαρακτήρες
  - Χρησιμοποιούνται ευρέως: grep, awk, perl, κ.λπ..
- Παραδείγματα:
  - **'0' - ταιριάζει μόνο με το χαρακτήρα 0 (μηδέν)**
  - **'0'|'1' - ταιριάζει με μηδέν ή με ένα**
  - **'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' - ταιριάζει με ψηφία**
  - **[0-9] - το ίδιο με το παραπάνω αλλά σε πιο συμπαγή μορφή**
  - **[0-9]* - σειρά από ψηφία (πιθανώς κενή)**

Θέματα σχεδιασμού λεκτικών μονάδων

- Ακέραιοι αριθμοί (π.χ. **10**)
  - Οι αρνητικοί ακέραιοι είναι μία λεκτική μονάδα ή όχι;
- Χαρακτήρες (π.χ. **'a'**)
  - Πως αναπαρίστανται οι μη εκτυπώσιμοι χαρακτήρες ή το **'** ;
- Αριθμοί κινητής υποδιαστολής (π.χ. **3.14e-5**)
  - Τι συμβαίνει με αριθμούς που δεν αναπαρίστανται κατά IEEE;
- Συμβολοσειρών (π.χ. **"hello world"**)
  - Πώς αναπαρίσταται ο χαρακτήρας **"**;

Σύνταξη γλωσσών προγραμματισμού

- Συγκεκριμένη σύνταξη (concrete syntax)
  - Ποια είναι τα ακριβή σύμβολα (χαρακτήρες ή άλλες αναπαραστάσεις) με χρήση των οποίων γράφεται το πρόγραμμα;
- Αφηρημένη σύνταξη (abstract syntax)
  - Μια αφαίρεση της συγκεκριμένης σύνταξης η οποία είναι η λογική αναπαράσταση της γλώσσας
  - Πιο συγκεκριμένα:
    - Μη διφορούμενη: δεν εγείρεται θέμα για την ερμηνεία των προγραμμάτων
    - Πιο κοντά στο "τι σημαίνει" το πρόγραμμα
  - Συχνά κάτι που χρησιμοποιείται από το μεταγλωττιστή (compiler) ή το διερμηνέα (interpreter) της γλώσσας

Συγκεκριμένη και Αφηρημένη Σύνταξη

- Παραδείγματα:

```
while i < N do
begin
   i := i + 1
end
```
Pascal

```
while (i < N)
{
   i = i + 1
}
```
C/C++

- Τα παραπάνω προγράμματα κάνουν το ίδιο πράγμα
- Η συγκεκριμένη σύνταξή τους διαφέρει σε αρκετά σημεία
- Η αφηρημένη τους σύνταξη είναι η ίδια

```
while i < N
   i = i + 1
```
Πιθανή αφηρημένη σύνταξη

- Η σημασιολογία τους είναι η ίδια

Οι επιλογές σχεδιασμού της γλώσσας C

Η ιστορία και οι επιλογές της C

- Αναπτύχθηκε μεταξύ 1969 και 1973 μαζί με το Unix από τον Dennis Ritchie

- Σχεδιάστηκε για προγραμματισμό συστήματος
  - Λειτουργικά συστήματα
  - Εργαλεία υποστήριξης / μεταγλωττιστές
  - Φίλτρα / Ενσωματωμένα συστήματα
- Η μηχανή ανάπτυξης (DEC PDP-11) είχε
  - 24K bytes of memory - of which 12K for the OS
- Πολλά στοιχεία της C λόγω έλλειψης μνήμης
  - Μεταγλωττιστής ενός περάσματος
  - Συναρτήσεις ενός επιπέδου (without nesting)

Μετατροπές (Conversions)

- Η C ορίζει κάποιες αυτόματες μετατροπές :
  - Ένας `char` μπορεί να χρησιμοποιηθεί ως `int`
  - Η αριθμητική κινητής υποδιαστολής πάντα γίνεται με `doubles`. Οι `floats` προάγονται αυτόματα σε `doubles`
  - Οι `int` και `char` μπορούν να μετατραπούν σε `float` ή σε `double` και αντίστροφα. Το αποτέλεσμα είναι απροσδιόριστο εάν μπορεί να υπερχειλίσει.
  - Η πρόσθεση ενός αριθμού (`int`) σε έναν δείκτη (pointer) δίνει αποτέλεσμα ένα δείκτη
  - Η αφαίρεση δύο δεικτών σε αντικείμενα του ίδιου (πάνω-κάτω) τύπου δίνει ως αποτέλεσμα έναν ακέραιο (`int`)

Δηλωτές (Declarators) της C

- Οι δηλώσεις έχουν τη μορφή:

basic type

```
static unsigned int (*f[10])(int, char*)[10];
```

specifiers declarator

- Declarator's notation matches that of an expression: use it to return the basic type
- Συντακτικά ίσως το χειρότερο χαρακτηριστικό της C: διότι συνδυάζει τόσο prefix (pointers) όσο και postfix τελεστές (arrays, functions)

Δηλώσεις συναρτήσεων (προ ANSI C)

Είχαν τη γενική μορφή

type-specifier declarator (*parameter-list*)

type-decl-list

{

declaration-list

statement-list

}

Για παράδειγμα

```
int max(a, b, c)
int a, b, c;
{
  int m;
  m = (a > b) ? a : b ;
  return m > c ? m : c ;
}
```


Επιλογές σχεδιασμού της C

- Οι πρώτοι compilers της C δεν έλεγχαν τον αριθμό και τον τύπο των ορισμάτων των συναρτήσεων
- Η μεγαλύτερη αλλαγή που έγινε στη C όταν αυτή έγινε ANSI standard ήταν η απαίτηση οι συναρτήσεις να ορίζουν τους τύπους των παραμέτρων τους

```
int f();

int f(a, b, c)
int a, b;
double c;
{
}
```

Παλιό στυλ δηλώσεων

```
int f(int, int, double);

int f(int a, int b, double c)
{
}
```

Νέο στυλ δηλώσεων

Δηλώσεις δεδομένων στη C

- Έχουν τη μορφή

type-specifier init-declarator-list **;**

declarator optional-initializer

- Οι αρχικοποιητές (initializers) μπορεί να είναι σταθερές, ή σταθερές εκφράσεις που διαχωρίζονται με κόμματα και είναι κλεισμένες σε αγκύλες
- Παραδείγματα:
  - `int a;`
  - `struct { int x; int y; } b = { 1, 2 };`
  - `float a, *b, c = 3.14;`

Κανόνες εμβέλειας (Scope rules)

Δύο είδη εμβέλειας στη C:

1. Λεκτική εμβέλεια
   - Βασικά, το μέρος του προγράμματος όπου δεν υπάρχουν λάθη αδήλωτων μεταβλητών ("undeclared identifier" errors)
2. Εμβέλεια των external identifiers
   - Όταν δύο identifiers σε διαφορετικά αρχεία αναφέρονται στο ίδιο αντικείμενο.
   - Π.χ., μια συνάρτηση που είναι ορισμένη σε ένα αρχείο καλείται από μια συνάρτηση σε ένα άλλο αρχείο.

Λεκτική εμβέλεια

- Εκτείνεται από το σημείο ορισμού στο αντίστοιχο } ή στο τέλος του αρχείου

```
int a;

int foo()
{
  int b;
  if (a == 0) {
    printf("A was 0");
  }
  b = a; /* OK */
}

int bar()
{
  a = 3; /* OK */
  b = 2; /* Error: b out of scope */
}
```


Εμβέλεια των external δηλώσεων

file1.c

```
int foo()
{
   bar(); /* Error */
}

int bar()
{
   foo(); /* OK */
}
```

file2.c

```
int boo()
{
   foo(); /* Error */
}

extern int foo();

int baz()
{
   foo(); /* OK */
}
```


Ο C Preprocessor

- Έρχεται σε αντίθεση με την ελευθέρας μορφής φύση της C: οι γραμμές του προεπεξεργαστή *πρέπει να αρχίζουν με #*
- Το κείμενο του προγράμματος περνάει μέσα από τον προεπεξεργαστή πριν εισαχθεί στο μεταγλωττιστή

Αντικατάσταση ενός identifier:

`# define` *identifier token-string*

Αντικατάσταση μιας γραμμής με τα περιεχόμενα ενός αρχείου:

`# include "`*filename*`"`

Βιβλιοθήκες της C

Header file	Περιγραφή	Τυπική χρήση
<assert.h>	Generate runtime errors	assert(a>0)
<ctype.h>	Character classes	isalpha(c)
<errno.h>	System error numbers	errno
<float.h>	Floating-point constants	FLT_MAX
<limits.h>	Integer constants	INT_MAX
<locale.h>	Internationalization	setlocale(…)
<math.h>	Math functions	sin(x)
<setjmp.h>	Non-local goto	setjmp(jb)
<signal.h>	Signal handling	signal(SIGINT,&f)
<stdarg.h>	Variable-length arguments	va_start(ap, st)
<stddef.h>	Some standard types	size t
<stdio.h>	File I/O, printing	printf("%d", i)
<stdlib.h>	Miscellaneous functions	malloc(1024)
<string.h>	String manipulation	strcmp(s1, s2)

Σχεδιασμός Γλωσσών Προγραμματισμού

Language design is library design.

-- Bjarne Stroustrup

- Τα περισσότερα προγράμματα φτιάχνονται από συναρμολογούμενα κομμάτια
- Μια από τις κύριες δυσκολίες στο σχεδιασμό γλωσσών είναι το πως τα κομμάτια αυτά μπορούν να βρεθούν μαζί και να συνυπάρξουν *αρμονικά* και *σωστά*