Software project
Gnome Graphics

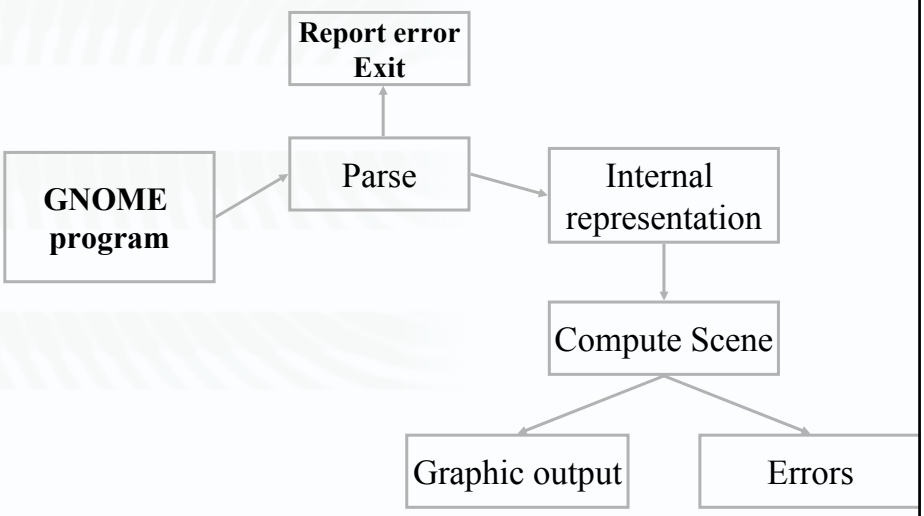
Olga Sorkine
sorkine@tau.ac.il

Andrei Scharf
asotzio@tau.ac.il

Office: Schreiber 002, 03-6405360

Web:
http://www.cs.tau.ac.il/~sorkine/courses/proj04/

The system

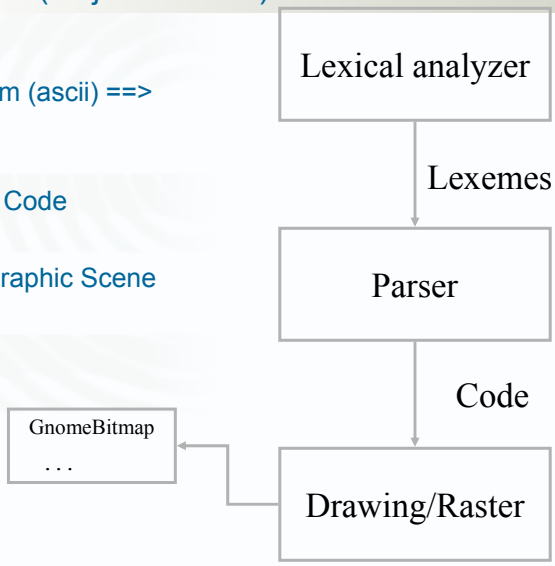


Design hints (major modules)

Lexical analyzer
- Input program (ascii) ==> tokens

Parser
- Tokens ==> Code

Raster
- Code ==> Graphic Scene



Lexical analyzer (translate)

Lex.prg

```
ROUTINE Draw10Segment($len)
{
  $x=$len*10
  Forward($x)
}
```

Tokens.h

```
#define ROUTINE_START 0
#define ROUTINE_NAME 1
#define OP_EQUALS 2
#define OP_TIMES 3
#define COMMAND 4
#define CURLY_OPEN 5
#define CURLY_CLOSE 6
#define VARIABLE 7
#define NUMBER 8
#define BRACK_OPEN 9
#define BRACK_CLOSE 10
```

Token Table

Lexeme	Value
ROUTINE_START	
ROUTINE_NAME	Draw10Segment
BRACK_OPEN	
VARIABLE	$len
BRACK_CLOSE	
CURLY_OPEN	
VARIABLE	$x
OP_EQUALS	
VARIABLE	$len
OP_TIMES	
NUMBER	10
COMMAND	Forward
BRACK_OPEN	
VARIABLE	$x
BRACK_CLOSE	
CURLY_CLOSE	

Parsing (validate)

- Input a "**tokens stream**"
- Examine the validity of the program
- Produce code

Parse.prg

```
ROUTINE ParseSample()
{
        $x=7
        $x2 = $x * $x
        $tmp = $x2 > 360
        Back(10)
        TurnLeft($x2)
}
```

Commands.h

```
#define Assign 1
#define AssignExp 2
#define Back 3
#define TurnLeft 4
#define Return 5
...
```

Parse Table

#	Command	P1	P2	P3	P4	Next
1	Assign	$x	7			2
2	AssignExp	TIMES	$x2	$x	$x	3
3	AssignExp	GREATER	$tmp	$x2	360	4
4	Back	10				5
5	TurnLeft	$x2				6
6	Return					-1

Parsing if

ParseIf.prg

```
ROUTINE ParseIfSample()
{
        $x=7
        $x2 = $x * $x
        $tmp = $x2 > 100
        if ($tmp)
          SetHeading(10)
          Forward($x)
        endif
}
```

#	Command	P1	P2	P3	P4	Next
1	Assign	$x	7			2
2	AssignExp	TIMES	$x2	$x	$x	3
3	AssignExp	GREATER	$tmp	$x2	100	4
4	If	$tmp	7			5
5	SetHeading	10				6
6	Forward	$x				7
7	return					-1

Parsing While

ParseWhile.prg

```
ROUTINE ParseSample()
{
        $x=7
        while ($x)
          $x=$x-1
        endwhile
}
```

#	Command	P1	P2	P3	P4	Next
1	Assign	$x	7			2
2	while	$x	4			3
3	AssignExp	MINUS	$x	$x	1	2
4	return					-1

Line number to go to if the expression is not true

Parsing function call

ParseFunc.prg

```
ROUTINE SqrLine($x)
{
        $y=$x*$x
        Forward($y)
}

ROUTINE ParseSample()
{
        $x=7
        SqrLine($x)
}
```

#	Command	P1	P2	P3	P4	Next
0	AssignExp	TIMES	$y	$x	$x	1
1	Forward	$y				2
2	Return					-1
3	Assign	$x	7			4
4	Call	0	Param_ptr			5
5	Assign	$y	$x			6
6	return					-1

Function	Line
Sqr	0
ParseSample	3

Line number where the routine's code starts

Pointer to linked list of parameters (or some other data structure of your choice

Software engineering

- Modularity
- Functionality
- Documentation
- Naming convention
- Data structures
- Design-Implement-Test

Functions

- Functions do one “thing”
- Repeated code is a function
- The name of the function should be self explanatory
- Functions are typically short < 60 lines
- Line length should be <= 80 characters

bad.c

```
void fndel(void* item)
{
   node* p = root;
   while (item != p->value)
                p = p->next;

        if (p != NULL)
        {
           p->next->prev = p->prev;
           p->prev->next = p->next;
           free(p);
        }
}
```

good.c

```
void RemoveItem(void* item)
{
        node* ItemsNode = Find(item);
        if (ItemsNode != NULL)
        {
                Delete(ItemsNode);
        }
}
```

Naming and Documentation

Document by writing “good” **code**

- Naming convention
- Meaningful names
- Simple and working code

Comments when appropriate

- Module/Class header
- Function header
- Special parts of code

documentation.c

```
/*
 * Description:
 *   Search the list for the
 *   input item and delete it
 *   if the item is in the list
 * Input:
 *   item - to look for and
 *          delete.
 * Remarks:
 *   May change root to NULL if
 *   the list contains one item.
 */
void RemoveItem(void* item)
{
        node* ItemsNode = Find(item)
        if (ItemsNode != NULL)
        {
                Delete(ItemsNode);
        }
}
```

Simple Code

Simple - working code is preferable.

- Less bugs
- Easy for others to read
- Easy to maintain

One exception:
The inner time-critical loop.

unreadable.c

```
void strcpy(char* d, char* s)
{
        for (;*d++=*s++;);
}
```

simple.c

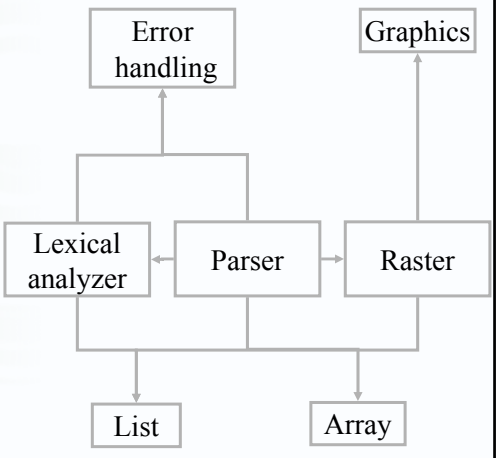
```
void strcpy(char* d, char* s)
{
        int i;

        for (i=0; s[i] != '\0'; ++i)
                d[i] = s[i];
        d[i] = '\0';
}
```

Modularity

A set of functions that manage one entity
- List (data structure)
- Parser

Purpose:
- Design tool, divide a difficult (large) problem to easier and smaller problems
- Distribute work among team members
- No code duplication
  - Less code
  - Fix a bug in one place
  - Ability to change design and improve incrementally
- Code re-use by writing standalone modules



Modularity

Implementation

1. .h module interface
   - Data structures of the module that the user uses
   - Function prototypes
   - Constants
2. .c implementation
   - Implementation of the functions that were defined in the .h
   - Prototypes, functions and constants that are internal to the module

Modularity - implementation

list.h

```
#ifndef __MYLIST_H
#define __MYLIST_H

typedef struct _List
{
          struct _List *next;
          void *data;
} List;

List* NewList();
void ListInsert(List* head, void
*item);

void ListDelete(List* head);

#endif /* __MYLIST_H */
```

list.c

```
#include "list.h"

/*
 * Allocates and prepares a new
 * list.
 * Return: NULL in case of an error
 *   or a pointer to the head of
 *   the list.
 */
List* NewList()
{
  List* new_list;
  new_list=malloc(sizeof(LIST));

  if (new_list == NULL)
    return NULL;

  new_list->next = NULL;
  new_list->data = NULL;
  return new_list;
}
```

Data Structures

Should be:

- Generic
- Handled through functions that manipulate them.
  - What happens if I need doubly linked-list?

ds.c

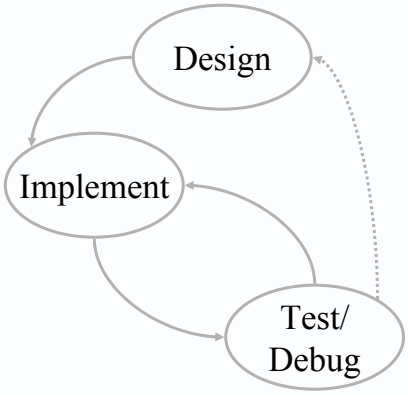
```
{
        Node* new;
...
        new = malloc(sizeof(Node));
        new->item = item;
        new->next = list->root
        list->root = new;
...
        new = malloc(sizeof(Node));
        new->item = item2;
        new->next = list->root
        list->root = new;
}
```

ds-function.c

```
{
...
        ListInsert(list, item);
...
        ListInsert(list, item2);
}
```

Design-Implement-Test

- Spend time on design before starting to code
- Leave enough time for debugging and unexpected design flaws



Suggested schedule

Learn the project	1w
Design	2w
Implement	5w
integration (port)	2w
Testing	3w
Write Documentation [3w]	In parallel

Total	13w

Asserts

array1.c

```
/*
 * Description:
 *   Set an array value
 * Input:
 *   array - pointer to the
 *      "array structure"
 *   item - to insert
 *
 * NOTES:
 *   assume the position is
 *   valid
 */
void Insert(ARRAY* a, int position,
    void* item)
{
    assert(position >= 0);
    assert(position < a->size);
    a->data[position] = item;
}
```

array2.c

```
/*
 * Description:
 *   Set an array value
 * Input:
 *   array - pointer to the array
 *           structure
 *   item - to insert
 *
 * NOTES:
 *   validate that the position
 *   is valid
 */
void Insert(ARRAY* a, int position, void*
item)
{
          if ( (position >= 0) &&
               (position < a->size) )
          {
                    a->data[position] = item;
          }
}
```

Asserts cont.

malloc1.c

```
/*
 * Description:
 *   Create a new array
 * Input:
 *   size - of the array
 * Return:
 *   new array
 * NOTES:
 *   assume malloc never fails
 */
ARRAY* CreateArray(int size)
{
          ARRAY* array;
          array = malloc(sizeof(ARRAY));
          assert(array); /* meaningless */
          array->size = size;
          array->data =
          malloc(sizeof(void*)*size);

          return array;
}
```

malloc2.c

```
/*
 * Description:
 *   Create a new array
 * Input:
 *   size - of the array
 * Return:
 *   new array, or NULL if failed.
 */
ARRAY* CreateArray(int size)
{
          ARRAY* array;
          array = malloc(sizeof(ARRAY));
          if (array == NULL) return NULL;
          array->data =
              malloc(sizeof(void*) * size);
          if (array->data == NULL) {
                    free(array);
                    return NULL;
          }
          return array;
}
```

How to use asserts

For every parameter of a function
- assert (parameter meets assumption)

For every call to a function
- assert(result meets assumptinos)

- Validate results of computations
- Validate **access** to NULL points
- Validate memory allocations
- Validate access to memory

Beware: asserts may be faulty themselves

Using global variables

FileManager.c

```
#include "FileManager.h"

FilesList* files;
...
```

Process.c

```
#include "FileManager.h"

{
...
        Write(files, 1, data);
...
}
```

FileManager.h

```
#ifndef __FILEMANAGER_H
#define __FILEMANAGER_H

typedef struct _FilesList
{
        . . .
} FilesList;

extern FilesList* files;
#endif /* __FILEMANAGER_H */
```

Common pitfalls

Plan your time
- Start working early

Test your project
- Try testing individual modules

Partners

Choose a partner you can work with

Meet every week or so
- See you're both on track
- Things change: synchronize

Unix tools

Editors:
- emacs
- vi
- pico (easier to learn)

Development environment
- kdevelop

Debuggers all based on gdb
- gdb: command line debuger
- ddd: gui to gdb

Help
- info or tkinfo
- man pages