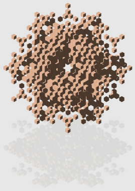
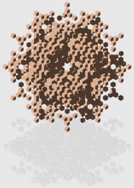


C Language Programming

A short course about C (actually about Lex and YACC too ☺)

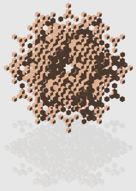
by Eidon (Eidon at tutanota.com), 27 June 2018 (version 1.0)

A Short Course on C Language Programming



- Eidon@tutanota.com
- Course slides at `https://github.com/Eidonko/ccourse`
- Course software at `https://github.com/Eidonko`

Objectives



Provide the audience with a set of

- ideas,
- methods,
- and know-how resulting from several years of design and development with the C language in an engineering environment.
- Problem solving with C: syntactical analysis.

Purpose and structure

- An introduction to C
- The `FILE` methodology. Classes. Data hiding. Examples (class assoc, objalloc, vf. . .)
- Literate programming. The cweb tools
- The GNU Autotools
- Other classes for embedded systems (class tom, the “art” framework)
- Linguistic support using C, Lex and YACC. Examples (the icgi interpreter, class FN, PvmLinda, the Ariel language...)
- Exercise sessions, case studies, project works

Advanced C Language for Engineering

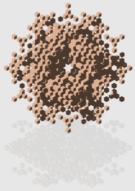
Exam: a project work (coding plus documentation) to be sent to me by the end of the year.

“The” problem

Teaching C to an audience with different backgrounds and different expectations is very difficult.

Tuning the course is difficult.

Exercise sessions are particularly problematic.



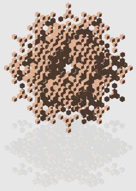
Solutions

Ideally, the course should be *ad personam*. Strong interaction with the teacher is suggested. Just mail me with your questions.

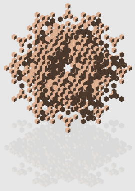
Practically speaking, exercises will be organized in 2 levels: for those with no practical experience vs. for those with practical experience.

- level A : for instance, learning how to log in a UNIX system, how to edit a C program, how to compile, and execute it
- level B : designing “improved” versions, e.g., code optimisations; writing test programs using the C classes at `https://github.com/Eidonko`, and so forth.

For experienced people, exercise sessions can be devoted to the design of the solution of the project works.

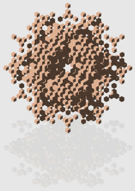


Introduction to C: Structure



- Introduction
- First examples
- Fundamental data types, derived data types
- Derived types: pointers and vectors
- Functions
- Structures and unions
- Data hiding

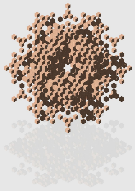
Introduction to C



- C and UNIX : success stories.
- C: developed by Ritchie to port UNIX from a PDP-7 to a PDP-11
- System programming language
- Most of the UNIX system is in C

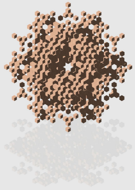
Introduction to C

- C is simple:
  - A small number of simple and powerful instructions
  - dealing with characters, numbers, addresses
  - No language facilities for handling complex objects (strings, lists...)
  - No language facilities for the I/O, for memory allocation...
- Those facilities are available as standard (or user-defined) libraries of functions

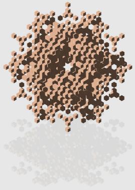


Introduction to C

- C is small:
  - Easy to describe, easy to learn
  - The C compiler is considerably simple, compact-sized, and easy to write
  - The C data types and control structures often have a direct equivalent in the machine language of many computers. This happens because C was modelled after the machine language of the PDP 11.
  - Hence, while in general the translation from a high level language instruction to machine language is in general one-to-many, the translation from C to machine language is **one-to-few**.
  - A very simple run-time system: for instance (PDP-11) :
    - routines for 32-bit or 64-bit $\star$ and $/$.
    - management of subroutine entry and exit.



Introduction to C



C is portable. . .

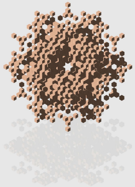
- but we need to follow some portability rules
- types have no well-defined size (e.g., in Python we have `numpy.int32`, in C we don't have an a-priori knowledge of the size of an `int`)
- we can use symbolic constants for this
- large set of standard libraries for I/O, string manipulation, memory allocation. . .

Introduction to C

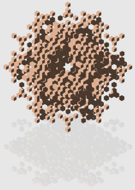
BCPL (typeless language by M. Richards) $\Rightarrow$ B Language (typeless lang. by K. Thompson) $\Rightarrow$ C Language (non-typeless, non-strongly-typed language) by D. M. Ritchie

Typeless Language : data are organized as multiple instances of the machine word. Everything is allowed. Example: Assembly

Strongly-Typed Language : Nearly every cast from type *A* to type *B* needs to be explicitly requested. Once a variable of a given type has been defined, also its *run-time checking* is defined in order to detect, e.g., overflow, underflow, subscript-out-of-range conditions, and so forth. ecc. Examples: Pascal, Ada.



Introduction to C

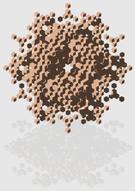


C is not a typeless language: basic types are available

- characters
- integers (in various sizes)
- floating point numbers (in different sizes)
- pointers
- facilities to build complex types from the basic types: arrays, records, “unions”, functions.

C is not a strongly-typed language: it is *permissive* about data casts. Its run-time executive does not cover run-time checking.

Introduction to C



C provides:

- the basic control-flow statements
  - statement grouping
  - selective statements
  - iterative statements
- pointers + *address arithmetic*

Introduction to C: Functions

- Note:
  - only call-by-value
  - recursion is allowed
  - single-level functions that can be compiled separately.
  - variables
    - scope: “block-structured fashion”
    - static vs. automatic

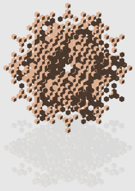
Introduction to C

“The only way to learn a new programming language is by writing programs in it; the first program to write is the same for all languages: print the words `"hello, world"`*.” [KR]*

“What we have to learn to do we learn by doing.” [Aristotle]

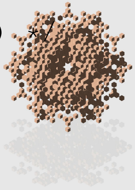
```
main()     /* prints "hello world." */
{          /* and goes to the next line */
    printf("hello world.\n");
}
```

Introduction to C



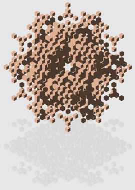
- Log into your Linux workstation
- type the `"hello, world"` program
- compile it
- execute it

Introduction to C: a classic example



```
/* The famous Fahrenheit to Celsius conversion :-) */
/* From the KC book! */
void main() {
    int inf, sup, step;
    float fahr, celsius;

    inf = 0;
    sup = 300;
    step = 20;

    fahr = inf;
    while  (fahr <= sup)  {
           celsius = (5.0/9.0) * (fahr-32.0);
           printf("%4.0f %6.1f\n", fahr, celsius);
           fahr = fahr + step;
    }
}
```

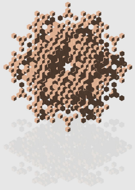
Introduction to C



Adapt the above code so to do the opposite conversion.

Try to answer the following questions:

- why `5.0/9.0` and not `5/9`?
- is `5.0/9` equivalent to `5.0/9.0`?
- is the `.0` really necessary in `fahr-32.0`?



Introduction to C: while loops

```
while ( test ) action
```

`test` is executed. If output is non-zero (“true”) then `action` is executed.

`action` can also be a composite action using curly brackets.

```
for (init; test; incr) action
```

is equivalent to

```
init;
while (test) { action; incr; }
```

Introduction to C

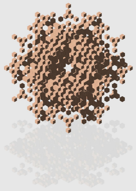
`init`, `test`, `incr`, and `action` are all optional arguments: even

```
for (;;) ;
```

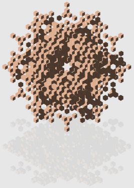
is allowed.

Note: `test` is a *statement*!

Both `while (a=3) ...` and `while (a==3) ...` are valid!



Introduction to C

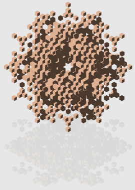


Exercise: modify the previous example so to change the `while` into `for`.

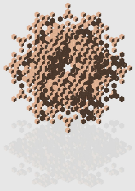
Exercise: modify the previous example so to print the table in inverse order.

Introduction to C: conditional statements

- statement “if”:
  if (condition) action$_1$; else action$_2$;
  (nesting and so forth.)
- “?:”
  (condition)? action$_1$: action$_2$;
- “switch”:
  switch(integer_expr) {
  case value$_1$: op$_1$; break;
  case value$_2$: op$_2$; break;
  ⋮
  case value$_n$: op$_n$; break;
  default: op$_{n+1}$; }



Introduction to C: loops and related statements



- for
- while
- do while
- break;
- continue;
- goto label

Introduction to C: defines and includes

In C, it is possible to define symbolic constants via the `#define` pre-processor directive, and to specify that an external file should be read in at compile time, via the `#include` pre-processor directive.

A few examples follow:

#define	BEGIN	{
#define	END	}
#define	IF	if(
#define	THEN	)
#define	INT32	long
#define	INT32	int
#include	`"file"`	
#include	`<file>`	

Introduction to C: other pre-processor directives

- macro substitution (e.g., `#define TRUE 1`)
- macro with parameters:
  #define $MAX(a, b)$ $((A) > (B)\ ?\ (A) : (B))$
  (Note: #define sqr(x) $= x * x$ is error-prone. Can you tell why?)
- Conditional inclusion: (`#ifdef` and `#ifndef`, `#else` and `#endif`.)

Introduction to C: first functions for standard I/O

- Defined in <stdio.h>
- getchar (e.g. c=getchar()), putchar (e.g. putchar(c)), `EOF`...

Example: copy of input to output — version 1.

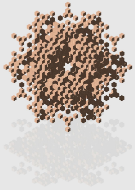
```
#include <stdio.h>
int main() { int c;
 c=getchar();
 while (c != EOF ) {
     putchar (c);
     c=getchar();
 }
}
```

Introduction to C: first functions for standard I/O

Example: copy of input to output — version 2.

```
#include <stdio.h>
main() { int c;

while ((c = getchar()) != EOF )
    putchar (c);
}
```

Note: parentheses around `c = getchar()` are necessary, because of the larger priority of `!=` with respect to `=`.

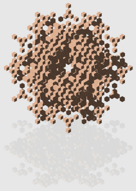


Introduction to C: first functions for standard I/O

Exercise: the just seen program can be easily adapted to work, e.g., as a “word counter” (reports the number of characters, words, and lines in the input), or as a filter to remove blanks or tabs, and so forth.

Input and output can be redirected with $<$ and $>$. Pipelines of programs can be built by chaining the programs with $|$.

Introduction to C: arrays



To declare an array in C one has to use the following syntax:

type name [dimension1][dimension2]..[dimension$_n$] ;

For instance, `int vect[10];`

This is equivalent to e.g. `vect = numpy.array(10, dtype=numpy.int)` in Python.

The index can be any integer expression.

There is no bound check like, e.g., in Ada.

Introduction to C: arrays

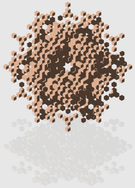
Note: declaring an array means

1. declaring a **pointer**
2. and allocating memory for the pointed objects.

The name of the array is indeed a pointer to the first element of the array. In C lingo, this is written as `vect == &vect[0]`.

Esercise: write a program, called report, that reports the occurrences of each digit char in the input stream. Use an array of ten elements corresponding to the ten decimal digits. Produce an histogram at end-of-input.

Introduction to C: arrays



Exercise: use two programs, one that outputs the ten integer numbers that count the occurrences of each digit char in the input stream, the other one that creates a histogram of its input values. Then use a pipeline to hook the two programs together.

```
report2 | histogram
```

Introduction to C: functions

Functions are named fragments of C code that can accept arguments and return a value of some type.

Functions are, e.g., printf(), getchar(), putchar(), main()...

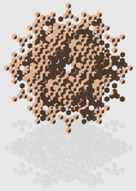
No function hierarchy is allowed. The only “special” function is main(). Functions can reside in a same source file or in more than one.

Introduction to C: functions

Functions have the following structure:

```
[type] name ( [ args ] )
{ [ declarations ] [ instructions ] }
```

```
  int main() {
      int i; int power (int, int);
      for (i=0; i<10; ++i)
          printf("%d %d\n", i, power(2,i));
  }
  int power (int x, int n) { int i, p;
    for (i = p = 1; i <= n; ++i)
       p=p*x;
    return (p);
  }
```



Introduction to C

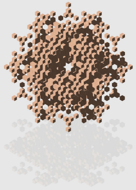
`return` closes the function and returns an output value (default: integer) to the caller.

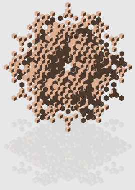
Arguments are passed **by value**: this means that the arguments are copied in temporary variables.

The only way to let a function modify an argument is by passing *the address* of the object to be modified.

Operator `&` returns the address of a variable.

Note that, when you need to pass an array, passing its name is indeed passing a pointer to the array.





Introduction to C: strings

Strings are available in C as arrays of characters. Any sentence enclosed between two `"`'s is an array of characters ending with character `'\0'` (NULL).

For instance, `"hello"` ⇒ 'h', 'e', 'l', 'l', 'o', 0

A very common mistake when learning C:

```
char s1[10] = "Hello ";
char s2[10] = "World";

s1=s2;     /* ERROR */
```

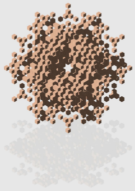
Introduction to C

As strings are arrays, we can easily pass a string to a function by passing its name, which points to its characters:

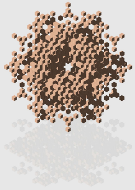
```
char a[] = "Pogo Possum";
printf("Vote %s for president.\n", a);
/* a = &a[0] */
```

Variables defined within a function cannot be “seen” from other functions.

Two main classes of variables: `automatic` vs. `static`. Automatic variables are allocated at function call and deallocated when the function stops. They require initialisation. Static variables are allocated at compile time. They are initialised to zero, and *keep the value we left in them when we get out of the functions where they are defined.*



Introduction to C



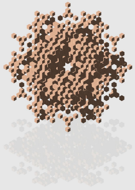
- identifiers have the following structure: <alpha> <alphanum> *
- six basic types: char, short, int, long, float, double.
- qualifiers: unsigned, register
- constants:
  - scientific notation, e.g., `1.5e3` → double
  - postfix notation, e.g., 145L → long
  - prefix notation: '0x44' → unsigned int, hexadecimal, '044' → unsigned int, octal
  - costant char. `'c'` = character c → char

Introduction to C

- special constants, e.g., `\n`, `\t`, `\0`, `\\`, `\"` and so forth $\rightarrow$ char
- “bit patterns”: `\ddd`, `ddd` being an octal number.
- string constants, e.g., `"string"` or `""`.

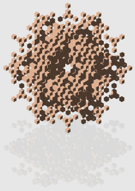
Exercise: write a function that counts the number of character pointed by its string argument (`strlen`).

Introduction to C: declarative instructions



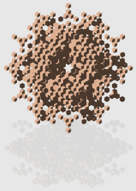
- Declarative instructions take the following form:
  typename list-of-variables;
- A declarative instruction can be followed by an initialisation:
  int n=5; float g=5.4;
- If the variable is an automatic one, then the initialisation is done *each time the variable is allocated* (each time the function in which the variable resides is called).

Introduction to C: declarative instructions



- Non-initialised automatic variables contain random value.
- If the variable is a static one, then the initialisation is done *only once* (at compile time).
- Static variables are always initialised. Default=0.
- *Global, static variables are only visible to the functions in the same source file*: this can be used for data hiding.

Introduction to C: operators



- binary: +, -, $*$, /, %
- unary -
- precedences: +,- $<_{\text{prec}}$ $*$,/,% $<_{\text{prec}}$ -

- `||` $<_{\text{prec}}$
- && $<_{\text{prec}}$
- == and !=, $<_{\text{prec}}$
- $> \; >= \; < \; <=$

Introduction to C: operators

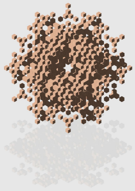
Expressions such as:

$i < lim$ && (c=getchar()) != '
n' && c != `EOF`

do not require extra parentheses: = $<_{\text{prec}}$!=, hence parentheses are required around `c=getchar()`.

Logic clauses are evaluated left-to-right. Evaluation stops when the truth value of a logic expression is ascertained.

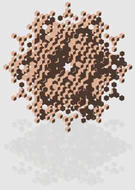
Introduction to C: derived types



The following operators define complex types derived from the basic types of C:

- `*` operator pointer-to,
- `[ ]` operator vector-of,
- `( )` operator pointer-to-function.

Introduction to C: derived types



- `char *p;` : `p` is of type “pointer-to-chars”
- `float v[10];` : `v` is a vector, i.e., a pointer to the beginning of a memory area allocated by the system and consisting of 10 floats, i.e., `v[0]`, . . . , `v[9]`.
- `int getX();` : `getX` is a pointer to a function returning an int.

Introduction to C: derived types

Operator `[]` has higher priority with respect to operator `*`. This means that

```
int *v[10]
```

means that `v` is a vector of ten pointers-to-int. Parentheses are necessary to change the meaning:

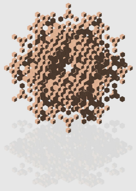
```
int (*v)[10]
```

means that `v` is a pointer to a vector of ten integers.

What's the difference in terms of sizes?

Introduction to C: derived types

1. `int *pi;` : pointer to integer;
2. `char **argv;` : pointer to pointer-to-char;
3. `float *(fvp)[5];` : pointer to vectors-of-5-floats
4. `long (*fp)(int);` : pointer to function reading an int and returning a long.



Introduction to C: derived types

The address-of (&) operator returns the address of a variable.

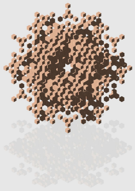
```
char c; char *pc;   pc = &pc;
```

Operator *, applied to a pointer, returns the pointed object.

```
char c1 = 'a';   char *p = &c1;
char c2 = *p2;   /* c2 == 'a' */
```

```
void func(char *s) { printf("bye %s\n", s) }

main() { void (*pfunc)(char*) = func;
         *(pfunc)("hello");
    }
```

Introduction to C: derived types

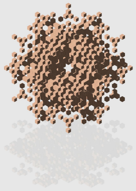


Pointers solve the problem of the lack of “call-by-reference” in C functions. For instance, in order to realize a function that swaps its argument, one may use the following strategy:

```
int swap(int *a, int *b) { int t;
    t = *a,    *a = *b,     *b = t;
}
```

The caller then needs to specify the addresses of the operands (s)he wants to swap, as in `swap(&i, &j)`.

Introduction to C: derived types



Arrays and pointers are strictly related to each other: In particular, if `int a[10];` then a $\equiv$ &a[0], a+1 $\equiv$ &a[1] and so forth. In other words,

$$\texttt{*(a+i)} \equiv \texttt{a[i]}$$

Any indexed array is equivalent to a pointer plus some offset, and vice-versa.

Big difference: the array is a constant, i.e., it can never appear on the left hand side of the = sign, as in

$$a = pa$$

or in

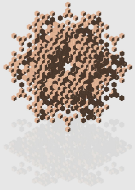
$$a++$$

Introduction to C: derived types

When passing an array to a function we are indeed passing the address of its first element; this address is copied (call-by-value) in a temporary variable. This variable may be a pointer.

Example:

```
char s[7] = "hello!";  /* s is an array */
int  i = strlen(s);

int strlen (char *x) { /* x is a pointer */
   int n;
   for (n=0; *x; x++)  /* hence, can be modified */
      n++;
   return (n);
}
```



Introduction to C: address arithmetics

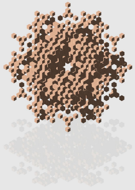
If p is a pointer, p++ lets p point to the next item. It is the language that takes care of moving p of the right amount of memory.

For instance,

```
                 int  ≡ 2 byte
int *p;          &p = 1222    &(p++) = 1224
              double  ≡ 8 byte
double *p;       &p = 5644    &(p++) = 5660
                  and so forth
```

In other words: if p is a pointer to an object of type t, then p+n points n objects further and p-n points n objects before. The actual size of the object doesn't matter.

Introduction to C: type casts



Implicit type casts occur when, in expressions, operands belong to different types. Conversions obey the following rules:

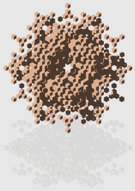
- `char` and `short` → `int` , `float` → `double`
- if an operand is `double` , the other becomes `double` and the result is `double`
- otherwise, if an operand is `long` , the other becomes `long` and the result is a `long`
- otherwise, if an operand is unsigned, the other one becomes unsigned and the result is unsigned.
- otherwise, operands and result are `int` .

Introduction to C: conversions

Converting a string of digits into a number, and vice-versa, requires specific support. Functions are available for this, e.g., `atoi()`:

```
int atoi(char s[]) { int i, n;
     n = 0;
     for (i=0; s[i]>='0' && s[i]<='9'; ++i)
         n=10*n + s[i] -'0';
     return (n);
}
```

Note how expression `s[i] - '0'` converts numeric character `s[i]` into the digit it represents.



Introduction to C: conversions

The following function can be used to convert an uppercase character into its lowercase counterpart.

```
lower(c)
int c;
{
  if (c >='A' && c <= 'Z')
     return (c + 'a' - 'A');
  else
     return (c);
}
```

Note: this program only works for code tables in which `'a'` follows `'A'`. This is true with ASCII and false with other codes (e.g., EBCDIC.)

Introduction to C: conversions

Problem: the C language does not specify anything about the sign of a `char` . Hence, when a `char` is converted to an `int` , the result may lead to a negative number.

A case of machine-dependance: for instance, on the PDP-11, a `char` is signed, hence if a character has the MSB set to one it becomes a negative integer when cast to `int` . On other platforms a `char` is unsigned.

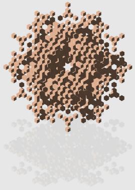
Introduction to C: conversions

The most common problem related to this is due to the value of the constant character `EOF`, i.e., -1. The following C code:

```
char c;

c=getchar();
if (c == EOF) .....
```

is faulty when executed on platforms in which `char` is `unsigned`, because by the conversion rules, -1 becomes 255 and the `if` is never true.

The problem is solved when we declare `int c`.



Introduction to C: conversions and types

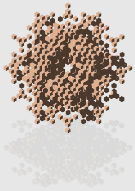
Sign of `EOF` on some platforms:

SUN3	HP9K	AIX	T800	MSDOS	VM
-1	-1	255	255	-1	255

`sizeof`'s on those platforms:

	SUN3	HP9K	AIX	T800	MSDOS	VM
short	2	2	2	2	2	2
int	4	4	4	4	2	2
long	4	4	4	4	4	4
float	4	4	4	4	4	4
double	8	8	8	8	8	8

Introduction to C: conversions



While in general, in C, zero means false and non-zero means true, the truth value in expressions such as

```
isdigit = c >= '0' && c <= '9';
```

is 1 for true and 0 for false.

Introduction to C: conversions

Explicit cast

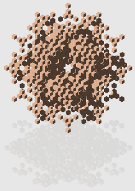
The cast operator changes the type of an object. For instance, the following expression:

```
celsius = ( (double)5 /9) * (fahr-32.0);
```

is equivalent to

```
celsius = (5.0/9.0) * (fahr-32.0);
```

Casting is invoked by specifying a type between parentheses.



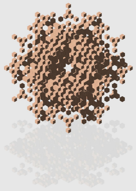
Introduction to C: increment/decrement operators

```
IN C:                  in Assembly:
int i = 0;             DATA segment
i++;          ⇒        i DB 0
                       ⋮
                       INC i
```

Operators such as ++ or −− may have a direct counterpart in the machine language.

- Operator ++ increments the contents of a variable. $x = n++$ is *not* equivalent to $x = ++n$.
- Operator −− decrements the contents of a variable. $x = n--$ is *not* equivalent to $x = --n$.

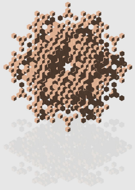
Introduction to C: increment/decrement operators



Esercise: function `purge(char s[], int c)` removes all occurrences of `c` from `s[]`.

Esercise: functions `strcpy()` and `strcat()`.

Introduction to C: increment/decrement of pointers

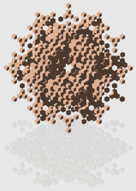


```
int strlen (char *p) { int i=0;
                       while (*p++) i++;
                       return i;
    }
```

`*p` returns the char pointed to by `p`. `*p++` does the same, but increments the pointer afterwards.

When does the while loop ends?

Introduction to C: increment/decrement of pointers



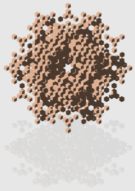
This is an elternative way to write function `strlen`:

```
int strlen (char *p) { char *q = p;
                       while (*p++) ;
                       return q - p - 1;
    }
```

Introduction to C: bitwise operators

The following six operands can be applied to any integer expression:

&	: bitwise AND
|	: bitwise OR
ˆ	: bitwise exclusive OR
<<	: left shift
>>	: right shift
˜	: unary complement.



Introduction to C: bitwise operators

Bitwise AND can be used to set up “masks”, e.g.,

```
c = n & 0177;
```

which zeroes all the bits from bit 8 onward. Bitwise OR sets bits: `x = x | MASK` sets to 1 the bits that are set in MASK. $<<$ and $>>$ are respectively arithmetical shifts to the left and to the right ($\rightarrow$ multiplication resp. division by 2).

Operator ˜ turns each 1 into 0 and vice-versa; it is used in expressions like, e.g., `x & ˜077`, which zeroes the last bits of `x`. Note that this is independent of the actual size of `x`, and therefore it is “more portable” than, e.g., `x & 0177700`.

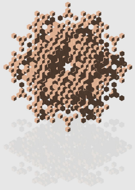
Introduction to C: bitwise operators

```
unsigned int
MIDbit (unsigned int x,
        unsigned int start,
        unsigned int num)
{
return((x >> (start+1-num)) & ~(~0 << num));
}
```

For instance, MIDbit(x, 4, 3) returns the three bits at position 4, 3, and 2. x $>>$ (p+1-n) shifts the bits of interest on the rightmost position in the word.

~ `0` is 11...1; n shifts to left lead to 11...1 $\underbrace{00..0}_{\text{n zeroes}}$;

complementing this value we get 00...0 $\underbrace{11..1}_{\text{n ones}}$.



Introduction to C: assignment operator

In C lingo, the left part of an assignment is called an lvalue, and the right part is called an rvalue:

lvalue = rvalue;

Expressions such as

i = i + 2;

require two memory accesses to the same memory location, which is referenced once in the lvalue and once in the rvalue. In C, an equivalent expression is:

i += 2;

which does only one access

Introduction to C: assignment operator

Binary operators

$$+ - * / \% << >> \& \,\hat{}\, \text{ and } |$$

can use notation e_1 op= e_2 instead of $e_1 = (e_1)$ op (e_2).

Note that `x *= y+1` is equivalent to `x = x*(y+1)`.

An example follows:

```
int bitcount(unsigned n) { int b;
   for (b=0; n != 0; n >>= 1)
       if (n & 01)
           b++;
   return (b);
}
```

Introduction to C: assignment operator

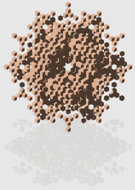
Let me consider the following two programs:

```
int main() { int n;
 n = 0;
 n = n+2+n*n;
}
```

```
int main() { int n;
 n = 0;
 n += 2+n*n;
}
```

When compiled with option `-Qproduce .s` (on an old Sun3 workstation lol), the output Assembly files differ in the following lines:

Introduction to C: assignment operator



```
16,19c16,17
<         movl      a6@(-0x4),d1
<         addql     #0x2,d1
<         addl      d1,d0
<         movl      d0,a6@(-0x4)
---
>         addql     #0x2,d0
>         addl      d0,a6@(-0x4)
```

(No differences with `-O`).

Introduction to C: assignment operator

The assignment form **op=** is

- more efficient
- more concise
- closer to natural language: “add 2 to i” is much more natural than “take i, add 2 to it, and write the result back to i.”

Note that the assignment operator returns the rvalue.

Pointers and arrays

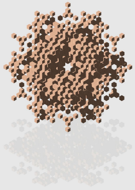
Function `strcpy(char *a, char *b);`

Assumption: NULL-terminated strings. Note that NULL is (int)0, i.e., “false”

Function `strcmp(char *s, char *t)`: returns a negative number if $s < t$, 0 if $s == t$, a positiven number if $s > t$.

```
strcmp(char *s, char *t) {
   for ( ; *s == *t; s++, t++)
       if (*s == '\0') return (0);
   return (*s - *t);
}
```

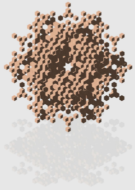
Pointers and arrays



Is this `#define` OK?

```
#define STRCPY(a,b) while (*a++ = *b++) ;
```

Pointers and arrays



To declare multidimensional arrays one declares arrays of arrays. For instance,

```
static int day_of_month[2][13] = {
 { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
 { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} };
```

It is now possible to compute the day-of-year for a given date, e.g., March 1st, 1991: 60th day of year 1991.

Pointers and arrays

```
int day_of_year(int day, int month, int year)
{ int i, leap;

  leap = year%4 == 0 && year%100 != 0
          || year%400 == 0;
  for (i=1; i<month; i++)
      day += day_in_month[leap][i];
  return (day);
}
```

Pointers and arrays

- int v[i][j]   vs. int v[i,j]
- entries are stored "by row": the rightmost index is the one that varies the most when entries are referenced in the order they are stored.
- initialisation: using curly brackets.
- when passing a bidimensional array to a function, it is mandatory that the number of columns be specified. For instance:
  `f(int day_in_month[2][13])`, or
  `f(int day_in_month[][13])`, or
  `f(int (*day_in_month)[13])`,
  i.e., pointer to array-of-13 integer entries.

Pointers and arrays

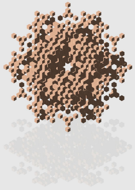
"Entries are stored by rows" means that, e.g.,

```
int v[100][200];
```

is allocated the same way as a

```
int v[100 × 200];
```

i.e., as if it were a mono-dimensional array of 20000 int's.



Pointers and arrays

Fortran stores objects the opposite way with respect to C:

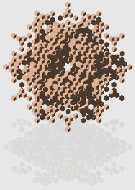
```
for (i..) for (j..) a[i][j]
```

is equivalent to

```
DO I.. DO J.. A(J,I)
```

Accessing element (i, j) in a $n \times m$ matrix means accessing element $k = i \times m + j$ in the associated mono-dimensional array. The same applies for N-dimensional matrices, $N > 2$.

Pointers and arrays

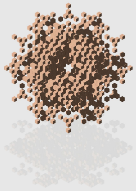


Note how, to compute value k for an N dimensional matrix whose dimensions are $(d_1, d_2, \ldots, d_N)$, it is necessary to know the values $d_2, \ldots, d_N$:

$$k = f(d_2, \ldots, d_N)$$

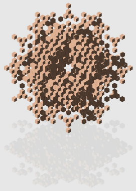
Note also that when we need to access sequencially all the elements of a multidimensional matrix it is preferable to use a pointer initialised to the first entry of the matrix.

Pointers and arrays



```
#define N 500
#define M 500
main() {
  int a[N][M];
  int i, j, c;
  int *pa = & (a[0][0]);
  int *pl = & (a[N-1][M-1]);

  while (pa < pl) {          for (i=0; i<N; i++)
     *pa = 0;                   for (j=0; j<M; j++) {
     c = *pa + 1;                 a[i][j] = 0;
     *pa = c;                     c = a[i][j] +1;
     pa++;                        a[i][j] = c;
  }                             }
```

Pointers and arrays



HP9K	0.7–0.8 second	1.2–1.3 seconds
SUN3	1.1 seconds	2.4 seconds

Pointers and arrays

Even when the access is not sequencial it is possible to exploit specific access patterns (e.g., constant stride access).

An interesting alternative with respect to multidimensional arrays is by using pointers. For instance, the computational cost to access entry (i, j) in a $n \times m$ matrix is the one for computing multiplication $(i * m)$ and addition $(\cdots + j)$.

Pointers and arrays

If we change from

```
int a[100][100];
```

to

```
int** a;
```

and if we properly allocate the 100 pointers in the row and, for each of them, the memory required for storing 100 int's, then accessing entry (i, j) means executing

```
*((*(a+i)+j))
```

that has a computational cost of only two additions. Furthermore, no dimension information is required anymore to access any element of the array.

Array of pointers

```
char *name_of_month (int n) {

 static char *names[] = {
     "illegal",
     "January",
     "February",
      .
      .
      .
     "December"
 };

 return ((n < 1 || n > 12)? names[0] : names[n] ;
}
```

Pointers and arrays

Argc and argv: a mechanism that allows a program to inspect the strings on the command line.

For instance, command `echo`:

```
main(int argc, char *argv[]) {
 while (--argc > 0)
   printf("%s%c", *++argv, (argc>1)?' ':'\n' );
}
```

Exercise: `compute 2 3 4 + *` (in inverse Polish notation).

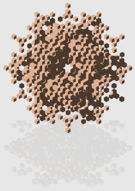
Exercise: write a function that associates user functions to the command options.

Pointers and arrays

Exercise: write a function that sorts the entries of an array of integers. Modify the function so that it requires a pointer-to-function, int (*confr)(int,int), to realize sortings in increasing vs. decreasing order.

Exercise: “opaque” function, working with pointers to object of unknown size.

Exercise (array of functions): design a scanner of a simple grammar. Tokens must correspond to the entries in an array of functions.



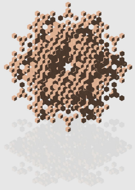
Structures

Keyword `struct` defines a “record.” An example follows:

```
struct cd {
    char author[30];
    int year;
    char producer[20];
};
```

This is a declaration of a *type*: no element of this type has been defined so far. No memory has been allocated. It is now possible to declare objects of type `struct cd` :

```
struct cd x, y, z;
```



Structures

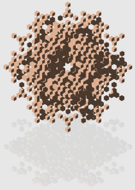
The members of a `struct` can be accessed via operator `.`.
For instance, in function on page 76:

```
struct cd d;

leap = d.year%4 == 0 &&
           d.year%100 != 0 ||
           d.year%400 == 0;
```

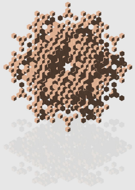
Nesting of structures is allowed: `struct a { struct disco c; } d;`

Access: `d.c.year`.

Typedefs.



Structures



A pointer to a structure can be declared, e.g., as follows:

`struct disco *a;`

Access: `(*a).year;`

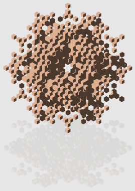
$$\texttt{(*a).year} \equiv \texttt{a->year}$$

Structures

Arrays of structures:

```
struct key {
   char *keyword;
   int  keycount;
} keytab[100];
```

Structures



Initialisation:

```
struct key Keywords[] = {
       "break", 0,
       "case", 0,
       "char", 0,
       .
       .
       "while", 0
};
```

Structures

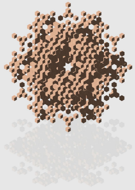
Write a program that writes a static arrays of structs to be used as look-up table for another program.

Structures can reference themselves:

```
struct tnode {
  char *word; int count;
  struct tnode *left;
  struct tnode *right;
};
```

or

```
struct nlist {
  char *name; char *def;
  struct nlist *next;
};
```



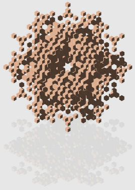
Structures: bitfields

Flag registers:

```
#define KEYWORD  01
#define EXTERN   02
#define STATIC   04
#define AUTOMT  010
 .
 .
flags |= EXTERN | STATIC;
 .
if ( flags & (EXTERN | STATIC) ) ...
```

It is possible to store in a same variable, `flags`, a series of conditions that can be “turned on” through a bitwise OR `|` and can be tested via the bitwise AND `&`. Clearly the definitions must be powers of 2. Octal implies `unsigned`.

Structures: bitfields



Structures can be used to specify flag registers via so-called "bitfields":

```
struct {
  unsigned is_keyword : 1;
  unsigned is_extern  : 1;
  unsigned is_static  : 1;
  unsigned is_regist  : 1;
} flags;
```

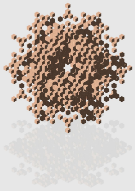
Accessing a field: standard way (`.` operator). For instance: `flags.is_extern`.

Structures: bitfields

Bitfields can be used as lvalues and rvalues as any other integer variable. This means that bitwise OR's and AND's are not necessary:

```
    flags.is_extern = 0

if (flags.is_extern == 0) ...
```

Structures: bitfields



- Bitfields can only be `unsigned` or `int`
- *Asking the address of a bitfield is illegal*
- Bitfields can also be “unnamed,” e.g., for padding
- The standard doesn’t specify if MSB-first or LSB-first is used.

Unions

An union is a variable having many identifiers associated to it. These identifiers may be of different type and hence different sizeof's. Each identifier refer to the same amount of memory, i.e., as many bytes as the "largest" type requires. For instance:

```
union point_x {
  char   *pc;
  float  *pf;
  double *pd;
  long    regarded_as_an_int;
} object;
```

Unions

Variable *object* has many “aliases”: it can be regarded as a pointer-to-char, if referred to as `object.pc`, or as pointer-to-double (`object.pd`), or as a long (`object.regarded_as_an_int`). The amount of memory is always the same, the one that is enough in order to store the “largest” among a (char*), a (float*), a (double*), or a (long).

Unions

Typical use of unions:

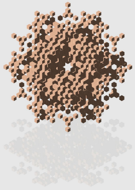
```
struct {
   int its_type;
   union {
       int    ival;
       float  fval;
       char  *pval;
   } uval;
} symbol_table[500];
```

If we store in `symbol_table[`*i*`].its_type` a code that represents the *type* of object *i*, then we can set up, e.g., arrays of objects of different types, or linked lists of different objects, and so forth.

Unions

A `switch` on `its_type` is the typical way to execute diverse actions depending on the nature of the current object, as it's done in the following excerpt:

```
for (i=0;i<500;++i)
  switch(symbol_table[i].its_type) {
    case 'i': printf("%d",
                symbol_table[i].uval.ival);
                break;
         .
         .
  }
```



Typedef

The typedef statement can be used to define new types without having to use cumbersome and long definitions. For instance,

```
typedef int LENGTH;
```

defines a type alias for `int` called `LENGTH`. Another simple example:

```
typedef char *STRING;
```

After this typedef the programmer can choose freely to define a pointer-to-char either via `char *` or with `STRING`.

Typedef

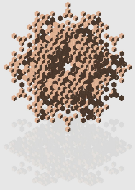
Typedef's are useful when types get complex and their definitions longer. For instance, type "pointer-to-function returning an `int`" can be aliased as follows:

```
typedef int (*PFI)();
```

Then, for instance, `PFI strcmp, swap;` is equivalent to

```
int (*strcmp)(), (*swap)();
```

Typedef



There are two valid reasons for encouraging the use of `typedef`:

- parametrizing a program with its types may turn into semplifying the solution of portability problems: for instance,

  ```
  typedef short integer;
  ```

  Moving the code to a machine where the role of short is played by int we only need to change one line of code:

  ```
  typedef int integer;
  ```

- enhancing a program's readability: a type called `LENGTH` brings with its name also a hint at its usage and meaning within the program.

Input and output

In C many functions are defined in the so-called “standard library”, `libc.a`. A set of headers refer to the functions and definitions in the standard library. The header file `stdio.h` contains the basic functions and definitions for I/O:

```
#include <stdio.h>
```

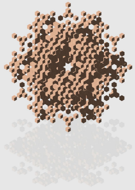
Input and output

C and UNIX define three standard I/O streams:

- `stdin`, i.e., the standard input stream, normally given by the characters typed at the keyboard. As in DOS, the redirection character $<$ allows to specify an alternative standard input stream as follows:

  ```
  prog < inputfile
  ```

- `stdout`, the standard output stream, normally given by the characters typed onto our display. Character $>$ redirects stdout on an other file, as in `prog > outputfile`.
- `stderr`, the standard error stream, is again by default our display. Is the stream where the programmer does (should) send the error messages. Can be redirected via 2 $>$, e.g., `prog 2> /dev/null`.



Input and output

Besides the stream redirection facilities, UNIX provides the concept of *pipe*: if we type

```
prog1 | prog2
```

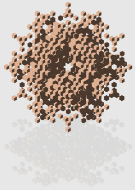
the `stdout` of prog_1 becomes the `stdin` of prog_2.

Input and output

Pipes are also available as C functions: for instance,

```
FILE *popen(const char *command, const char
                  *type);
        int pclose (FILE *stream);
```

the popen() function creates a pipe between the calling program and the command to be executed. `command` consists of a shell command line. `type` is an I/O mode, either r for reading or w for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is w, by writing to the file stream; and one can read from the standard output of the command, if the I/O mode is r, by reading from the file stream.



Input and output

```
FILE *f = popen("date", "r");
FILE *g=popen("/bin/sort", "w");
```

with the first one we can, e.g., read the output of command `date`. The second one connects to service `/bin/sort` so to ask the service of an external command.

Input and output

`printf()`

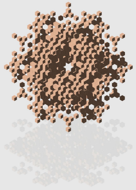
printf() converts, formats and prints its arguments (except the first one, a format string). The format string consists of two classes of objects: characters, that are simply copied onto stdout, and conversion strings, each of which controls the conversion and the printing of a further argument to printf().

Conversion strings start with character `%` and end with a *conversion character*. Between `%` and the conversion character there can be:

Input and output

- a hyphen, which implies left adjustment of the argument convertito
- a number specifying the minimum size of the output
- a “dot” character, separating the previous numerical field from. . .
- a number that specifies either the maximum number of characters that we can print from a string, or the number of decimal digits in a float or double.
- an `l` that means long integer.

Input and output



Conversion characters:

d	decimal
o	unsigned octal
x	unsigned hexadecimal
u	unsigned decimal
c	character

Input and output

Conversion characters:

s	string
e	float or double printed in “scientific” notation
f	ditto, decimal notation
g	as **f**.

(several other conversion characters exist.)

Input and output

```
scanf(formato, args...)
```

scanf() is the dual function of printf(): it reads characters from stdin and interprets and converts them according to what specified in its first argument, writing results into the following arguments. *These arguments must be (valid) pointers!*

```
int i;   float f;   char name[50];

scanf("%d  %f  %s", &i, &x, name);
```

Given input: `25 54.32E-1 Thompson` <*return*> the program associates 25 to `i`, 5.432 to `f` 5.432 and string `"Thompson"` to `name`.

The FILE Methodology

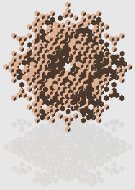
General idea:

- The user declares a pointer to an abstract entity called FILE:

  FILE *f;

- FILE can be seen as a *service*: the user must open a connection, send requests, receive replies, and close the connection in order to interact with that service.
- The user does not see the shape of the FILE struct; (s)he only sees its interface.

The FILE Methodology



- The user connects to the service via function fopen:

  FILE *f; f = fopen("myfile", "r+");

- Then the user sends requests to the service and gets replies from it:

  n = fread(myptr, sizemyobj, num_elems, f); if (n == num_elems) ...

- Finally, the user disconnects:

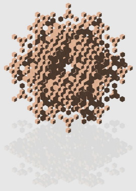
  fclose(f);

How to create a new FILE-like class

Example: class ASSOC

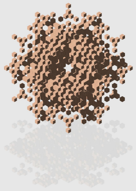
- Associative arrays: a mechanism with which one creates an *association* between two entities.
- A special class is the standard array, which sets up an association between integers and C objects:
  - double d[5];
- $0 \Rightarrow$ *d, ..., $i \Rightarrow$ *(d+i), ...

How to create a new FILE-like class



- Associative arrays create associations between any two objects.
- ASSOC creates association between any two *pointers*.
- “url0” ⇒ “https://github.com/Eidonko”: (char*) ⇒ (char*)
- “cos” ⇒ cos(): (char*) ⇒ (double(*)())
- “myfile” ⇒ f: (char*) ⇒ (FILE*)

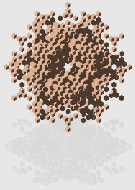
Class ASSOC



- Public header file: assoc.h (see `https://github.com/Eidonko/art`)

```
#ifndef _ASSOC_HEADER
#define _ASSOC_HEADER

typedef struct { brick root, *current, *p, *last;
} ASSOC;

#define A_OK       1     /* the couple is not deleted */
#define A_DELETED 2     /* the couple is deleted */

#define A_ALLOC    (-1) /* error code from unsuccessful malloc() *

/* A global variable, aerror, keeps track of the last error */
char aerror[512];
```

Class ASSOC



```
/* The functions' prototypes: */
ASSOC *aopen(void);
void   aclose(ASSOC*);
int    awrite(ASSOC*, void*, void*);
void  *aread(ASSOC*, void*);
void  *anext(ASSOC*);
void   arewind(ASSOC*);
void   adel(ASSOC*, void*);

ASSOC *acgi(void);
ASSOC *ascgi(char*);
ASSOC *aargcgi(char**, int);

ASSOC *aenv();
#endif /* _ASSOC_HEADER */
```

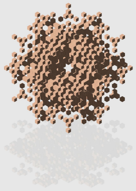
Class ASSOC

- The user has to supply a function to compare objects via their pointers
- strcmp()-like function
  - int myobj_cmp(void* l, void* r);
  - returns 0 when (*l) is equivalent to (*r)
  - returns a negative value when (*l) is “less than” (*r)
  - returns a positive value when (*l) is “greater than” (*r)

Class ASSOC

Using ASSOC (see also `https://github.com/Eidonko/art/`)

```
ASSOC *a;
a=aopen(strcmp);
awrite(a, "rome", "italy");
awrite(a, "london", "england");
awrite(a, "madrid", "spain");
printf("string ``madrid'' is associated with ``%s''\n", aread(a,
arewind(a);
while (s = (char*)anext(a) ) /* prints 3 associations */
   printf("domain %s: codomain = %s\n", s, aread(a, s));
adel(a, "london");
while (s = (char*)anext(a) ) /* prints 2 associations */
   printf("domain %s: codomain = %s\n", s, aread(a, s));
aclose(a);
```

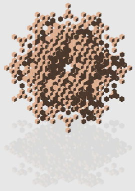


Class ASSOC

```
<Form method=POST action="/path/to/testweb">
Welcome to YAUD, yet another URL database. You can perform a numb
of actions on a local database of URLs; select an action, please!

<SELECT NAME="command"><OPTION SELECTED>view
<option>add <option>search
</SELECT>   # "command" can be associated to "view", "add", or "s

Please enter your URL...  <input name=url><p>
Please enter a brief description...  <input name=description><p>
If you want, tell me your name...  <input name=donator>
        # "url", "description" and "donator" are associated to st
Press this button to execute the command: <INPUT TYPE="submit" VA
```

Class ASSOC

- A special function, ASSOC* acgi(void), returns an associative array
- This array contains the associations:
  - “command” ⇒ “view”, or “add’, or “search”
  - “url” ⇒ some string
  - “description” ⇒ some string
  - “donator” ⇒ some string
- Another array associates strings to pointers-to-functions:
  - “view” ⇒ view()
  - “add” ⇒ add()
  - “search” ⇒ search()

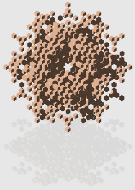


Class ASSOC

- The testweb.c program

```
void contrib(ASSOC *a) {
        ASSOC *b;
        int view(ASSOC*), add(ASSOC*), search(ASSOC*);
        int (*func)(ASSOC*);

        b=aopen(strcmp);
        awrite(b, "view", view), awrite(b, "add", add),
        awrite(b, "search", search);

        s = aread(a, "command"); /* dereference "command"... */
        /* ...and what "command" is associated to */
        if (s != NULL) func = aread(b, s);
        else           printf("invalid code\n");

        if (func != NULL)    (*func)(a);
```

Class ASSOC



- Function aenv() reads the environment strings and sets up associations of the type "REMOTE_ADDR" ⇒ "134.58.63.88" or "SCRIPT_NAME" ⇒ "/cgi-bin/atest.cgi"
- See `https://github.com/Eidonko/art/atest.c` and `.cgi`

Exercise: design a class of C functions for managing a stack of objects of any type.

Class ObjAlloc

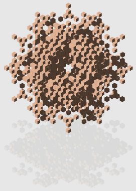
- An object allocation manager
- The user can define an array of a given size $s > 0$
- When the user stores value $s + 1$, the array is increased in size of a given amount $t > 0$.
- The previous contents are preserved.
- `https://github.com/Eidonko/ObjAlloc`

Class ObjAlloc

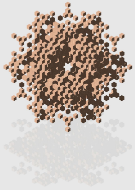
```
ObjAlloc_t *OA_open(size_t elem_sz, size_t s, size_t t)
```

- elem_sz : the size of the elements pointed to by the allocation object.
- s : the initial amount of elements pointed to by the object.
- t : the allocation increment.

```
/* current number of elements */
int OA_cardinality(ObjAlloc_t *o)
/* inserts element at position index */
void *OA_insert(ObjAlloc_t *o, size_t index,
                void *element);
/* returns the plain array maintained by o */
void *OA_array(ObjAlloc_t *o)
/* search elements in the objalloc array */
int OA_lsearch(ObjAlloc_t* o,void* elem,
    int(*cmp)(void*,void*),int*index);
int OA_bsearch (ObjAlloc_t* o,void* elem,
    int(*cmp)(void*,void*),int*index);
```



Class TOM : a Time-Out Management System



Time-out $\equiv$ an object that postpones a function call by a certain number of clock units.

Application-level time-out support :

- A common requirement for distributed services like, e.g., *membership protocols*
- A requirement of the *timed-asynchronous distributed system model*

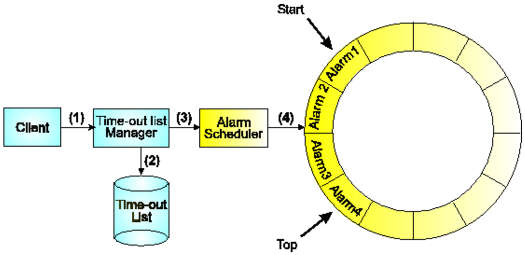
TOM :

- A class of C functions managing lists of time-outs
- An architecture for application-level time-out support

See code at `https://github.com/Eidonko/TOM`

Class TOM — Architecture

TOM: a client-server application

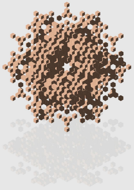


(1) Client sends requests to TOLM

(2) TOLM updates its time-out list

(3) When a time-out expires, TOLM sends an alarm request to $\mathcal{AS}$

(4) The $\mathcal{AS}$ choses an alarm server—or waits for any



Class TOM — API

1. /* declarations */
   TOM *tom; /* tom is the time-out manager descriptor */
   timeout_t t1, t2; /* two time-out objects, t1 and t2 */
   int my_alarm(TOM*), another_alarm(TOM*); /* two alarms */

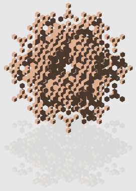
Class TOM — API

2.

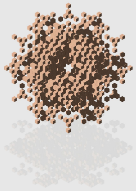
```
/* definitions */
tom ← tom_init(my_alarm);
tom_declare(&t1, TOM_NON_CYCLIC, TOM_SET_ENABLE,
                 TIMEOUT1, DEADLINE1);
tom_declare(&t2, TOM_CYCLIC, TOM_SET_DISABLE,
                 TIMEOUT2, DEADLINE2);
tom_set_action(&t2, another_alarm);
```

Class TOM — API

3. /* insertion */
   ```
   tom_insert(tom, &t1),
   tom_insert(tom, &t2);
   ```
4. /* control */
   ```
   tom_enable(tom, &t2);
   tom_set_deadline(&t2, NEW_DEADLINE2);
   tom_renew(tom, &t2);
   tom_delete(tom, &t1);
   ```
5. /* deactivation */
   ```
   tom_close(tom);
   ```



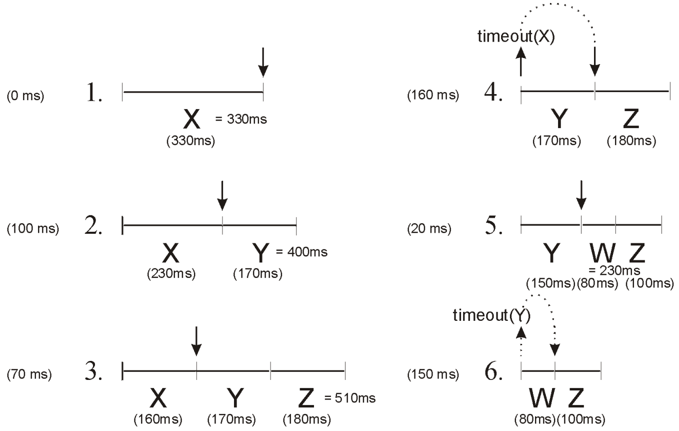
Class TOM — Server-side management



The time-out list is managed so that

- The top entry is the closest-to-expiration
- All the other entries' deadlines are relative to the expiration of the top entry
- A server-side protocol is used to adjust the list when inserting / deleting an entry from the top / middle / end of the list

Class TOM — Server-side management



Class TOM — Congestion

Problems:

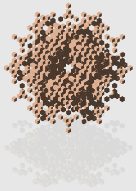
- Time-out management implies a delay
- Time-out detection delay is not always negligible
- Effect of concurrent execution on the same node
- *Alarm execution delay is not negligible*

Class TOM — Congestion

Alarm $\equiv$ a big source of non determinism:

- It is defined by the user $\Rightarrow$ unknown duration
- Alarm often triggers some communication $\Rightarrow$ communication and synchronisation delays
- Alarms compete with each other, e.g., for a memory port or a channel

Class TOM — Congestion



⇒ Risk : non negligible delay between

$t_{\text{expiration}}$ and $t_{\text{alarm execution}}$

⇒ Need : a way is needed

- to manifest,

  and, if possible,

- to keep under control

the **congestion** that is due to concurrent alarm execution.

Class TOM — Congestion

TOM supports multiple alarm execution threads.

- Does this solve the problem of congestion?
- (Are there cases in which the problem is solved, or at least, softened?)

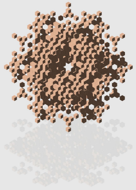
Class TOM — Congestion

Experiments have been performed.
Variables:

- alarm latency (δ),
- time-out deadline,
- number of alarm threads (τ),
- *competition*.

TOM_CYCLE = 50000 clock ticks (50ms).

Output: actual time of alarm minus expected time of alarm.



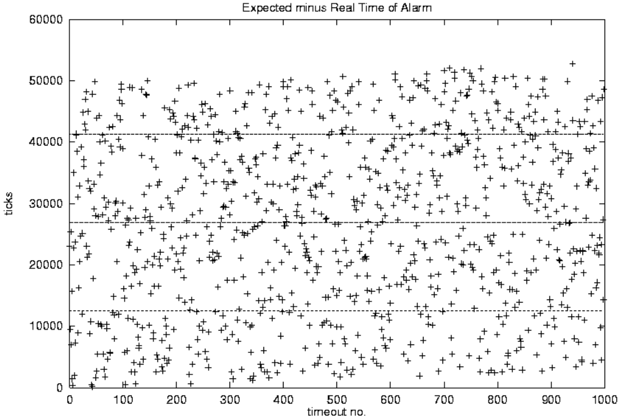
Class TOM — Congestion

Experiment 1:

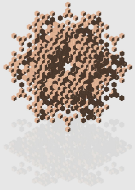
- $\delta \approx 50\mu$s
- Deadline: uniform in $[0, T]$
- Single alarm thread
- No competition (alarms execute sleep())

Deadline=50μs = overhead for calling a function and copying a 20-byte message on a DEC Alpha board running the TEX o.s.

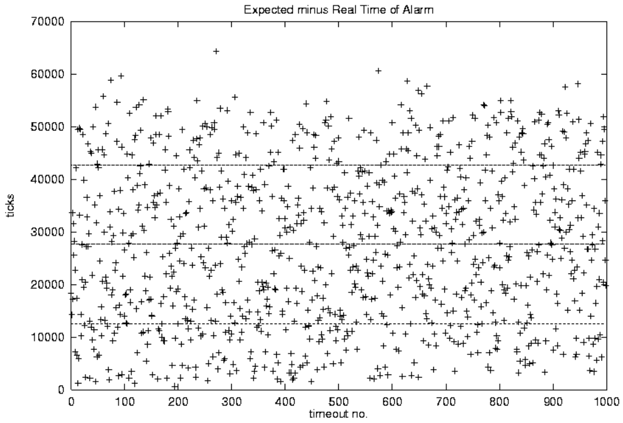
Class TOM — Congestion

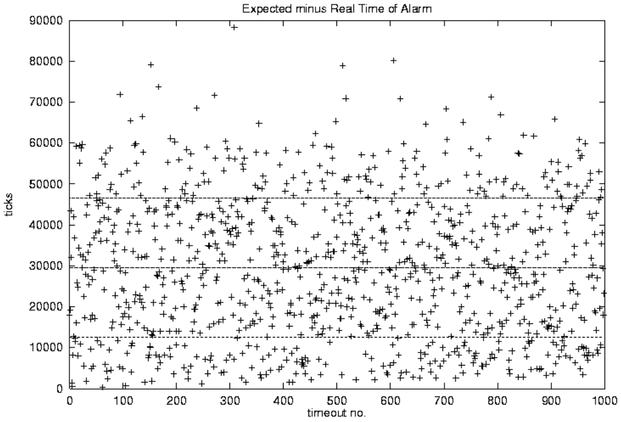


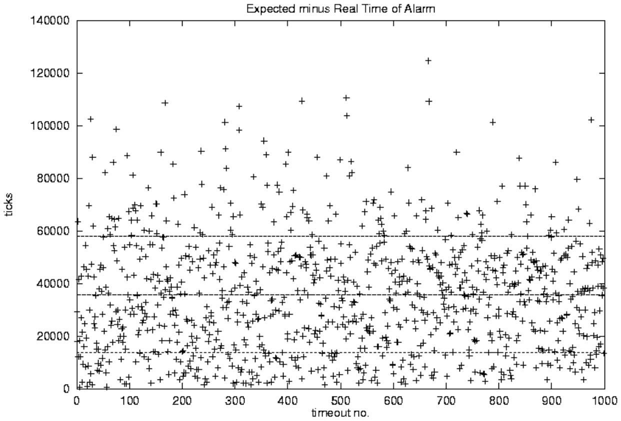
Class TOM — Congestion



- Max=52787, min=157, avg=26892.74, stdev=14376.89
- 20 $>$ TOM_CYCLE

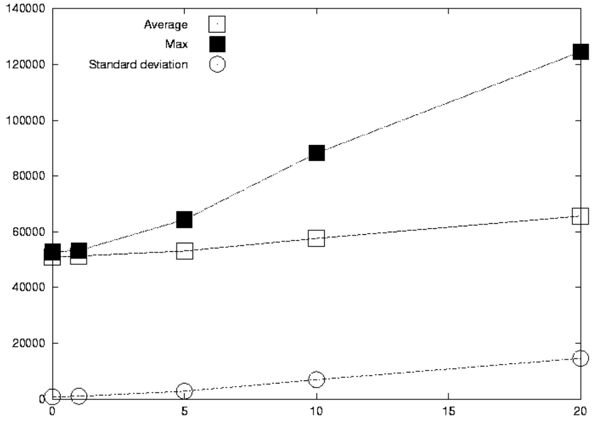

Expected minus Real Time of Alarm
ticks
timeout no.
0
10000
20000
30000
40000
50000
60000
70000
0
100
200
300
400
500
600
700
800
900
1000

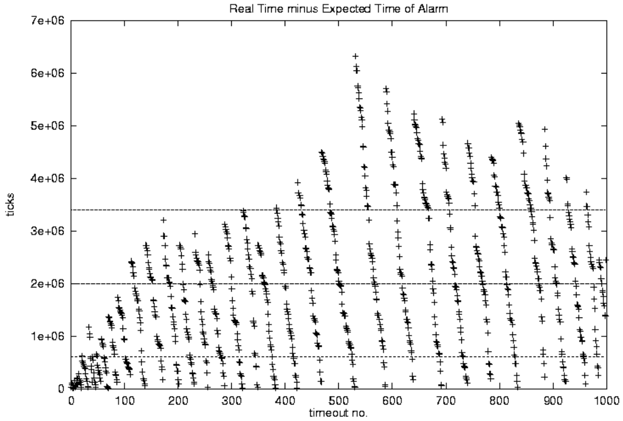

Expected minus Real Time of Alarm
ticks
timeout no.
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
0
100
200
300
400
500
600
700
800
900
1000

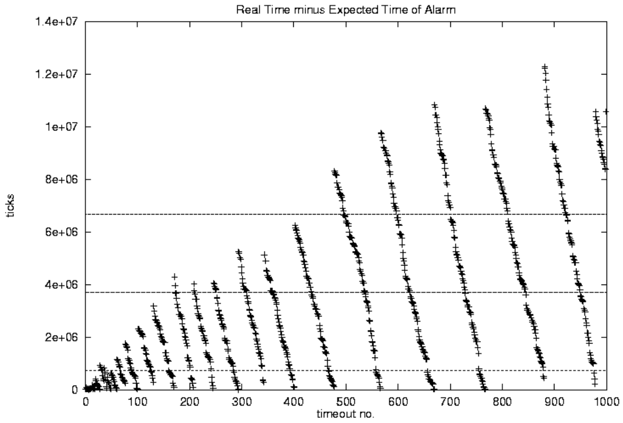


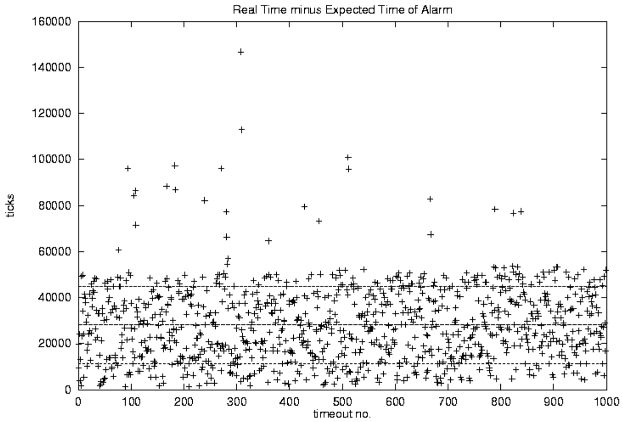
Expected minus Real Time of Alarm
ticks
timeout no.
0
20000
40000
60000
80000
100000
120000
140000
0
100
200
300
400
500
600
700
800
900
1000

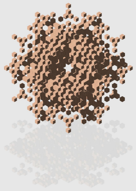
Class TOM — Congestion

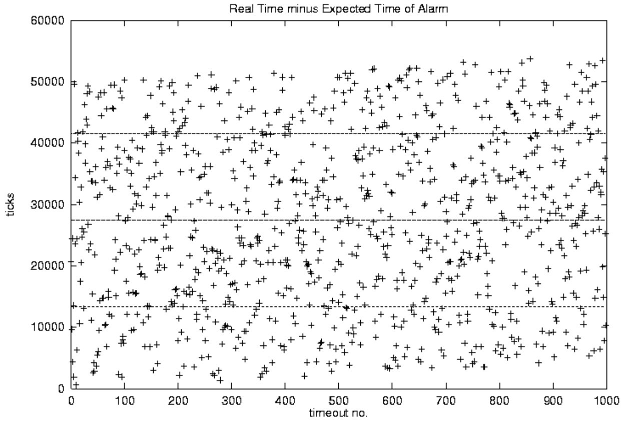



Real Time minus Expected Time of Alarm
ticks
7e+06
6e+06
5e+06
4e+06
3e+06
2e+06
1e+06
0
0
100
200
300
400
500
600
700
800
900
1000
timeout no.

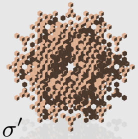

Real Time minus Expected Time of Alarm
ticks
1.4e+07
1.2e+07
1e+07
8e+06
6e+06
4e+06
2e+06
0
0
100
200
300
400
500
600
700
800
900
1000
timeout no.


Real Time minus Expected Time of Alarm
ticks
timeout no.
0
20000
40000
60000
80000
100000
120000
140000
160000
0
100
200
300
400
500
600
700
800
900
1000




Real Time minus Expected Time of Alarm
ticks
timeout no.
60000
50000
40000
30000
20000
10000
0
0
100
200
300
400
500
600
700
800
900
1000

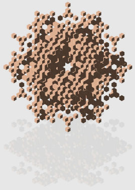
Class TOM — Congestion



τ	μ	σ	*exce–ptions*	max	μ'	σ'
1	3692158.86	2966039.23	974	12277216	3790057.80	2943375.88
2	33674.46	27688.17	140	264882	83244.94	38052.06
3	28177.01	16801.97	54	146684	66737.87	20705.76
4	27621.63	14276.78	45	100260	52871.73	7358.90
5	27435.40	14087.22	44	53712	51375.93	1001.36
6	27475.84	14107.32	44	53992	51443.95	1008.77

- μ and σ = average and stdev of the 1000 outcomes.
- μ' and σ' = average and stdev of the exceptions

Class TOM — Congestion



Congestion as a result of competition! (Alarms compete for the CPU). $\delta = 20$ms.

τ	μ	σ	γ	max	μ'	σ'
1	35251.93	21667.08	226	126931	66223.70	13713.40
2	35533.45	21584.35	248	127403	64377.88	13803.45
3	35735.19	21606.70	250	125910	64681.21	13305.83

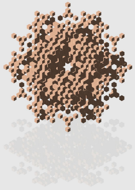
Conclusion: in the worst case adding alarm executors *does not* influence congestion!

Class TOM — Congestion

Solutions: resource-specific.

- Multicore CPU that dynamically and transparently schedules the parallel execution of alarms on the available cores.
- I/O $\Rightarrow$ parallel I/O systems.

Class TOM — Case Study: the Backbone



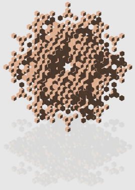
The backbone: a distributed middleware for fault-tolerant embedded applications.

The backbone tolerates crash failures of nodes and backbone components.

Based on TOM:

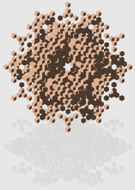
- IA_SET_TIMEOUT, IA_CLEAR_TIMEOUT
- MIA_SEND_TIMEOUT, TAIA_RECV_TIMEOUT
- MIA_RECV_TIMEOUT, TAIA_SEND_TIMEOUT

Class TOM — Conclusion



- Application-level time-out support
- Available on Windows, Parsytec EPX, TEX microkernel
- Multi-threaded architectures ⇒ better exploitment of a system's resources
- Problem of congestion due to the concurrent execution of alarms
- Successfully used in European projects
- "Algorithm of mutual suspicion"

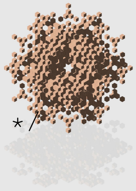
Class FN



- FN : a class for application-level input and interpretation of user-defined functions
- The user types in a mathematical function. This function can be then be evaluated.
- See `https://github.com/Eidonko/FN`
- Same principle and shape of class FILE:

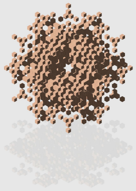
```
FN *f;     double *x, *y, z; float d;
printf("Please, type in a function of variables x and y\n"
f = fnopen();
```

Class FN



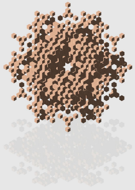
```
/* x and y point resp. to variables FN.x and FN.y */
x = fnmemory(f, 'x');
y = fnmemory(f, 'y');

printf("Please, type in a value for x and a value for y\n"
scanf("%f", &d); *x = d;
scanf("%f", &d); *y = d;

printf("The value of your function in %lf and %lf is %lf\n
       *x, *y, fnread());
```

Class FN



- Written with LEX and YACC
- Test program on the web: Julia sets
- Example: julia -1 1 -1 1 200, re(x,y)=x*x-y*y, im(x,y)=2*x*y+0.99)

Lexical and Syntactical Analysis with Lex and YACC

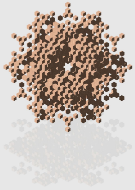


The capability

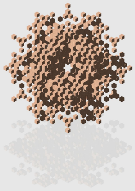
- to compose valid sentences in a given language, as well as
- to verify that a given string represents a valid sentence in a given language

builds upon two lower level capabilities:

Lexical and Syntactical Analysis with Lex and YACC



1. classification: the capability of decomposing a stream of characters into a stream of lexical entities (words, punctuation, delimiters) (lexical analysis), and
2. verification: the capability to recognize the syntactical correctness of a sentence, starting from a stream of lexical entities (syntactical analysis).



Lexical and Syntactical Analysis

Given, for instance, the mathematical expression:

$$sin(a + \sqrt(0.4))$$

1. the first capability means translating the stream of characters that corresponds to the above expression, i.e.,

   (`'s'`, `'i'`, `'n'`, `' '`, `'('`, `'a'`, ...)

   into a stream of tokens, or syntactical atoms:

   (`"sin"`, `'('`, `"a"`, `'+'`, `"sqrt"`, ...)
2. the second capability is the one that allows us to verify the syntactical correctness of the sentence, given a certain "grammar," i.e., in this case, the grammar of well-formed mathematical formulae.

Lexical and Syntactical Analysis

The above mentioned capabilities are experienced by the human being as inherent and natural abilities, of which one has not even full awareness.

When one has to set up, e.g., an interpreter of a computer language, or any other software module that needs to recognize a given structure in its input stream, then it is useful to set up a hierarchical structure at the base of which there are tools for lexical and syntactical analysis.

These tools are software systems that ease the development of lexical and syntactical analyzers. In UNIX, for instance, two standard utilities are available: Lex and YACC (or their GNU equivalents: flex and bison!)

Lexical and Syntactical Analysis

Lex and YACC allow to speed up considerably the development of parsers, translators, compilers, interpreter, conversion tools.

They have been especially designed for combined use and for hosting user-defined C routines where needed.

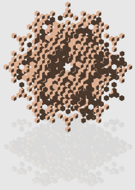
LEX: a lexical analyzer generator

LEX may be defined as a “tokenizator”: given a stream of chars, LEX performs a classification of groups of contiguous characters. These groups are called tokens, i.e., words and symbols that are *atomic* from the viewpoint of syntactical analysis.

For instance, LEX can translate string $sin(a + sqrt(0.4))$ in a set of couples “(token, token #)”, e.g., as follows:

- “sin”, `FUNCTION`
- “(”, `'('`
- and so forth.

The token # identifies the class the token belongs to.



LEX

LEX can be used

- either as a stand-alone tool, so to perform simple translations or compute statistics on the lexycal atoms,
- or in conjunction with a parser generator (e.g., YACC).

input $\overset{\text{LEX}}{\Rightarrow}$ tokens / errors $\overset{\text{YACC}}{\Rightarrow}$ valid / invalid sentences $\overset{\text{User code}}{\Rightarrow}$ user-defined actions

LEX

LEX writes a deterministic FSA from a list of **regular expressions** (regex). Regardless the number of rules supplied by the user, and regardless their complexity, the LEX FSA breaks the input stream into tokens in a time that is proportional to the length of the input stream.

The number of rules and their complexity only influence *the size* of the output source code.

Structure of a LEX program

The general structure of a LEX program is as follows:

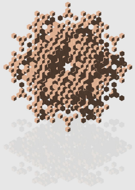
```
[ Definitions ]
%%
[ Rules ]
[ %%
User functions ]
```

Definitions and *User functions* can be missing.

Hence, the minimum size LEX program is the following one:

```
%%
```

LEX

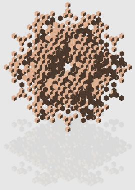


LEX performs its classification via a list of **regular expressions** (*regex*) that the user needs to supply via a standard language.

Regex's describe *patterns of characters* to be located in the text. LEX reads these regex's and produces a FSM that recognizes those patterns.

FSM's are indeed the simplest conceptual tool with which to recognize words expressed by regex's.

Metacharacters in LEX



LEX uses the same regex recognizer used by most of those UNIX tools that do pattern matching: `vi`, `sed`, `awk`, `find`, `grep`, for instance, adopt the same set of agreement based on the same set of “metacharacters”:

```
" \ [ ] ^ - ? . * + | { } $ / ( ) % < >
```

(Python, Perl, Java, and others, adopt slightly different sets.)

Metacharacters in LEX

`"` the quotation mark operator is the simplest metacharacter: all the characters of a string betweeb quotation marks are interpreted as plain (non-meta) characters.

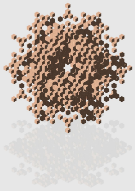
`[ ...]` Squared parentheses (pair []) specify classes of characters. For instance, `[xyz]` means: "*a single* `x`*,* `y` *or* `z` *char*"
The hyphen sign between any two chars *a* and *b* means that all the chars between ord(*a*) and ord(*b*) are specified. For instance, `[A-Z]` means "*any uppercase letter*", while `[A-Za-z]` means: "*any letter*".
Furthermore, `[\40-\176]` for instance selects a range of characters, that is, the one between *octal*(40) and *octal*(176).

Metacharacters in LEX

`[^ ...]` Character “^” , within the squared parentheses, means “complementary set”. For instance, `[^0-9]` means “*any char but the digits*”.

`\` (Backslash) is the same as in the C language function printf.

`.` (Dot) means “*any character but* `'\n'`”.

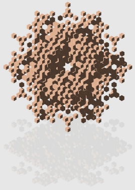


Metacharacters in LEX

`?` The question mark goes after optional strings of characters. For instance, `ab?c` means: "*either* `'ac'` *or* `'abc'`".

`*` Postfix operator "star" means *ZERO* or more instances of a given class. As an example, `[^a-zA-Z]*` means "*zero or more instances of non-alphabetic chars*".

`+` Postfix operator "plus" means *ONE* or more instances of a given class. For instance, `[xyz]+` means "*any non-empty string, of any size, consisting of any of the characters* `'x'`, `'y'` *and* `'z'`", such as e.g. `xyyyyyyzz`.

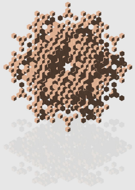


Metacharacters in LEX

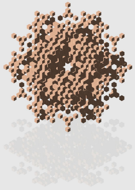
Operators `()` and `|`. Parentheses group a set of characters into one object. For instance, in `(xyz)+`, operator `+` is applied to string `xyz`. Within a group, the OR between entities is specified via metacharacter `|`. For instance,

```
(ab|cd+)?(ef)*
```

means "*zero or more instances of string* `"ef"`*, possibly preceded either by string* `ab` *or by* `cd+` *(*`c` *followed by one or more instances of* `d`*)*".



Metacharacters in LEX



`^`: This char, if not within square parentheses, means “at begin-of-file or right after a newline.”

`$`: This means “at the end of a line” or “at end-of-file”, i.e., if the following char is either `'\n'` or `EOF`. For instance, `(riga|row)$` means “string `riga` or string `row` followed either by `\n` or by `EOF`.

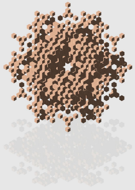
`/`: Infix operator slash checks whether an entity is followed by another one. For instance, `a/b` means “character `a`, only when followed by character `b`”. Note that `ab/\n` is equivalent to `ab$`.

Metacharacters in LEX

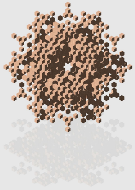
`{}`: Curly brackets have two meanings:

- When grouping two comma-separated numbers, as in `(xyz){1,5}`, they represent a *multiple instance*. The above example means "*from one to five instances of string* `xyz`".
- When grouping letters, they represent the value of a regex alias (see further on).

`%` Character `%` is *not* a metacharacter but has a special meaning.



LEX Definitions



A LEX source file may include up to three sections; the first one is the one including the LEX definitions. Definitions include a list of regex's:

```
letter          [a-zA-Z]
letters         {letter}+
```

These are the rules:

1. At column 1, an identifier is supplied,
2. then some blank or tab chars,
3. and finally a regex.

The identifier becomes an alias for its regex. To dereference an alias one has to put curly brackets around it.

LEX Rules

The Rules section is mainly a list of *associations* in the form

$$r \Rightarrow a$$

where r is a regex and a is a list of *actions*, i.e., user defined C language statements that are executed when the corresponding regex is recognized.

For instance:

```
%%
begin           printf("{");
end             {
                    putchar('}');
                }
```

LEX Rules : Actions

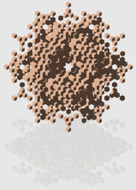
Note: when no rule is verified, a default rule is executed: `ECHO`.

(The FSA written by LEX has a `switch` statement with a `default:  ECHO;`.)

- This means that, e.g., there is no need to supply rules for the so called “literal tokens,” i.e., single characaters whose token number is equal to their ASCII code.
- To “sift out” some portion of the input, one needs to recognize it and to associate a null action to it.

To remove newline characters:

```
%%
\n          ;
```



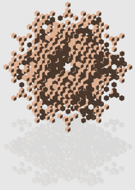
LEX Rules : Actions

Some “simple transformations” can be useful in order to facilitate the import of a file.

Some word processors, such as Word, regard paragraphs as a single line and separate paragraphs with `\n`.

The following LEX script converts every single `\n` into character space.

```
%%
\n\n          ECHO;
\n            putchar(' ');
```



LEX Rules : Variables

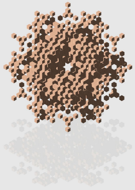
When a regex is recognized, the corresponding string (the token) is copied in a `char*` called `yytext`. This is true also for literal tokens.

This script is similar to the previous one:

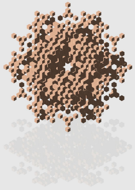
```
%%
[^\n]\n[^\n]   { putchar(yytext[0]);
                 putchar(' ');
                 putchar(yytext[2]);
               }
```

Action `ECHO` is actually a `#define`:

```
#define ECHO puts(yytext)
```



LEX Rules : Variables



Variable `int yyleng` is the number of characters of the string which verifies the current rule; in other words,

```
yyleng == strlen(yytext)
```

For instance:

```
%%
[0-9]+              dig += yyleng;
[a-zA-Z]+           alp += yyleng;
(.|\n)              oth++;
```

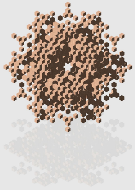
LEX Rules : Variables

The above program has some bugs:

1. Variable `dig` etc. have not been declared.
2. No output message is provided at the end.

LEX produces a C program. No checks are done on the correctness of this program. It may also contain syntax errors in the actions (actions are simply copied as strings into the output program.)

LEX Rules : Functions



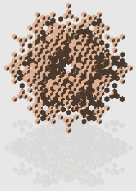
A number of functions are available to the LEX user:

```
yymore()
```

Next string is attached to the current value of `yytext`.

```
%%
\"[^"]*    {
            if (yytext[yyleng-1] == '\\')
                yymore();
            else
                do_that(yytext);
           }
```

LEX Rules : Functions



```
%%
\"[^"]*    {
            if (yytext[yyleng-1] == '\\')
                yymore();
            else
                do_that(yytext);
           }

        "he␣said␣\"hi\"."
        "he␣said␣\
                  "hi\
                      "."
```

LEX Rules : Functions

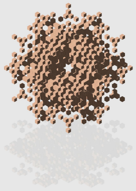
`yyless()`

“Sends back” a given number of characters.

```
%%
=-[a-zA-Z] {
          printf("Operator =- is ambiguous: ");
          printf("not recognized.\n");
          yyless(yyleng-2);
          manage_assignment();
          }
```

(In the early days of C, `a =- b` had the same meaning of `a -= b`).

`yyless(`*x*`)` pushes back onto the input `yyleng` − *x* characters.



LEX Rules : Functions

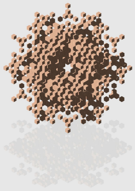
`int input()` reads the next input character. (Character `NULL` [that is, `(int)0`] is interpreted as end-of-file condition)

`void output(char c)` writes `c` onto the output stream

`void unput(char c)` “pushes back” `c` into the input stream.

The user can choose between a standard version of these functions or make use of his/her own functions with the same name and prototype.

LEX Rules : Functions



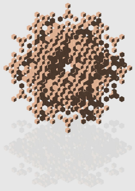
```
int yywrap(void)
```

This system (or user-) function is called when an `EOF` is encountered. The system version of this function returns `1`, which means “end of processing.” The user can substitute this function with a new version which, if it returns `0`, let the execution continue until a new `EOF` is encountered.

This way it is possible, e.g., to process more than one input file during the same run.

Furthermore, `yywrap()` allows the user to specify end-of-job functions (for instance, printing of the output and so forth.)

LEX Rules



LEX adopts two steps to select which user rule to apply:

1. The rule that recognizes the largest string is always preferred.
2. If more than one rule recognize largest strings, it is chosen the rule the user has specified first in the LEX script.

LEX Rules

Within a same rule, LEX returns the largest possible string:

```
%%
\'.*\'    { yytext[0] = '[';
            yytext[yyleng-2] = ']';
            printf("%s",yytext);
          }
```

produces a program that, once read string

```
'hi' -said- 'how are you?'
```

writes the following string on the output:

```
[hi' -said- 'how are you?]
```

LEX Rules

When LEX selects which rule to execute, it creates an ordered list of possible candidates. The one to be executed is the one on top. When the action includes macro

```
REJECT;
```

the following two actions take place:

1. the input string is sent back onto the input stream;
2. the rule is removed from the list. The rule that is selected is therefore the new top one.

LEX Rules

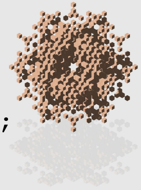
`REJECT` is useful, e.g., to count all the “digrams” in a given text:

```
%%
[A-Z][a-z] {    digram[yytext[0]][yytext[1]]++;
                REJECT;
           }
(.|\n)     ;
```

each digram in the text is located by the first rule, because it returns a string of *two* characters while the second one returns a string of just one character.

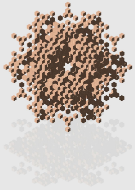
`REJECT` writes back the two characters of the digram onto stdin and “fires” the first rule. The second one is executed. A character is removed from the input stream.

LEX Rules



```
%%
[a-z][a-z] {  extern int dig[26][26];
              dig[yytext[0]-'a'][yytext[1]-'a']++;
              REJECT; }
(.|\n)     ;
%%
int dig[26][26];
int yywrap() { int i, j;
        for (i=0; i<26; i++)
          for (j=0; j<26; j++)
             if (dig[i][j])
                printf("digram [%c%c] = %d\n",
                            'a'+i,'a'+j, dig[i][j]);
        return 1;
}
```

Output stream in LEX

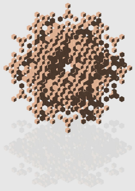


LEX allows to include in the output C source code any useful information (header files, declaration of global variables and so forth.)

Inclusion can be done in three “zones” of the output source file:

1. at the beginning of the file, that is, before any of the functions,
2. at the beginning of function `yylex()`,
3. at the end of the file.

Output stream in LEX



The three zones in the output source code correspond to the following zones of the LEX script:

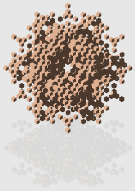
1. In *Definitions*,
2. On top of *Rules*, i.e., right after the first `%%`;
3. In *User Functions*.

Output stream in LEX

Case **3** is trivial. For **1** and **2**, we need to distinguish the text to be processed by LEX from the text that needs to be copied verbatim in the output file. To do this, one can follow any of these ways:

- `[ \t]+.*` (at least a blank space or tab character at column zero, then the data to be flushed onto the output file.)
- Anything between `%{` and `%}`.

Practical use of LEX



1. `lex` *`source`*`.l`
2. `gcc lex.yy.c -ll`
3. `a.out < input`

Practical use of LEX

File `lex.yy.c` contains function `yylex()` i.e., the actual scanner. Compiling `lex.yy.c` with the system library `libl.a`, a `main()` function is automatically supplied which calls function `yylex()`.

The user can substitute this default `main()` with one of their own design.

Doing this, one can choose between either automatically generating an executable or “piping” LEX output to other programs—for instance, syntactical analyzers.

LEX: Selection of a scanning context

Writing a lexical analyser can be made easier when using more than one scanning context. A scanning context is a set of scanning rules that apply within a certain context and do not apply in other contexts.

Classical example: the presence of string `/*` may imply the activation of a set of rules that are completely different from the standard rules. The same applies for constant strings.

The context switch can be done in various ways:

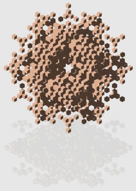
- flag method,
- start conditions,
- multiple scanners

Practical use of LEX

Selection of a scanning context: flag method

```
        int flag=0; /* starts with a tab! */
%%
"/*"    flag=1;
"*/"    flag=0;
.       |
\n      if (flag==0) putchar(*yytext);
```

It is the programmer's responsibility to use the method in a coherent way.



Practical use of LEX

Selection of a scanning context: start conditions/multiple scanners

An identifier, called “start condition,” is associated to some rules. The rule becomes part of the lexical context identified by the start condition. The current start condition can be changed at any time:

```
any          (\n|.)
%start       REMARK
%%
"/*"             BEGIN REMARK;
"*/"             BEGIN 0;
<REMARK>{any} ;
{any}            putchar(*yytext);
```

Finally, one can write multipler scanners and then activate the

LEX: Definitions

Apart from regex aliases, in *Definitions* it is possible to specify "internal codes" for any character:

```
%T
1          Aa
2          Bb
.....
26         Zz
%T
```

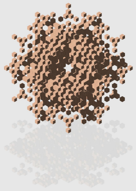
This allows definitions such as `[Dd][Oo][Uu][Bb][Ll][Ee]` to be avoided when the case of letters in the input is not important.

Note: code 0 is illegal; codes greater than $2^{\texttt{sizeof(char)}\times 8}-1$ are illegal; once a table has been defined, LEX only recognizes the characters in that table.

Practical use of LEX: exercises

- Write a CGI script that translates extended HTML to plain HTML
  - Clause <IF> <THEN> <ELSE> <FI> <REXEC> ...
  - <IF>REMOTE_ADDR = 134.58.63.88<THEN> ... <ELSE> ... <FI>
- Write a scanner to recognize the lexical atoms of a programming language

Practical use of LEX: exercises



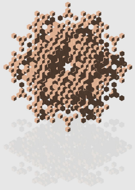
- Write a simple “translator” for a Pascal-like pre-processor

```
BEGIN {
END   }
EQ    ==
IF    if(
THEN  )
END;  }
END.  }
```

LEX: bibliography

1. M.E Lesk, E. Schmidt, *LEX - a lexical analyzer generator*, in ConvexOs Tutorial Papers, CCC, 1993.
2. `http://www.combo.org/lex_yacc_page/` : the lex & yacc page

Syntactical Analysis with YACC



YACC : *Yet Another Compiler-Compiler*

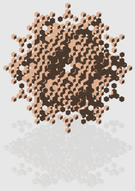
YACC has been defined by its authors as a system for describing the input structure of a program.

The YACC programmer is required to supply:

1. the syntactical structure of the input
2. C code to be executed when the syntax rules are recognized.

On the basis of these data, YACC writes a C program with a parsing routine.

YACC



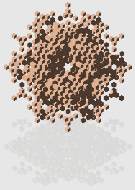
The parsing routine calls a lower level routine, called `yylex()`, in order to get the next lexical atoms in the input stream.

YACC works with grammars of type **LALR(1)**, plus rules to solve ambiguities.

YACC

The general structure of a YACC script strictly follows the one of a LEX script:

```
[ Definitions ]
%%
Rules & Actions
[ %%
User functions ]
```



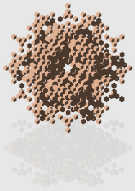
YACC

In particular, the structure of *Rules & Actions* is similar to the corresponding section of a LEX script: it includes a set of *grammar rules*, plus *actions* that are associated to each rule.

Each time a rule is recognized, the corresponding actions are executed.

Actions may return values and use the values returned by other actions.

YACC



YACC rules have the following structure:

lhs `:` *rhs* `;`

where *lhs* is a non-terminal symbol and *rhs* is a sequence of zero or more terminal or non-terminal symbols, “literals,” and actions.

Identifiers for terminal and non-terminal symbols follow the rules of the C language, with the addition that character `'.'` is considered as a letter.

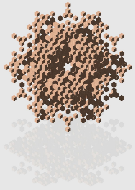
YACC

A literal is a constant character defined as follows:

```
literal : QUOTE char QUOTE
        | QUOTE BACKSLASH char QUOTE
        | QUOTE BACKSLASH od od od QUOTE
         ;
```

(being `QUOTE` character `"` and `od` an octal digit.)

Character “`|`” is an OR. It is used when more than one rule has the same *lhs*.



YACC: user definitions

As with LEX, the parentheses `%{` and `%}` allow to include in the output of YACC any C source code. This code is global with respect to the parser function and to the user functions.

YACC uses a number of identifiers starting with “`yy`” for internal purposes. This prefix must be avoided.

YACC: token declaration

Lexical atoms (the tokens) must be explicitly declared in *Definitions*. This is done, for instance, by writing one or more lines such as the following one:

`%token` $nome_1$ $nome_2$ $\ldots$

All the symbols that have not declared as tokens are implicitly declared as non-terminals (NTs).

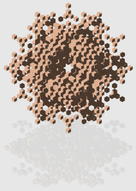
Note: each NT must be the *lhs* of at least one rule.

YACC: specification of the start symbol

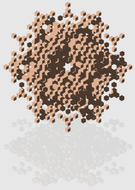
The declaration of the start symbol of the grammar may be done as follows:

`%start` *name*

in *Declarations*. If this specification is missing, it is assumed that the start symbol is the *lhs* of the first grammar rule specified by the user.



YACC: the endmarker token



A special token marks the end-of-input. This is called *endmarker* in YACC lingo.

If the tokens encountered between the start of processing and the *endmarker* (not including the latter) *verify* the start symbol, then the parsers successfully stops processing after having read the *endmarker*.

Reading the *endmarker* before the start symbol is verified leads to an error.

YACC: actions

Within each rule, the programmer can specify some *actions* to be executed each time that rule is recognized while analysing the input stream.

Actions may return values and use the values returned by other actions.

Also the tokens returned by `yylex()` may have values.

Actions are a number of C statements between curly brackets. Each action can return a value by setting variable `$$`. For instance:

```
    { action(); $$=1; }
```

returns 1.

YACC: actions

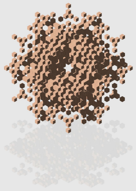
Also the rules may return values. This value is either the value of the first component or the value of variable `$$`. For instance:

```
A : B;
```

is equivalent to

```
A : B { $$ = $1; } ;
```

YACC: actions



The following example shows how it is possible to use the values returned by previous rules.

```
expr    :   '('   expr   ')'
            {    $$ = $2;    }
        ;
```

In other words, $*i* is the value returned by RHS[*i*].

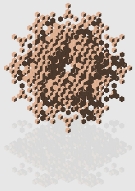
YACC: actions

An example follows:

```
expr :   expr    infix_op     expr
         {  $$ = node( $2, $1, $3);  }
     ;
```

For instance, function `node()` may allocate an object and return its address. This is used when building syntax analysis trees.

Note: values returned by rules and actions are integers by default.



YACC: function `yylex()`

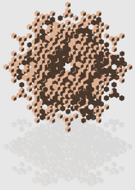
Function `yylex()` returns an integer—the token number. This number is either a literal (when in $[0, 255]$) or a symbolic constant $s > 256$ that describes the lexical “class” the recognized string belongs to. For instance, `NUMBER`.

Function `yylex()` also returns the actual string that was found in the input. That string is kept in variable

```
extern X yylval;
```

where **X** is either `int` or can be defined by the user.

```
%%
[0-9]+  { yylval=atoi(yytext); return NUMBER; }
```



YACC: token numbers

The choice of which integers to use with token can be done

- *automatically* by YACC, which associates the integers from 257 one by one to the tokens that have been declared with the `%token` keyword.
- *implicitly* for literals, to which is associated the ASCII code.
- *explicitly* by the YACC programmer, who can associate an integer greater than 0 after the name of a token or a literal in section *Declarations*.

YACC: token numbers

Token numbers must be different.

When executing `yacc` with the `-d` option, a header file is created, called `y.tab.h`, which contains all the token numbers. This file can be included, e.g., in the LEX script as follows:

```
%{
#include "y.tab.h"
%}
```

Note: the C program produced by LEX can be either compiled separately or even included in the YACC output program by specifing in *User functions* the following statement:

```
#include "y.tab.c"
```

YACC: choice of the lexical analizer

LEX is usually “*the*” lexical analizer to be used with YACC. Under specific circumnstances, though, LEX may be less suited than an ad-hoc lexical analizer. For instance, when recognizing a Fortran grammar, it may be difficult with LEX to express conditions that depend, e.g., on the column where a given command starts.

YACC algorithm

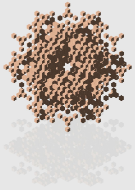
YACC produces a FSM with two parallel stacks representing states and values:

```
short yyssa[YYINITDEPTH]; /* the state stack */
YYSTYPE yyvsa[YYINITDEPTH]; /* the value stack */
#define YYPOPSTACK   (yyvsp--, yyssp--)
```

The parser can read and store the next input token [this is called look ahead token (*lat*) in YACC lingo.]

The state stack, S, is a vector of integers. The current state is always TOP(S).

Initially *lat*$=\Lambda$, `sp=0`, TOP(S)=0.

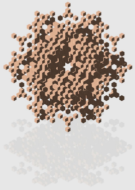


YACC algorithm

The FSM mainly makes use of four basic actions: *shift*, *reduce*, *accept* ed *error*. (A fifth action, *goto*, is actually a *shift*).

The main loop of YACC is in two basic steps:

1. On the basis of TOP(S):
   is *lat* required? Yes: read *lat* (call `yylex()` and so forth.)
2. On the basis of both TOP(S) and *lat*:
   choose next action.



YACC algorithm

YACC uses a symbolic language to describe the structure of the output FSM. Running YACC with the option

```
-v
```

a file called `y.output` is produced. This file contains a number of statements in the form:

symbol opcode operand

or

. opcode operand

where *opcode* is `shift` or `reduce` and so forth, and *operand* is an integer that represents either a *state* or a *grammar rule*.

YACC algorithm

Reading `y.output` is a way to ease up the debugging of a YACC program. Now it is described how to interpret `y.output`.

$\boxed{\textit{shift}}$

symbol `shift` *new-state*

Within the current state:

- if *lat*$=\Lambda$, read(*lat*).
- if *lat*=*symbol*, push(*new-state*, `yylval`).
- *lat* $\leftarrow \Lambda$

YACC algorithm

Push and pop are executed in parallel on the two stacks.

“.” means “any symbol”.

YACC algorithm

shift makes the stacks grow. The dual action is

$\boxed{\textit{reduce}}$

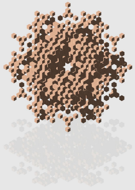
`. reduce` *rule*

reduce comes into play the moment the parser has finished the scan of an *rhs* and needs to replace it with the non-terminal that is *lhs* in the corresponding rule.

Usually *reduce* is unconditioned, though it may also take the form

symbol `reduce` *rule*

Rule is an integer identifying a given rule.



YACC algorithms: reduction algorithm

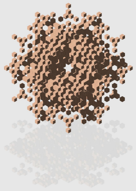
- the action corresponding to the rule is executed;
- `for (i=0; i<` ν`(`*rhs*`); i++) pop();`
- TOP(*S*) (the so-called "*uncovered state*") is inspected
- ...when a command like this:

  lhs `goto` *x*

  is found, the following command is executed:

  `push(`*x*`).`

goto differs from *shift* in that it does not erase the *lat*.



YACC algorithms: reduction algorithm

reduce purges the states corresponding to the *rhs* and lets the YACC FSM behave as if it had encountered symbol *lhs*.

As a special case, void rules such as

```
A : ;
```

mean:

- do not execute any pop();
- inspect the current state TOP(*S*)...
- ...looking for an instruction such as
  `A goto` ***x***

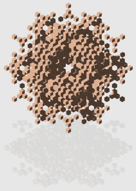
YACC algorithms: reduction algorithm

When a rule is encountered, right before executing a *reduce*, the action associated with that rule is executed.

This action has access to all the values of its components by means of variables `$1`, `$2` and so forth.

These numbers represent displacements within the value stack.

YACC algorithm



$\boxed{\textit{accept}}$

```
if (lat == endmarker) return OK;
```

i.e., if the entire input has been inspected and it verifies the rules, then conclude with success.

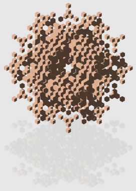
$\boxed{\textit{error}}$

If in the uncovered state there is no valid next state, and if *lat* is not equal to *endmarker*, an error condition is raised.

YACC algorithm

The above five basic actions are the key to understand file `y.output`. A (classic) example follows:

```
%token  BARKS   DOG    THE
%%
sentence:       subject    verb
        ;
subject :       THE    DOG
        ;
verb    :       BARKS
        ;
```



YACC: the y.output file

```
state 0
  $accept : _sentence $end

  THE  shift 3
  .  error

  sentence  goto 1
  subject  goto 2

state 1
  $accept :  sentence_$end

  $end  accept
  .  error
```

```
state 3
  subject : THE_DOG

  DOG   shift  6
  .  error

state 4
  sentence : subject verb_

  .  reduce 1


state 5
  verb :  BARKS_    (3)
```

YACC: the y.output file

```
state 2
  sentence :  subject_verb

  BARKS  shift 5
  .  error

  verb  goto 4
```

```
  .  reduce 3

state 6
  subject : THE DOG_

  .  reduce 2
```

YACC: the y.output file

The underscore character ("_") marks the border between what the parser has "seen" already and what is yet to come.

$accept is equivalent to `sentence` followed by the *endmarker*.

Now let's suppose we have the following as our input string: `"THE DOG BARKS"`.

Initially, *S* only contains state 0 and *lat* is undefinito: $(S = (0), lat = \Lambda)$.

State 0: a *shift* requires reading the *lat*: (*lat*=`THE`).

Action `THE` *shift* `3` brings the system to state 3 and erase *lat*: $(S = (0, 3), lat = \Lambda)$.

State 3: same as above: (*lat*=`DOG`).

YACC: the y.output file

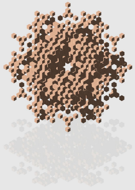
State 6: unconditioned reduction via rule 2 ($S = (0)$, *lat* $= \Lambda$, lhs=`subject`): two `pop()`'s remove states 6 and 3 and "uncover" state 0.

State 0: a *goto* brings to state 2: ($S = (0, 2)$, *lat* $= \Lambda$)

State 2: *lat* is read and `BARKS` *shift* `5` is executed ($S = (0, 2, 5)$, *lat* $= \Lambda$).

State 5: by unconditioned reduction, state 5 is purged: ($S = (0, 2)$, *lat* $= \Lambda$, lhs=`verb`).

YACC: the y.output file



State 2: a *goto* brings to state 4: ($S = (0, 2, 4), lat = \Lambda$).

State 4: reduction by rule 1: ($S = (0), lat = \Lambda$, lhs=`sentence`)

State 0: a *goto* brings to state 1: ($S = (0, 1), lat = \Lambda$).

State 1: reading the *endmarker* brings to *accept*.

YACC: the y.output file

As an exercise, verify the behaviour of the parser when reading invalid input strings, e.g., `THE DOG DOG`, `THE DOG BARKS THE`, and so forth.

Some minutes spent on the interpretation of `y.output` can save hours of debugging time.

YACC: associativity, priorities, ambiguities

A YACC rule is said to be an ambiguous rule if there exists an input string to which two or more different structures can be associated.

For instance, given

```
  expr  :  expr '-' expr ;
```

and given input “$a - b - c$”, two possible structures (i.e., interpretations) exist:

$$(a - b) - c \tag{1}$$
$$a - (b - c) \tag{2}$$

YACC: associativity, priorities, ambiguities

YACC detects those ambiguities. In YACC lingo, they are called "*shift/reduce conflicts*" and "*reduce/reduce* conflicts".

Let us suppose we are in the following situation:

```
expr - expr_ - expr
```

At this point the parser must arbitrarily choose between:

1. a *reduce*, which brings to `expr_ - expr` followed by an other *reduce*,
2. a *shift*, which brings to `expr - expr - expr_` and finally to two *reduce*'s.

YACC: associativity, priorities, ambiguities

As it is clear from the example, reduction implies left association, while a *shift* implies a right association.

Ambiguities are called SHIFT/REDUCE CONFLICTS. It is also possible that YACC cannot choose between two or more reductions, which is called a REDUCE/REDUCE CONFLICT.

YACC: associativity, priorities, ambiguities

Conflicts of type s/r and r/r are not considered as *errors* but rather as warnings. YACC goes on producing its parser choosing what to do on the basis of the following rules:

1. in case of s/r conflict, execute a *shift*;
2. in case of r/r conflict, it is chosen the *reduce* that the user specified first in the YACC script.

YACC: associativity, priorities, ambiguities

An other example:

```
stat : IF '(' cond ')' stat
     | IF '(' cond ')' stat
           ELSE stat
     ;
```

When the input is like follows:

`if (`c_1`) if (`c_2`)` S_1 `else` S_2

the parser needs to choose between two different interpretations.

YACC: associativity, priorities, ambiguities

Let us consider the following situation:

`if (`c_1`) if (`c_2`)` S_1_ `else` S_2

At this point,

- a *reduce* can take place, in which case `else` matches with the first `if`,
- a *shift* can take place. This is the correct interpretation e.g. in C.

YACC: associativity, priorities, ambiguities

Some arithmetical operators have their own associativity conventions, and by agreement there are priorities between them. Therefore a method is required in order to set a priority among operators and to choose beforehand the type of associativity that is required.

YACC: associativity, priorities, ambiguities

The kind of associativity of an operator can be defined in YACC by the three directives:

```
%left  %right %nonassoc
```

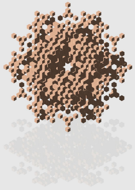
They also represent an alternative way to declare tokens and literals with respect to `%token`. For instance,

```
  %right   '='
  %left    '-'   '+'
```

choose right association for the assignment operator

(that is, $a = b = c$ means $a = (b = c)$)

and the left association for `'+'` and `'-'`.



YACC: associativity, priorities, ambiguities

Each row defines a priority level. The earlier the specification appears in the source file, the lower its priority:

$$'=' \prec ('+', '-') \prec \cdots$$

YACC: associativity, priorities, ambiguities

For instance,

$$a = b = c * d - e - f/g;$$

is interpreted as

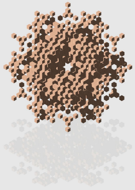
$$a = (b = (((c * d) - e) - (f/g)));$$

Keyword `%nonassoc` specifies that a certain operator must *not* be applied more than a single time. For instance, in Fortran the following expression

```
A .LT. B .LT. C
```

is not valid. `%nonassoc` tokens catch these conditions.

YACC: associativity, priorities, ambiguities

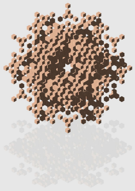


Case exist in which a same sign, for instance `'-'`, has two different meanings and priorities:

```
expr : expr '=' expr
     | expr '*' expr
     | expr '-' expr
     | '-' expr
```

Unary “minus” has greater priority than that of diadic “minus”. In this cases one can make use of a fictious token and operator `%prec`:

YACC: associativity, priorities, ambiguities



```
%left '+' '-'
%left '*' '/'
%left UMINUS
%%
expr : expr '-' expr
     |   ....
     | '-' expr   %prec   UMINUS /* same priority of UMINU
```

YACC: associativity, priorities, ambiguities

Operator priorities cast priorities among the rules.

We define the priority of a rule as either the priority of the last token/literal in its *rhs* or the priority specified with `%prec`.

We can now summarize the rules concerning the conflicts:

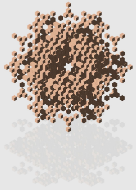
YACC: associativity, priorities, ambiguities

Two properties come into play when solving an s/r or an r/r conflict: priority and associativity.

1. of the current rule, and
2. of the token currently being read.

- if 1. **or** 2. do not have a priority / associativity, then the default rules are executed (shift vs. reduce, order among reduces) and <u>a warning is issued</u>;
- if 1. **and** 2. have a priority / associativity, the we select the conflict on the basis of the priority and, when priority coincides, on the basis of associativity.
  <u>No warning is issued</u>.

YACC: error management

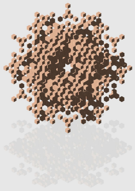


In general, error management is not a trivial task. When some inconsistency appears during the elaboration it is important to provide the user with a valuable assistance.

Choosing to stop processing at the first error, or to loose control and report many fictious errors triggered by a first one, are not good choices from the user viewpoint.

What is needed is to be able to detect the next consistent state (if any) and to continue processing from that point on. This way, e.g., further syntax errors can be reported.

YACC: error management



In order to perform non-trivial error management, YACC defines the token called

$\boxed{\texttt{error}}$

which can be used in any rule as a marker for possible places where we expect some error to show up. In these places we can place some code for error management.

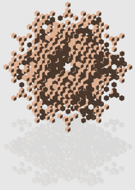
YACC: error management

The parser executes a number of `pop()`'s from the stack of states until it gets out of the error condition. At this point, *lat* is equal to the error token, which means that rule is satisfied and the corresponding action is executed.

At the end of error management, *lat* is set to the value of the token that introduced the error condition.

If there's no `error` token in a rule in error, the execution is aborted.

YACC: error management



The standard function for error management is as follows: the parser looks for the first three legal token instances and then starts back processing from the first of these.

This context loss may lead to inconsistencies: if, e.g., the three valid tokens are not at the beginning of a rule, the parser may take a wrong path that could result in weird errors.

YACC: error management

Errors of this type may be due to rules such as:

```
stat :  error ;
```

Better rules take the form, e.g., of

```
stat :  error ';' ;
```

In this case we synchronize the parser with the beginning of a new statement.

YACC: error management

The following macro:

```
yyerrok;
```

tells YACC to cancel the current error condition. This is useful when doing interactive processing, e.g., like follows:

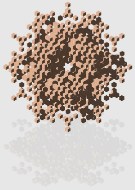
```
input : error '\n'
        { yyerrok;
          puts("? Redo from start.\n");
        }
        input
        { $$ = $4; }
      ;
```

YACC: error management

As already said, *lat* is normally set to the value of the token that triggered the error. It is also possible to clear the *lat* as follows:

```
yyclearin;
```

YACC: output



YACC produces the source file `y.tab.c` or *filename*_`tab.c`). This file mainly contains function

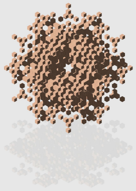
```
int yyparse()
```

To each call of function `yyparse()` corrispond one or more call to function `yylex()`.

When `yyparse() == 1`, the parser has found an error.

When `yyparse() == 0`, the parser has executed *accept*.

YACC: User functions



The user needs to supply two functions, e.g., after the second “%%”: `yyerror(char*)` and `main()`.

These functions may also be very simple:

```
#include <stdio.h>
main() { return yyparse(); }

yyerror(char*s) {
  fprintf(stderr,  "%s\n", s);
  /* or write line number, or... */
}
```

YACC: Special functions and variables

```
yyerror(), yychar, yydebug
```

The following variable contains the token number of the *lat* the moment the error took place:

```
yychar
```

This is an information that may be returned to the user, e.g., with `yyerror()`.

YACC: Special functions and variables

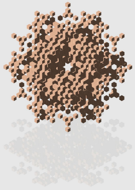
```
yyerror(), yychar, yydebug
```

Variabile

```
int yydebug;
```

is normally 0. When this is not true, YACC prints a verbose description of the decision it took during its analysis:

```
  if (yydebug)
    fprintf(stderr, "Shifting \
token %d (%s), ", yychar,
yytname[yychar1]);
```



YACC: Rules of style

Kernighan’s rules:

- Use uppercase letters for tokens, lowercase letters for non terminals
- Write rules and actions on different lines
- Group all the rules sharing a same *lhs* by means of operator “`|`”
- At end-of-rule, write the closing “`;`” at the same column of “`:`” and “`|`”
- Indent *rhs* by two tabs, actions by three tabs.

YACC: optimal performance

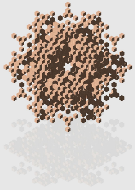
The program produced by YACC is a pushdown automaton. This matches particularly well with left recursion. This means that it is much better to use rules of the form:

```
    nt  :  nt  etc ;
```

with respect to those of the form:

```
    nt  :  etc  nt ;
```

The latter ones force YACC to perform a large quantity of *shift*'s that may even lead to a stack overflow condition.



YACC: optimal performance

Hence the following rules are preferable:

```
list_id   : id
          | list_id  ','  id
          ;
```

YACC: technicalities

Let us consider the following YACC rule:

```
seq    :  /*  NOTHING */
           { init_seq(); }
       |    seq   item
           { manage_new_item($2); }
       ;
```

Function `init_seq()` is called just once, right before processing the first `item`, while instruction `manage...` is executed at each new `item`.

YACC: return values

The value stack of YACC is by default based on integers.

This user can choose any different type.

That stack is organized as a vector of `union`'s. The programmer can declare such `union` and associate the name of its members with the tokens and non-terminals that return a value.

When the user does declare the `union`, the following string is attached to any reference like `$$` or `$`*i*:

. *field-name*

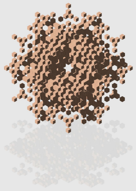
YACC: return values

Example:

```
%union {
   char  *String;
   double Real;
   int    Integer;
}
```

An equivalent way to shape this union is by defining explicitly type `YYSTYPE`:

```
typedef union {
      char  *String;
      double Real;
      int    Integer;
   } YYSTYPE;
```



YACC: return values

One can associate a field-name to a token:

```
%left    <Integer>    '+'   '-'
%right   <Real>       '='
```

One can associate a field-name to a non-terminal:

```
%type    <String>    expr
%type    <Real>      number
```

YACC: return values

One can associate a field-name to an action:

```
expr : '(' strexp ')'
        { $<Real>$ = atof( $<String>2 );
        }
     ;
```

i.e., “`$<`”, followed by a field-name, followed by “`>$`”

YACC: bibliography

1. S. C. Johnson, Yacc: Yet Another Compiler Compiler, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.
2. T. Mason, D. Brown, *lex & yacc*, 2nd edition. O'Reilly and Associates, inc. 2012.
3. `http://www.combo.org/lex_yacc_page/` : the lex & yacc page

LEX and YACC: an example

Lex and YACC sources for the FN class (`https://github.com/Eidonko/FN`)

LEX source (excerpt)

```
dreal   ([0-9]*\.[0-9]+)
ereal   ([0-9]*\.[0-9]+[eE][+-]?[0-9]+)
int   [0-9]+
real   ({dreal}|{ereal}|{int})
acos   [Aa][Cc][Oo][Ss]
asin   [Aa][Ss][Ii][Nn]
atan   [Aa][Tt][Aa][Nn]
cos   [Cc][Oo][Ss]
sin   [Ss][Ii][Nn]
tan   [Tt][Aa][Nn]
exp   [Ee][Xx][Pp]
log   [Ll][Oo][Gg]
log10   {log}10
sqrt   [Ss][Qq][Rr][Tt]
ceil   [Cc][Ee][Ii][Ll]
```

An excerpt of y.output

```
    0  $accept : line $end
    1  line : expr '\n'
    2  $$1 :
    3  line : error '\n' $$1 line

    4  expr : '(' expr ')'
    5       | expr '^' expr
    6       | expr '*' expr
    7       | expr '/' expr
    8       | expr '+' expr
    9       | expr '-' expr
   10       | '-' expr
   11       | SIN expr
   12       | COS expr
   13       | TAN expr
   14       | INDEX

state 0
        $accept : . line $end  (0)
```

Closings

For more information:
contact me via Eidon at tutanota.com !

With thanks to Professor Till Tantau, whose “beamerexample-lecture” I modified here to create this presentation!