Part 2:
Diablo Data Structures

- Goals
- Linker data structures
- Internal representation
- Construction of graphs
- Concrete Data Structures
- Manipulation
- Dynamic Members

Data Structures: Goal

Easy to manipulate

- ✔ CFG
- ✗ SSA

Reliable

- ▶ model control flow conservatively
- ◀ precise = not too conservative

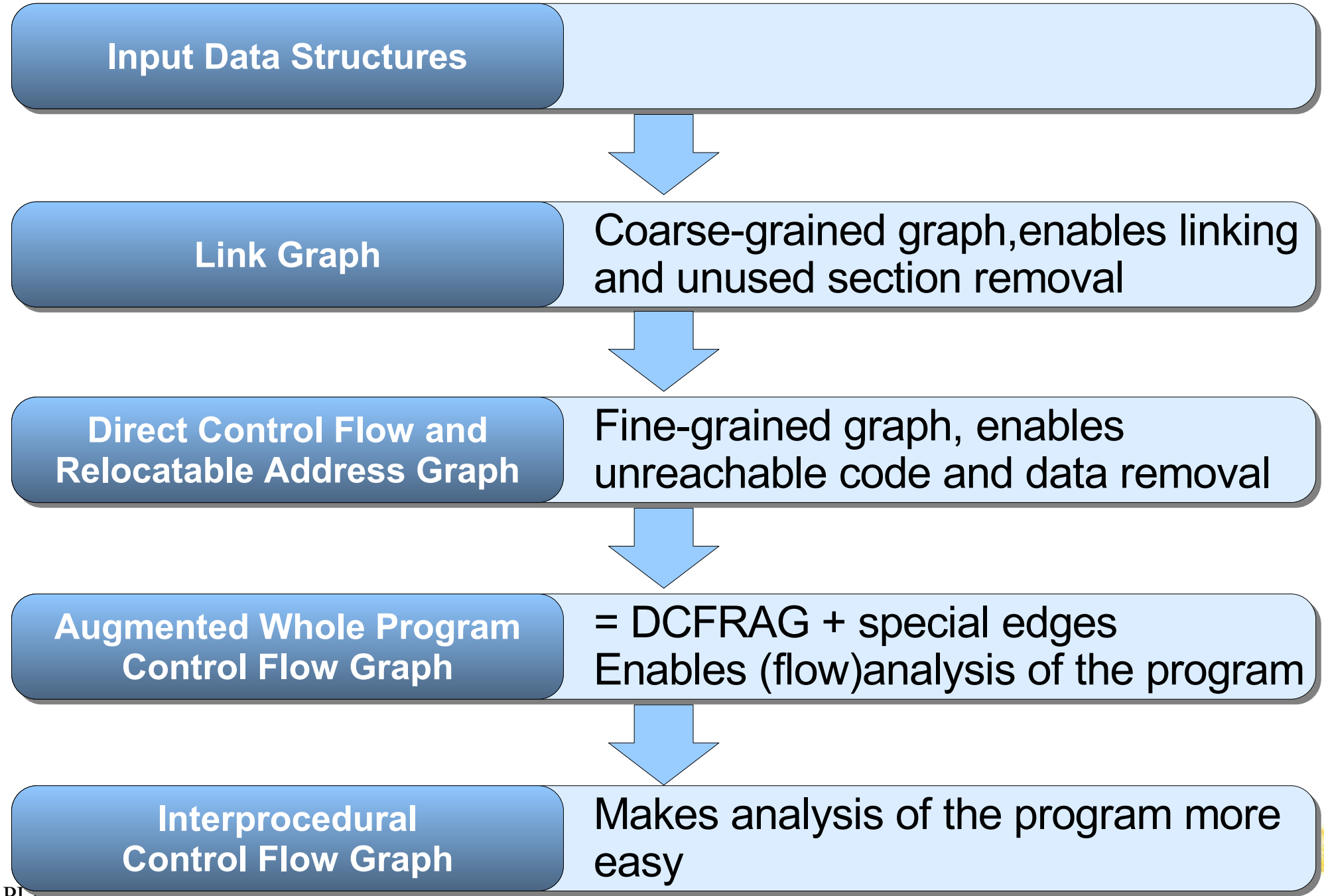
Retargetable

- ➔ abstract architecture specific details

Extensible

- ➔ easy to augment basic data structures with extra data

Transition in small steps



Part 2:
Diablo Data Structures

- Goals
- Linker data structures
- Internal representation
- Construction of graphs
- Concrete Data Structures
- Manipulation
- Dynamic Members

Linker Data Structures: our input

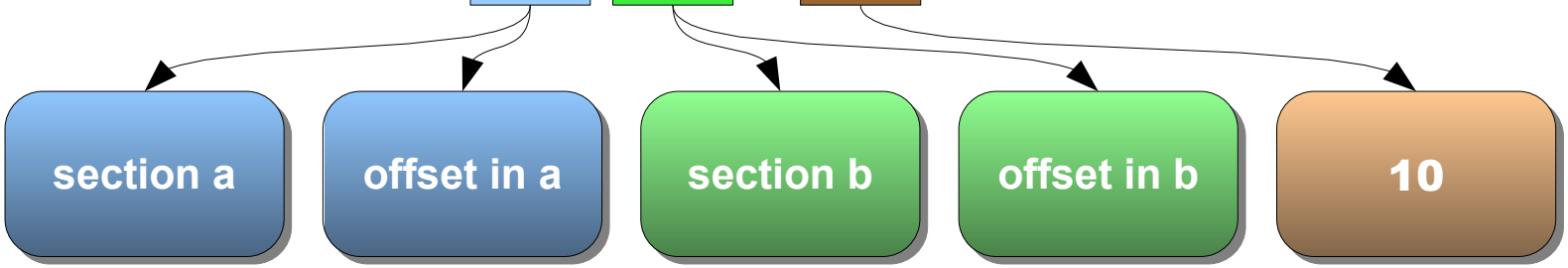
- Most are simply an abstract representation of data used by a linker:
  - archives (containers of relocatable objects)
  - relocatable objects (container of sections)
  - section (containers of data)
- Interesting structures for Diablo:
  - relocs (relocation information)
  - symbols

Relocatable objects

- Represent every entity that can need relocating
  - have an address and a size
  - can be used in symbols and relocs
  - can be changed by relocs
- e.g. sections are relocatable objects

Symbols

- labeled address expression
  - use the value of relocatable objects to calculate an address
  - example:
    - symbol “offset_between_section_a_and_b_plus_10”
    - code = “R01 R00 – A00 +$”



- used during symbol resolution
  - order
    - > means it overwrites other symbols
  - can create data (e.g. bss)

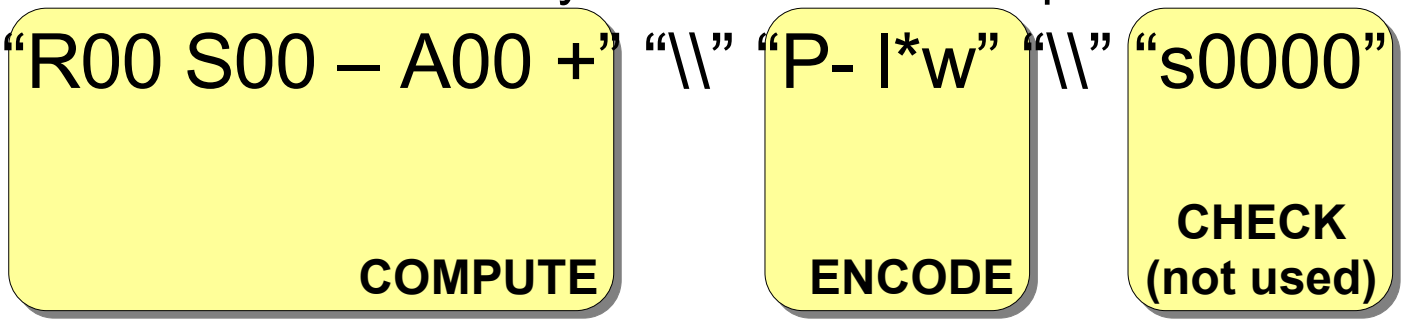
Relocations

- use the address of relocatable objects and symbols
- to compute an address
- put it in the desired encoding
- write it somewhere in a relocatable object
- checks if relocation was successful (no overflow)

Relocations

- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed

– “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”



Relocations

- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed
  - “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”

Relocations

- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed
  - “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”

Relocations

- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed

– “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”

Relocations

- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed
  - “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”

Relocations

- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed

– “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”

Relocations

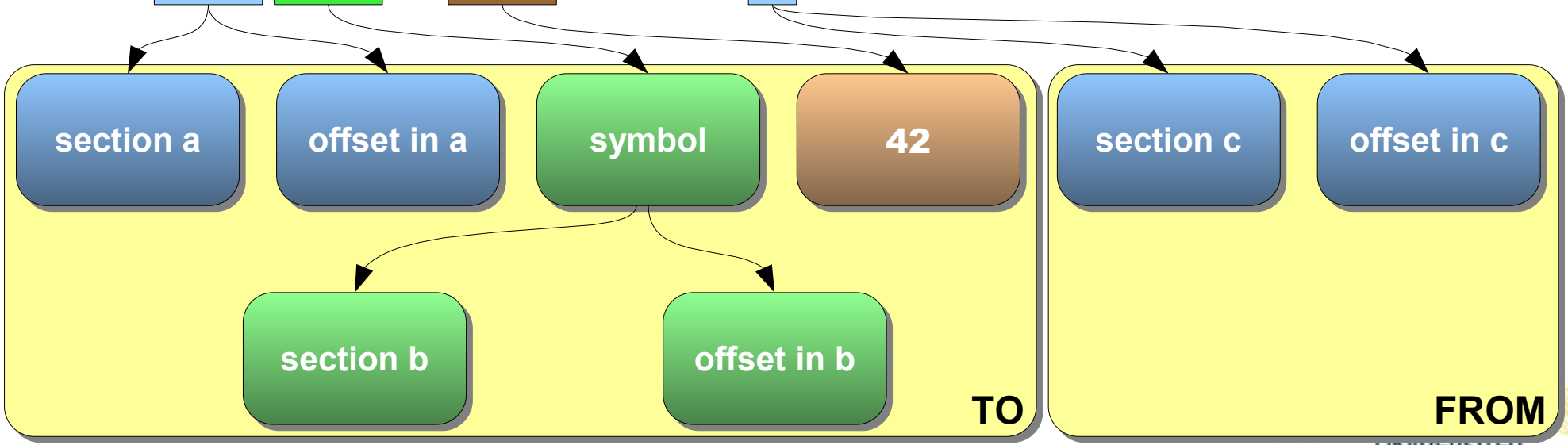
- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed

– “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”

Relocations

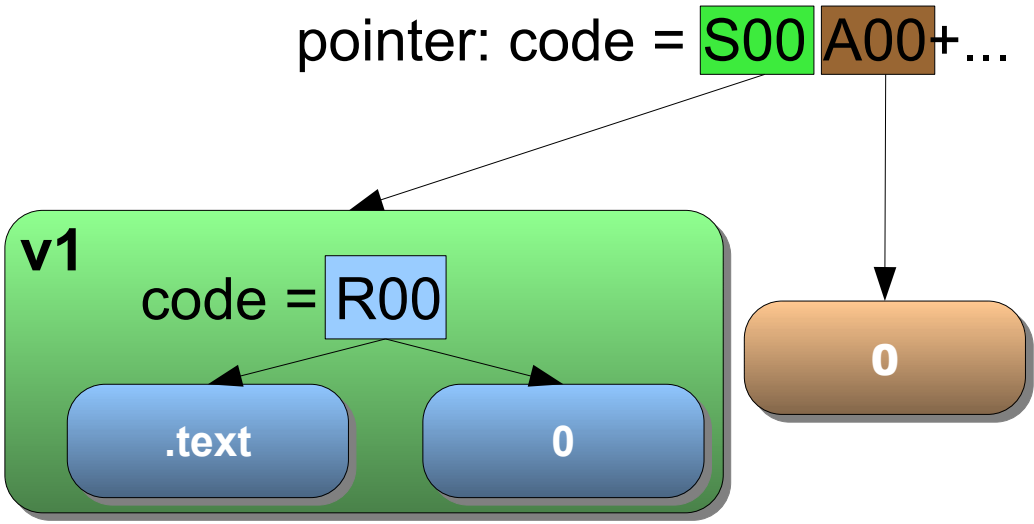
- Example
  - for an instruction “load immediate pc relative” that
    - loads the address of (an offset in) a relocatable object
    - minus the address of a symbol
    - plus some value (addend)
    - and stores this address pc relative (instruction encoding)
    - but automatically increases with the pc when executed

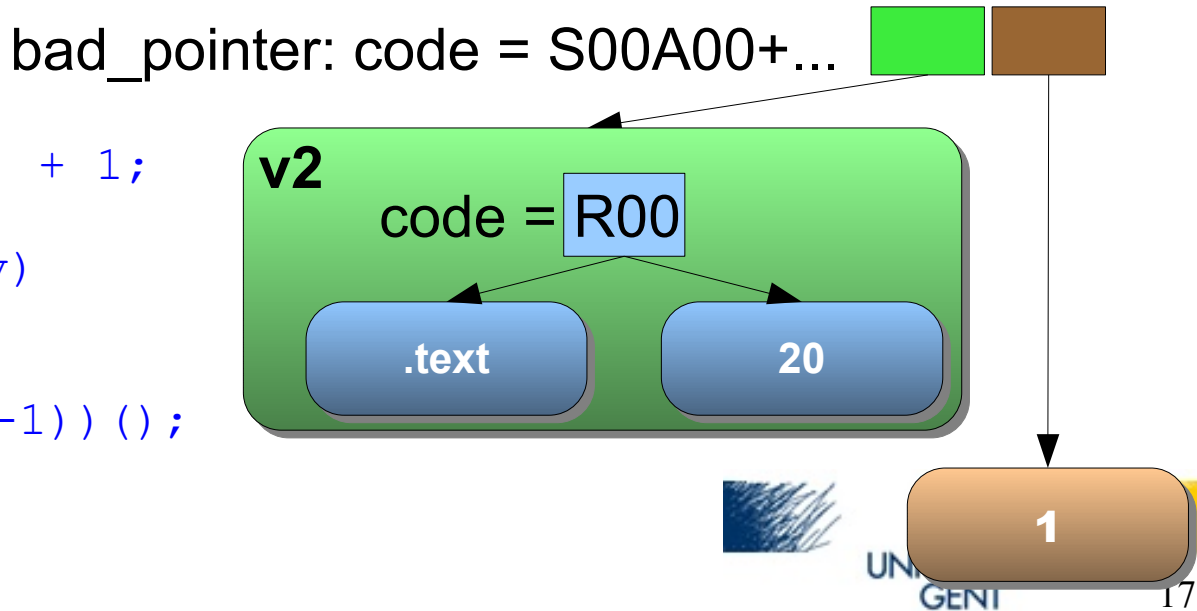
– “R00 S00 – A00 +” “\\” “P- I*w” “\\” “s0000” “$”



Relocation subtleties

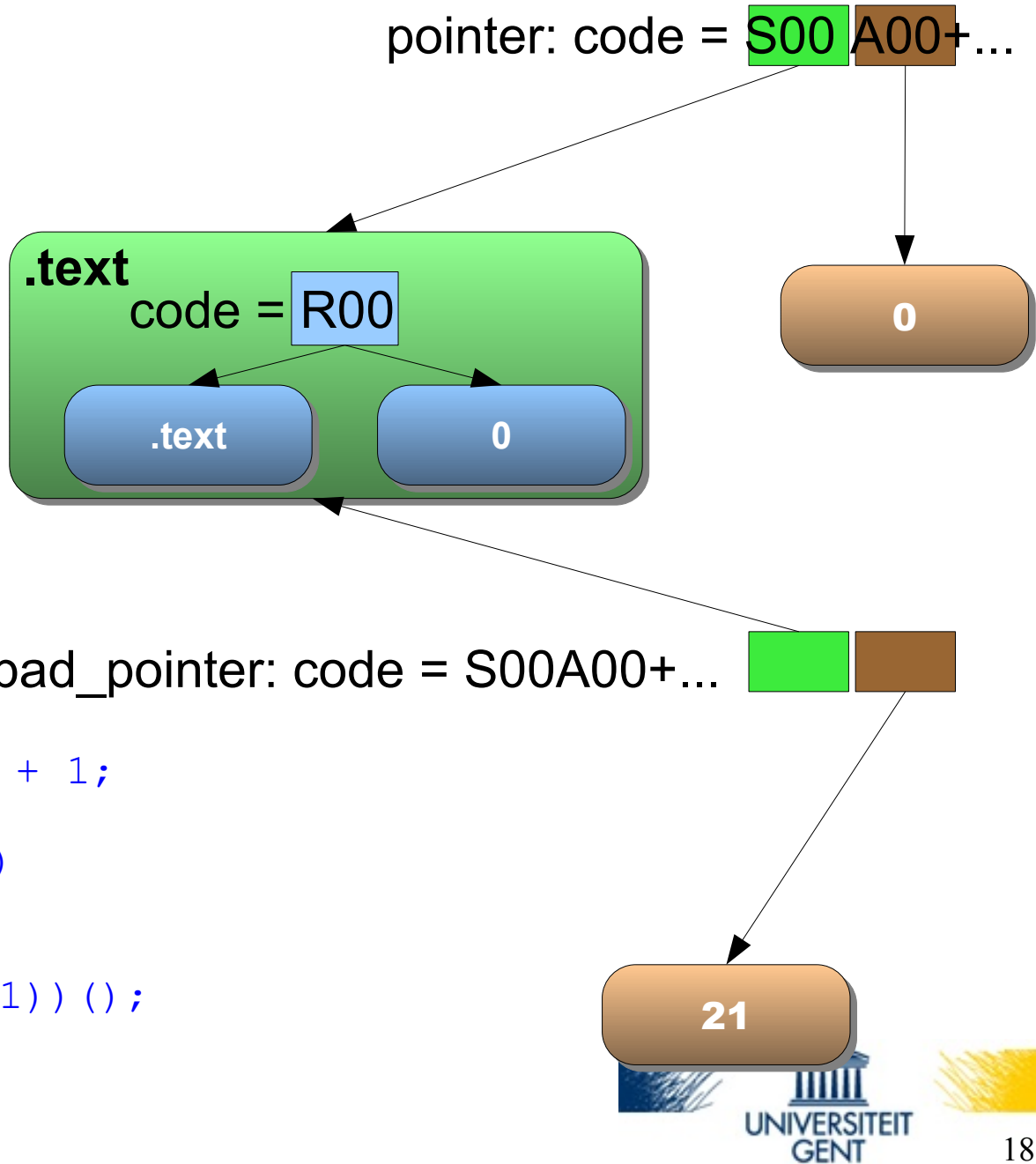
```
#include <stdio.h>

void v1()
{
    printf("v1\n");
}

void v2()
{
    printf("v2\n");
}

void (*pointer)() = v1;

int *bad_pointer = ((int *) v2) + 1;

int main(int argc, char ** argv)
{
    pointer();
    ((void (*)()) (bad_pointer -1))();
    return 0;
}
```





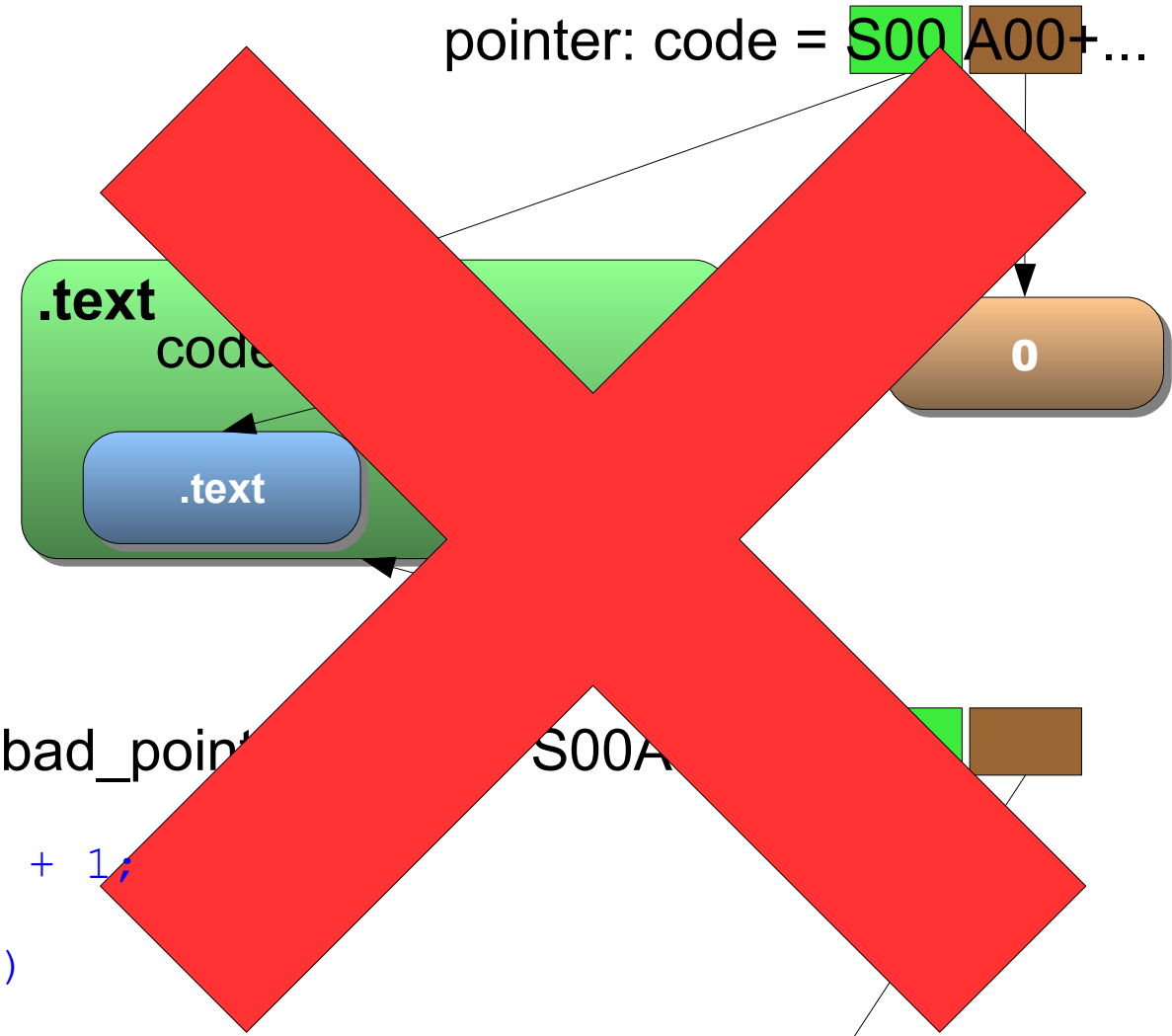
Relocation subtleties

```
#include <stdio.h>

void v1()
{
    printf("v1\n");
}

void v2()
{
    printf("v2\n");
}

void (*pointer)() = v1;

int *bad_pointer = ((int *) v2) + 1;

int main(int argc, char ** argv)
{
    pointer();
    ((void (*)()) (bad_pointer -1))();
    return 0;
}
```



Relocation subtleties

```
#include <stdio.h>

void v1()
{
    printf("v1\n");
}

void v2()
{
    printf("v2\n");
}

void (*pointer)() = v1;

int *bad_pointer = ((int *) v2) + 1;

int main(int argc, char ** argv)
{
    pointer();
    ((void (*)()) (k
    return 0;
}
```

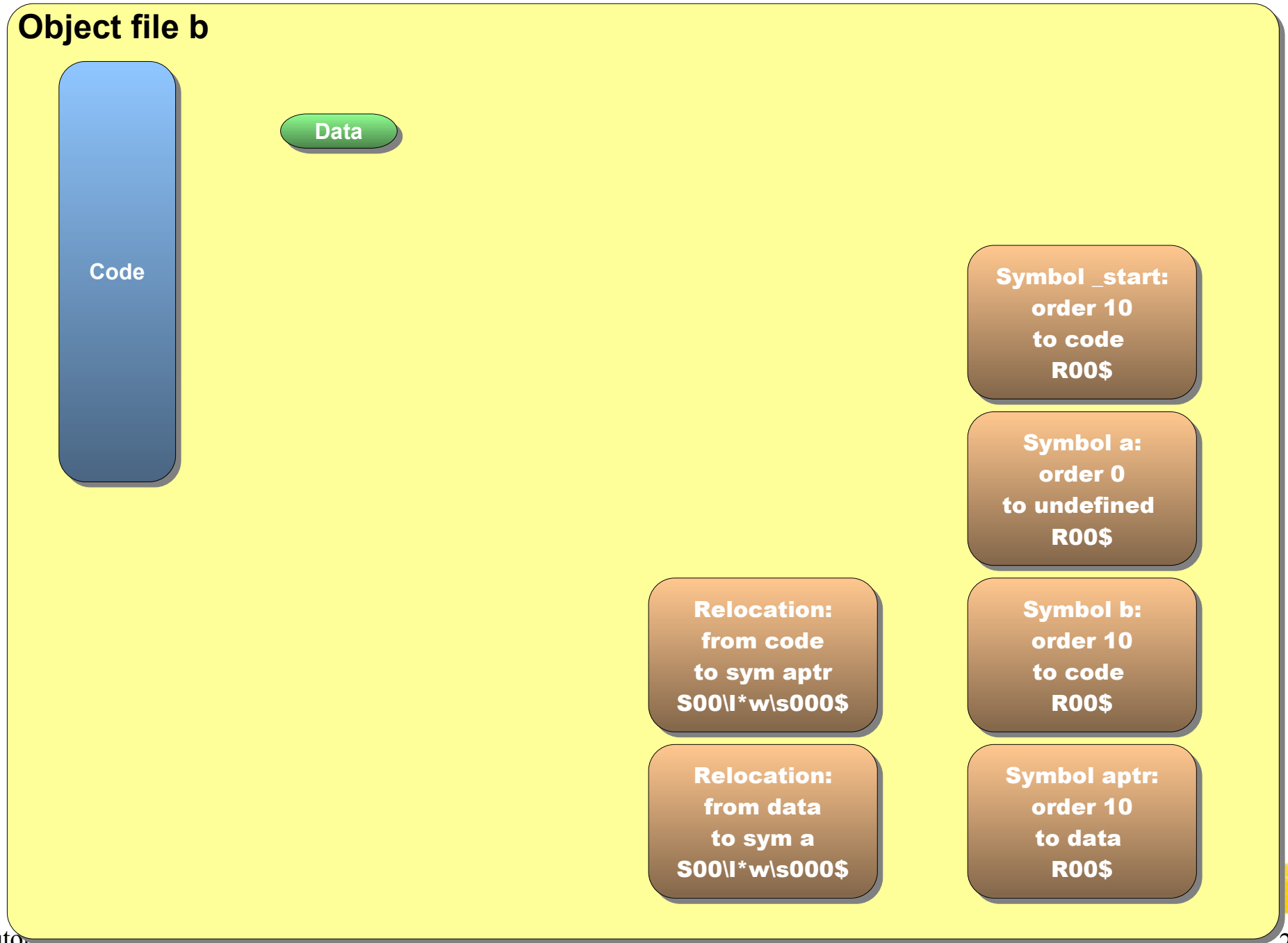


RELOCATIONS & SYMBOLS SHOULD NOT BE RELAXED

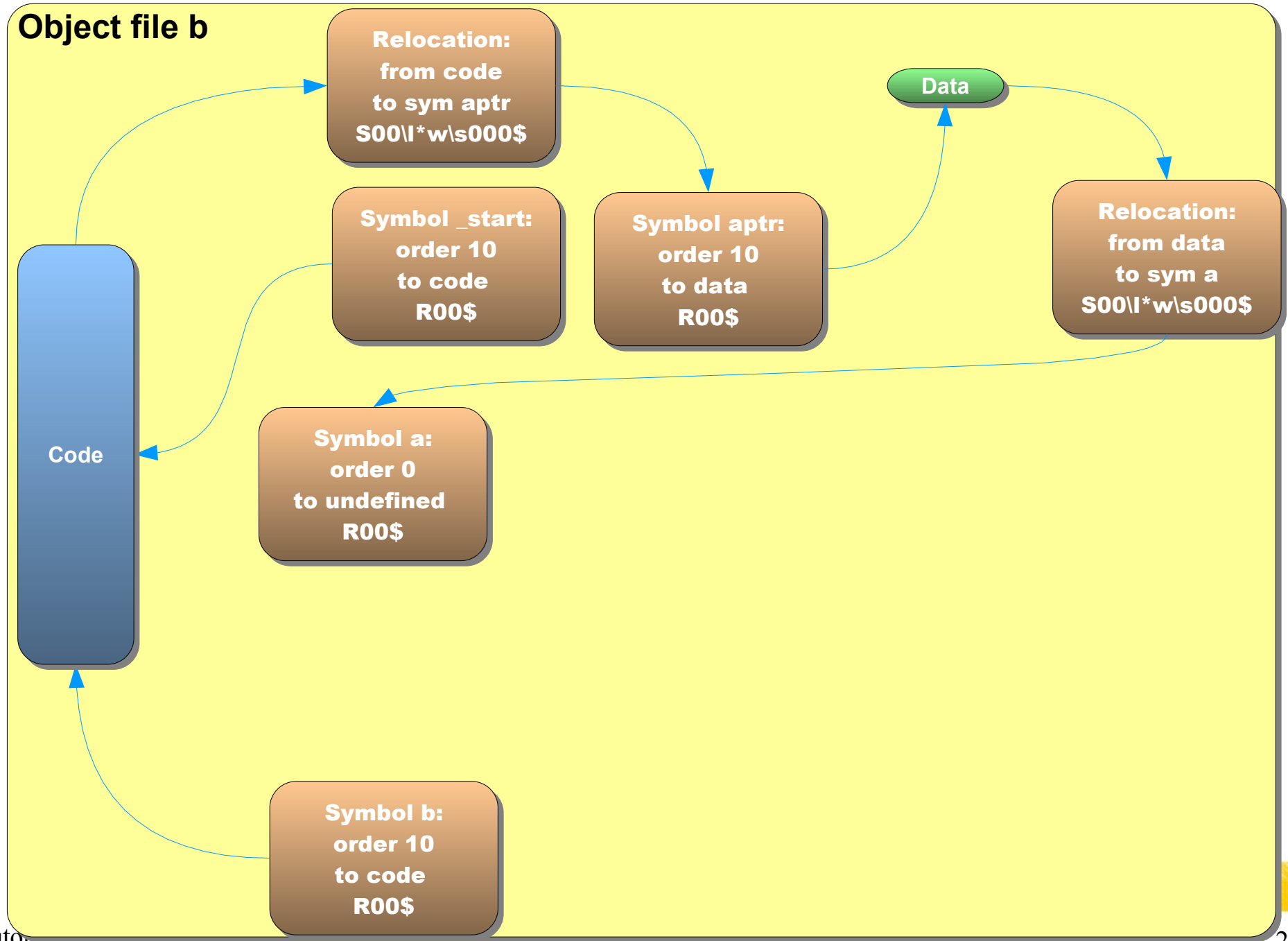
Part 2:
Diablo Data Structures

- Goals
- Linker data structures
- Internal representation
- Construction of graphs
- Concrete Data Structures
- Manipulation
- Dynamic Members

Object files



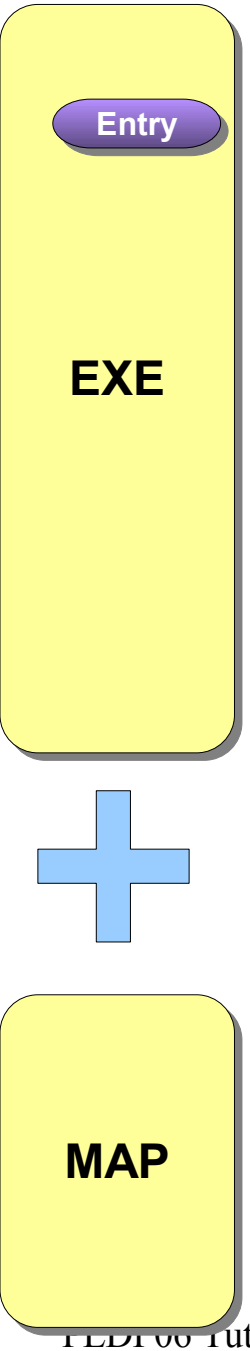
Graph representation



Part 2:
Diablo Data Structures

- Goals
- Linker data structures
- Internal representation
- Construction of graphs
- Concrete Data Structures
- Manipulation
- Dynamic Members

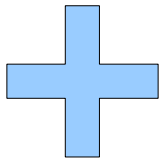
Reading information



Reading information

Entry

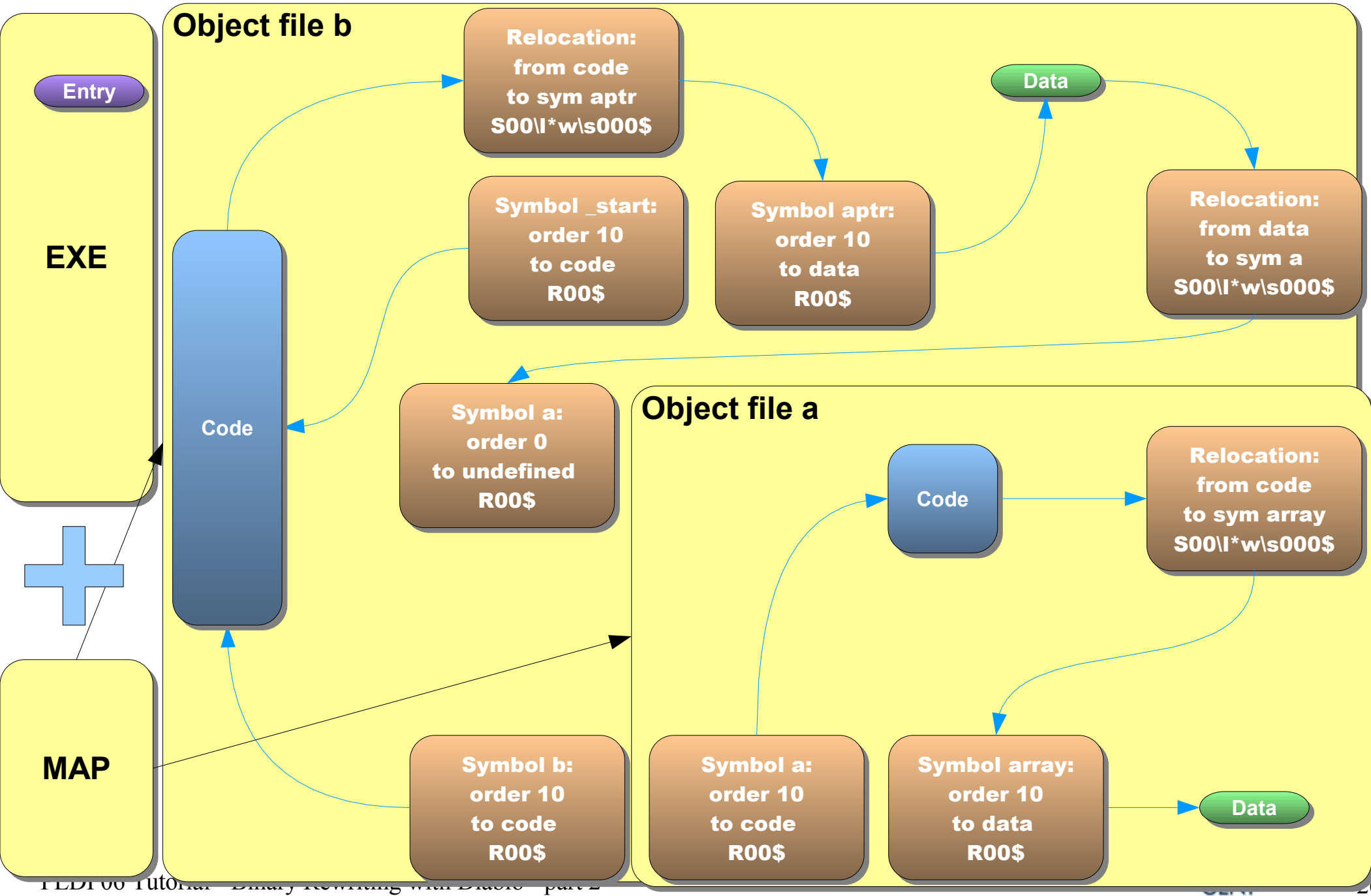
EXE



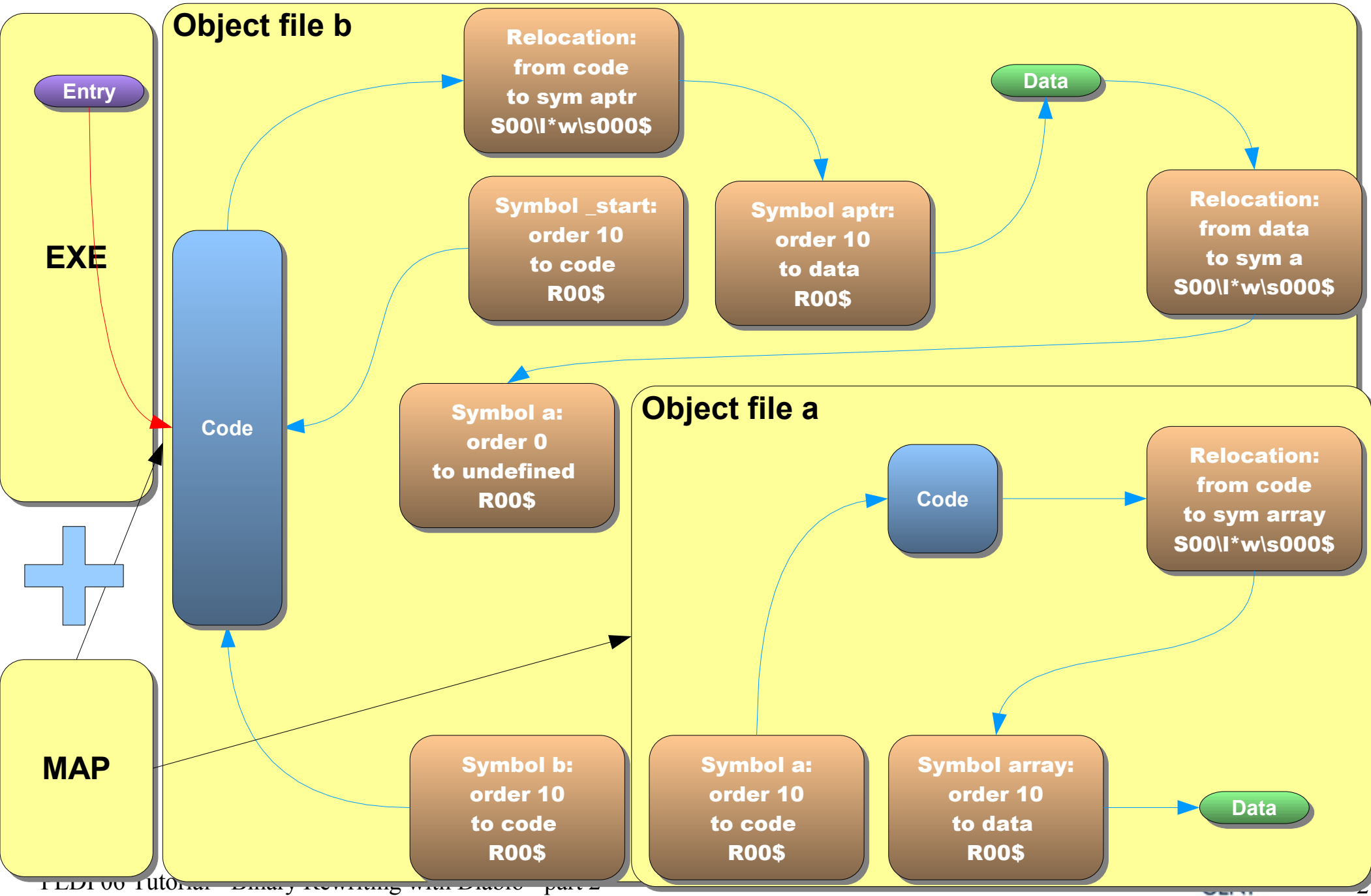
MAP

```
.text              0x8048120   0x5e4d40

 .text             0x8048120   0x24 /usr/lib/crt1.o
                   0x8048120                _start
 .text             0x8048144   0x22 /usr/lib/crti.o
 *fill*            0x8048166
 .text             0x8048170   0xc4 /usr/lib/crtbeginT.o
 *fill*            0x8048234
 .text             0x8048240   0x56f app_procs.o
                   0x8048300                app_run
                   0x80482a0                app_abort
                   0x80482e0                app_exit
                   0x8048240                app_libs_init
 *fill*            0x80487af
 .text             0x80487b0   0xf33 main.o
                   0x80487b0                main
```

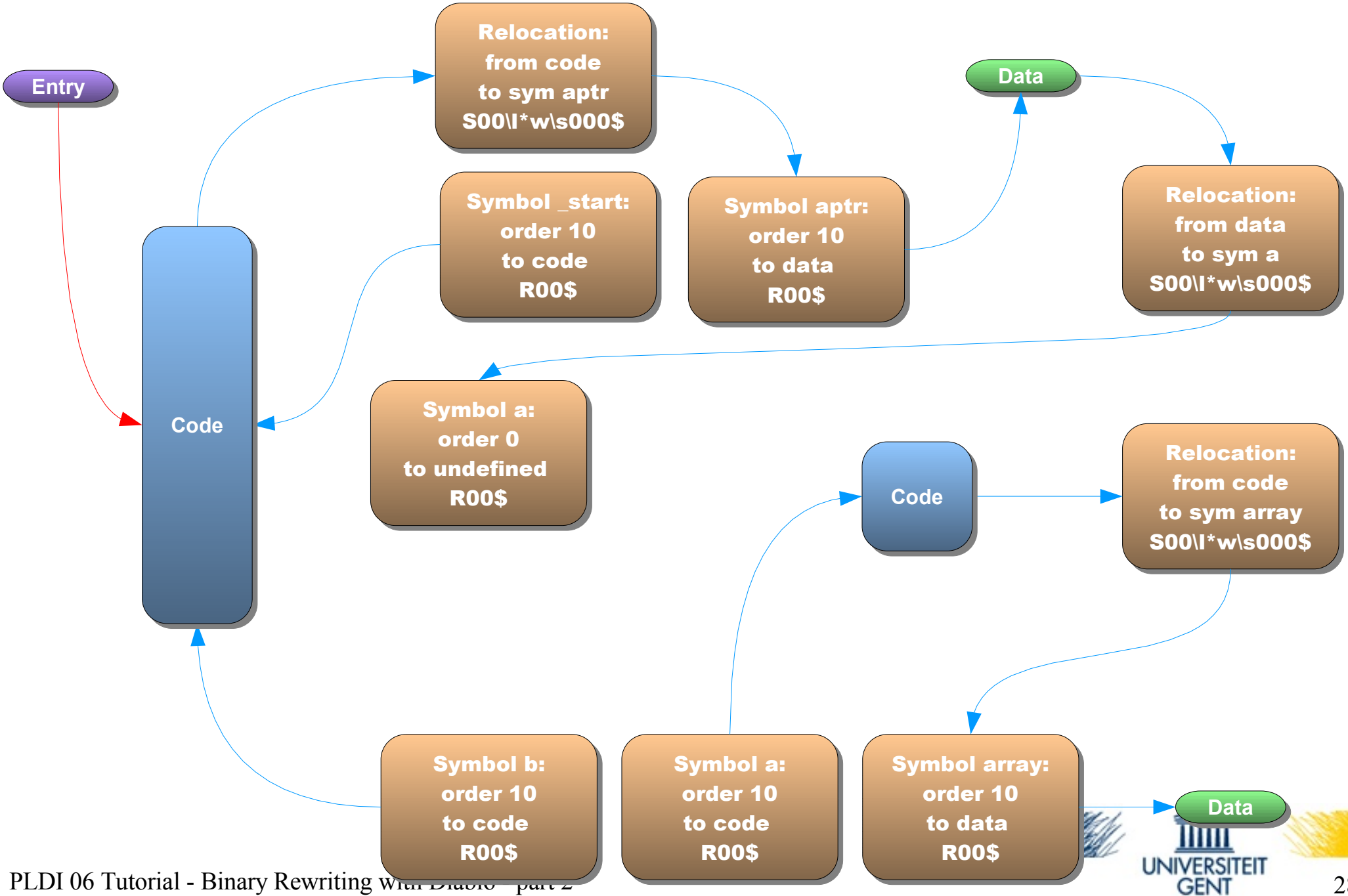
Reading information



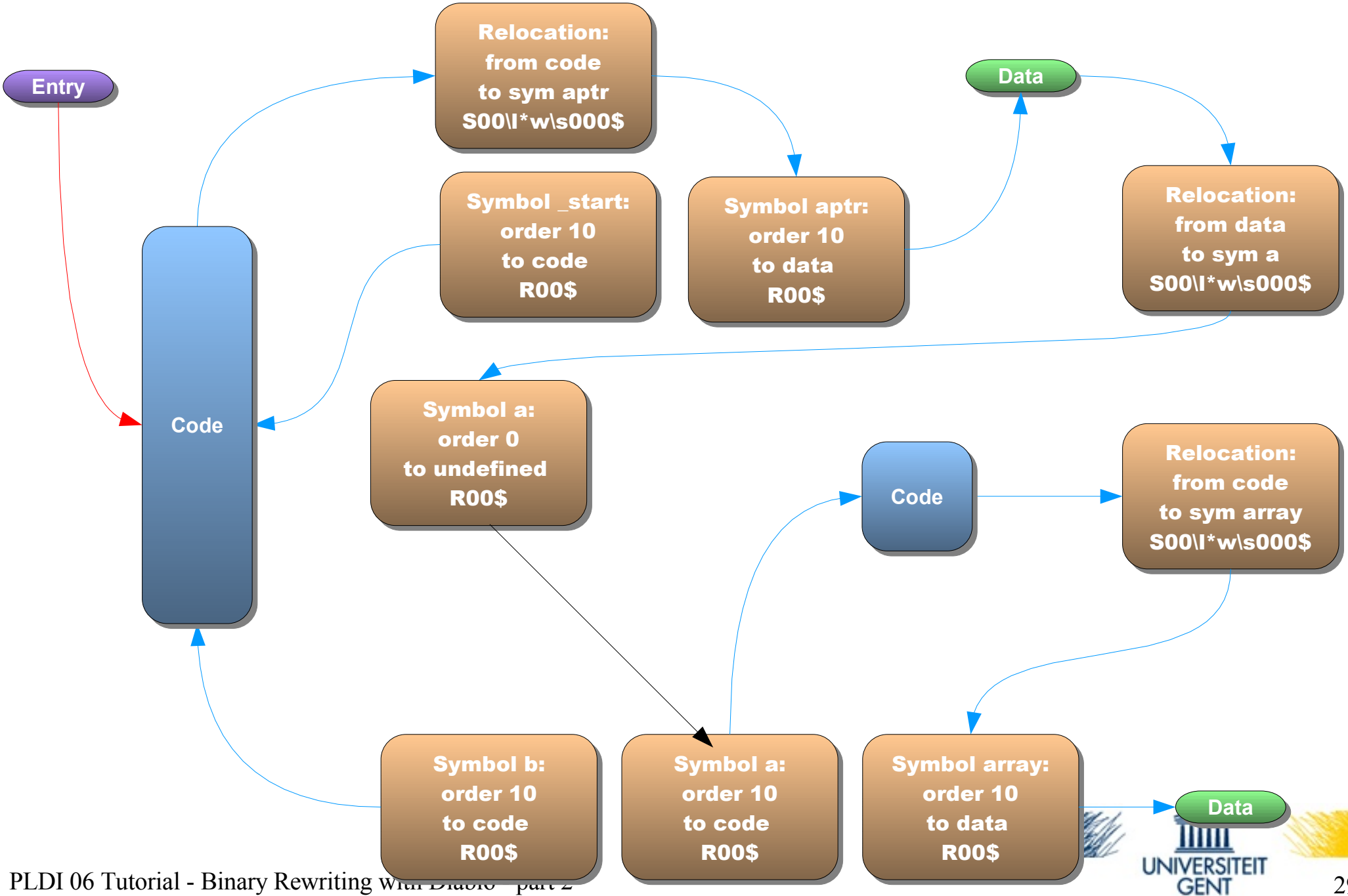
Reading information



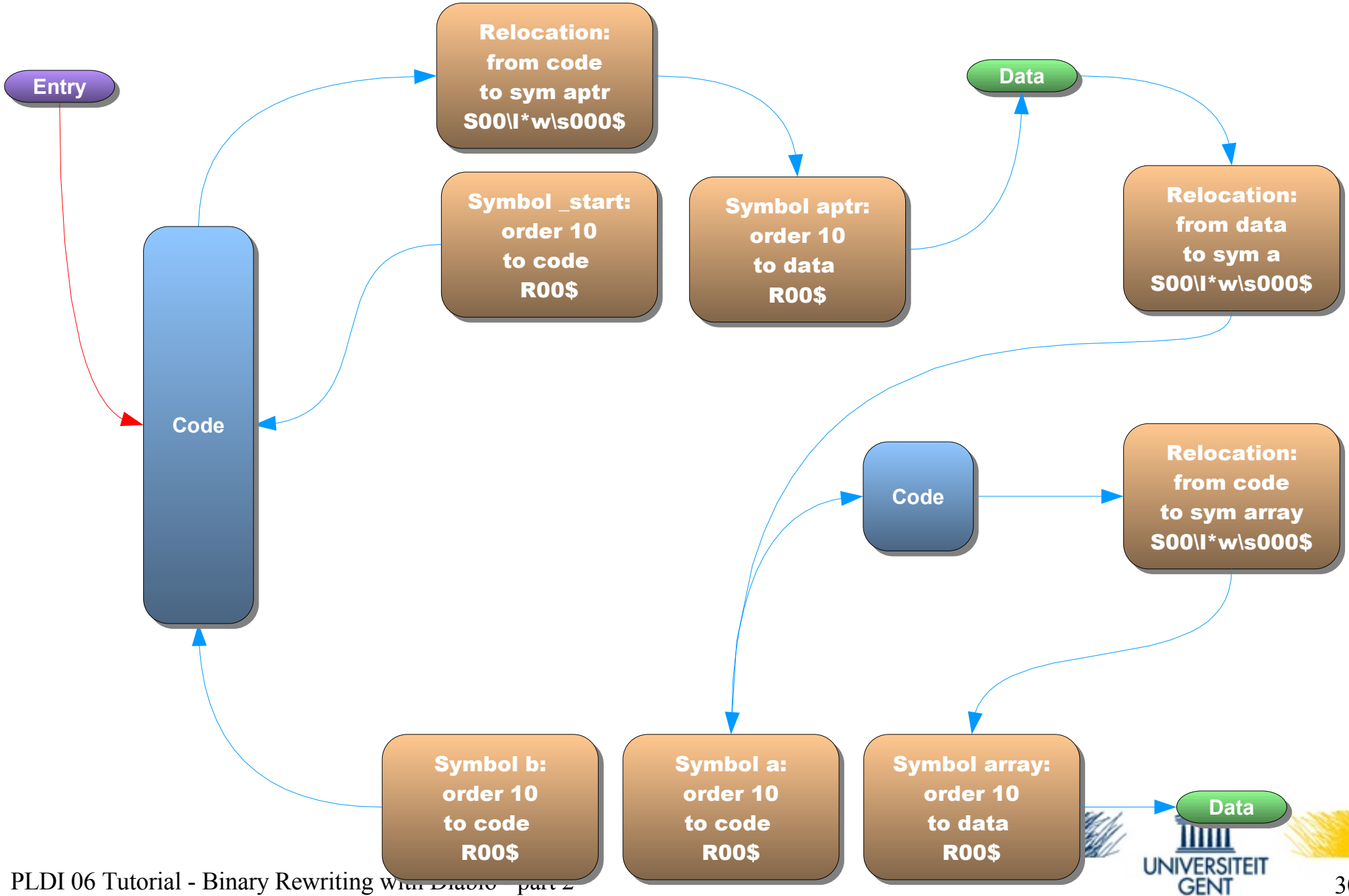
Reading information



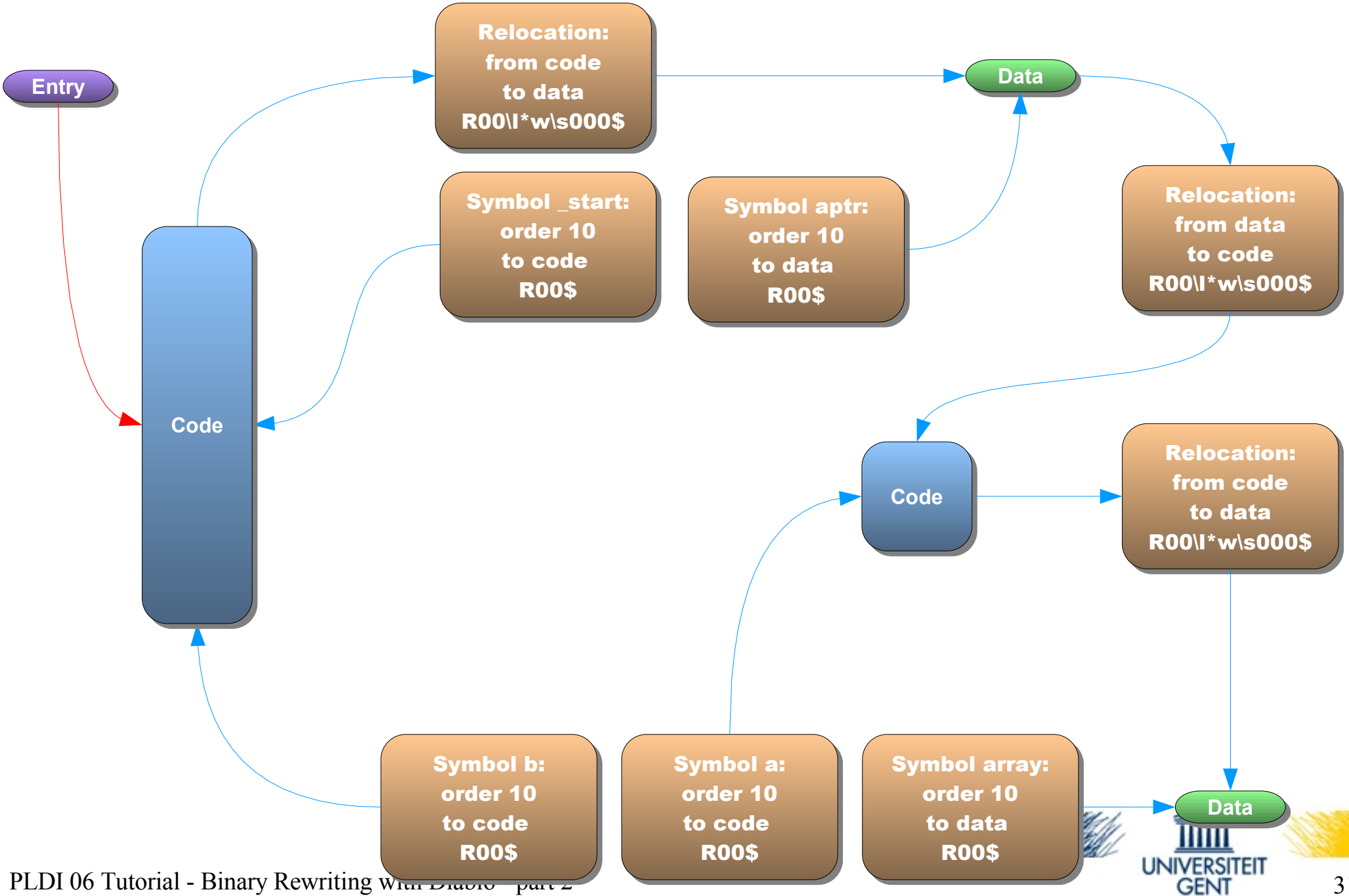
Symbol Resolution



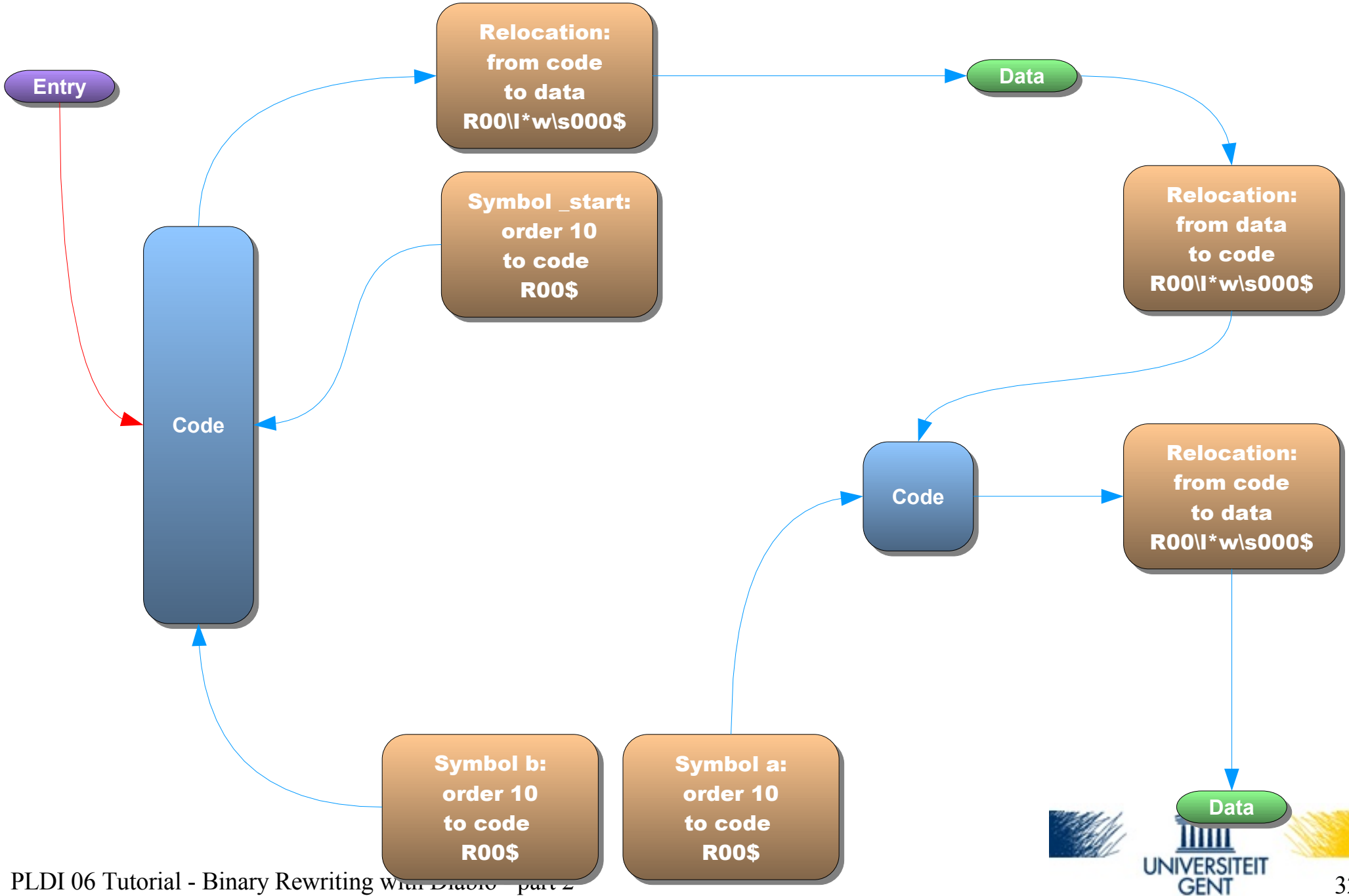
Symbol Resolution



Linkgraph


Entry
Relocation:
from code
to data
R00\l*w\s000$
Data
Symbol _start:
order 10
to code
R00$
Symbol aptr:
order 10
to data
R00$
Relocation:
from data
to code
R00\l*w\s000$
Code
Code
Relocation:
from code
to data
R00\l*w\s000$
Symbol b:
order 10
to code
R00$
Symbol a:
order 10
to code
R00$
Symbol array:
order 10
to data
R00$
Data

Linkgraph


Entry
Relocation:
from code
to data
R00\l*w\s000$
Data
Relocation:
from data
to code
R00\l*w\s000$
Symbol _start:
order 10
to code
R00$
Code
Code
Relocation:
from code
to data
R00\l*w\s000$
Symbol b:
order 10
to code
R00$
Symbol a:
order 10
to code
R00$
Data

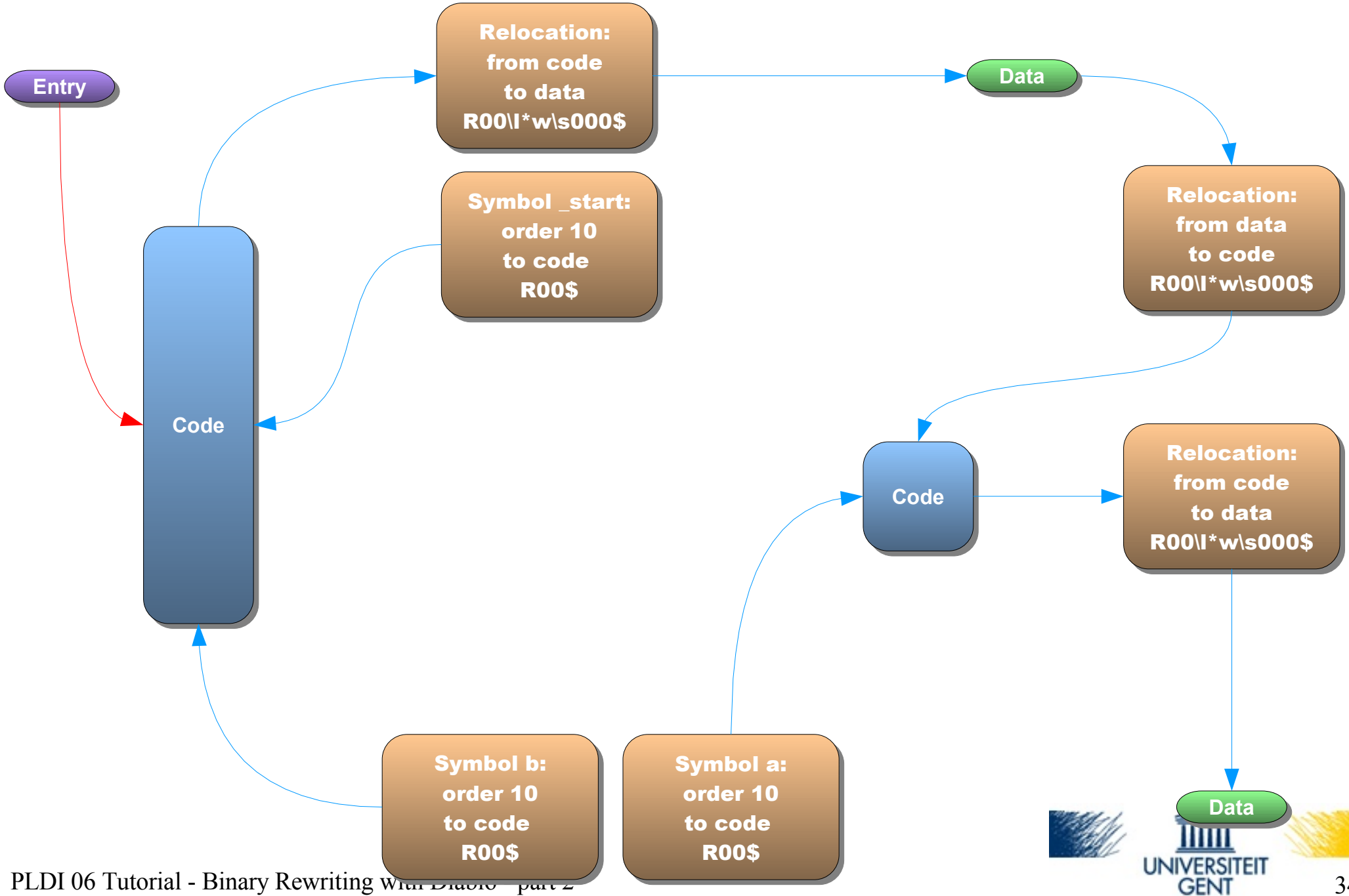
Uses of the linkgraph

- ✔ Linking (placing sections, relocating)
- ✔ Apply linker optimizations (remove unused sections)

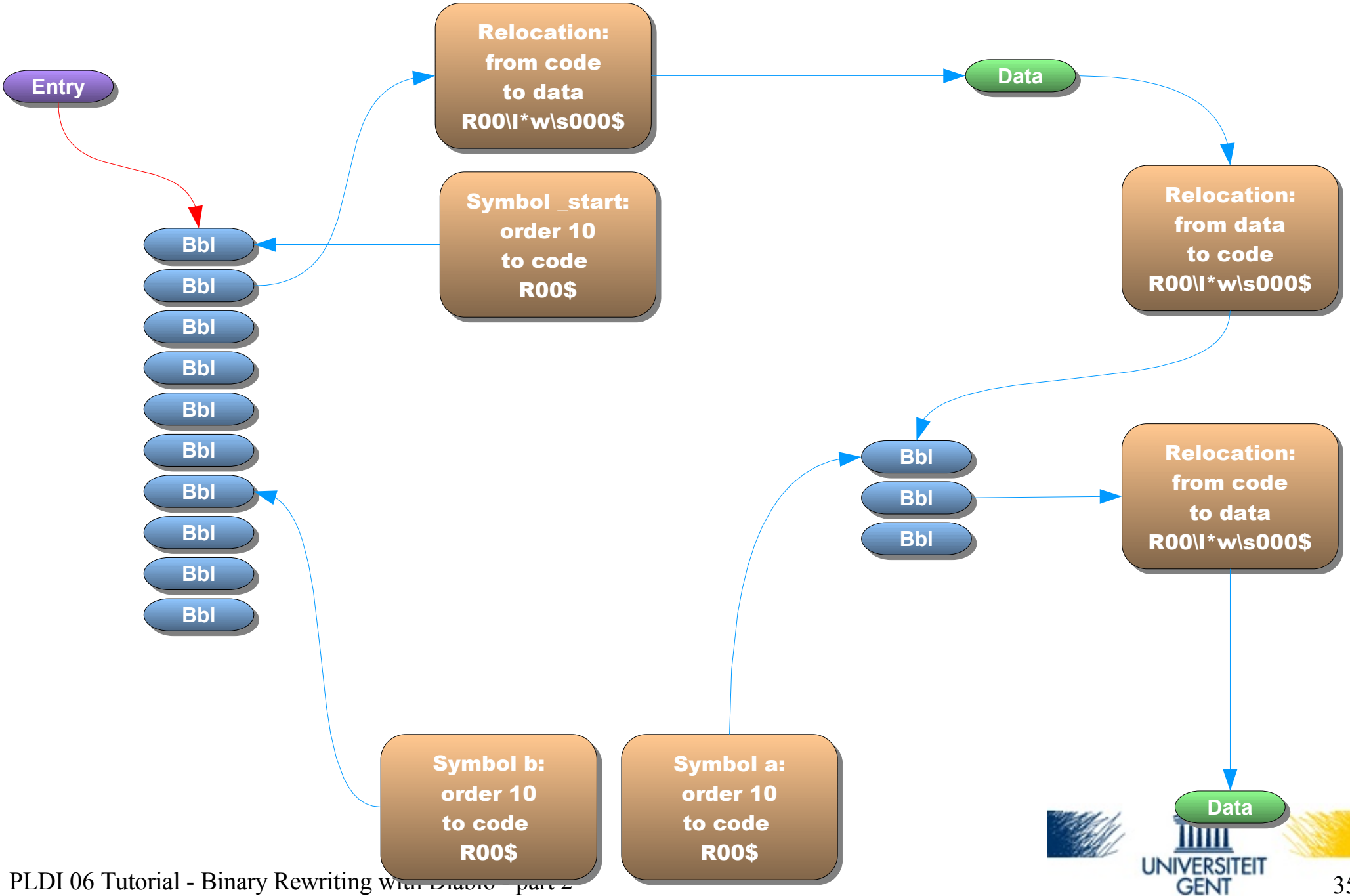
- ✗ fine-grained transformations

- ➔ Need for a more fine-grained graph: DCFRAG Direct Control Flow and Relocatable Addresses

Linkgraph

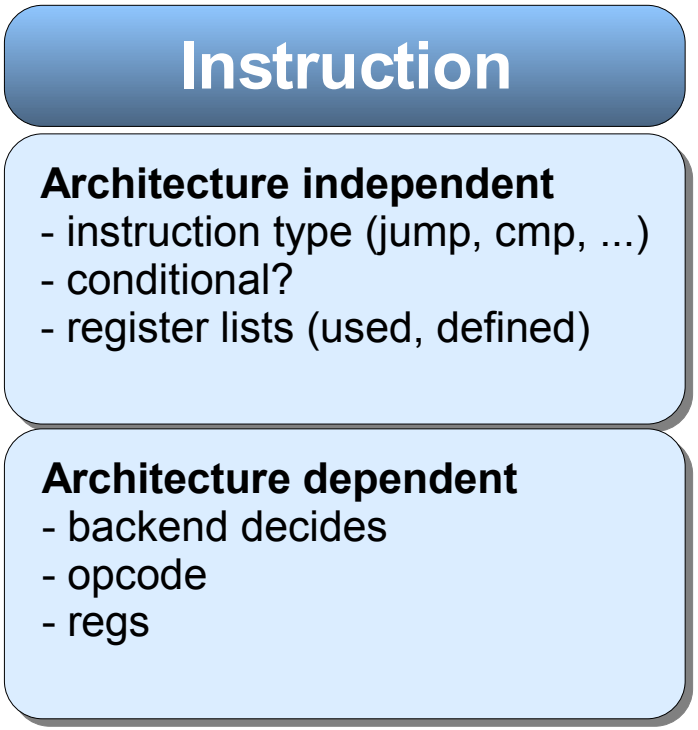

Entry
Relocation:
from code
to data
R00\l*w\s000$
Data
Relocation:
from data
to code
R00\l*w\s000$
Symbol _start:
order 10
to code
R00$
Code
Code
Relocation:
from code
to data
R00\l*w\s000$
Data
Symbol b:
order 10
to code
R00$
Symbol a:
order 10
to code
R00$

Disassembler



Disassembler

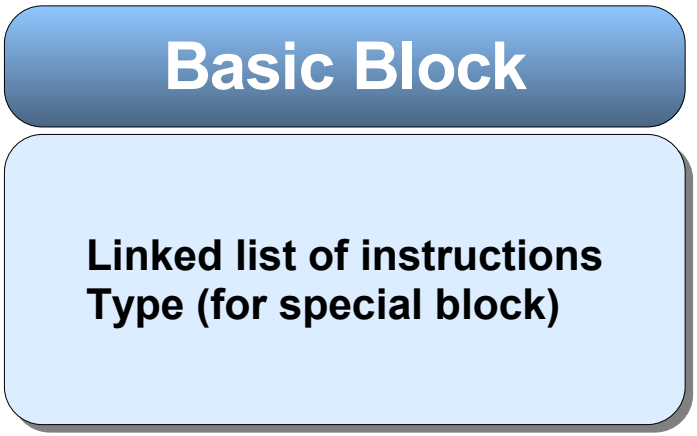
- Disassemble



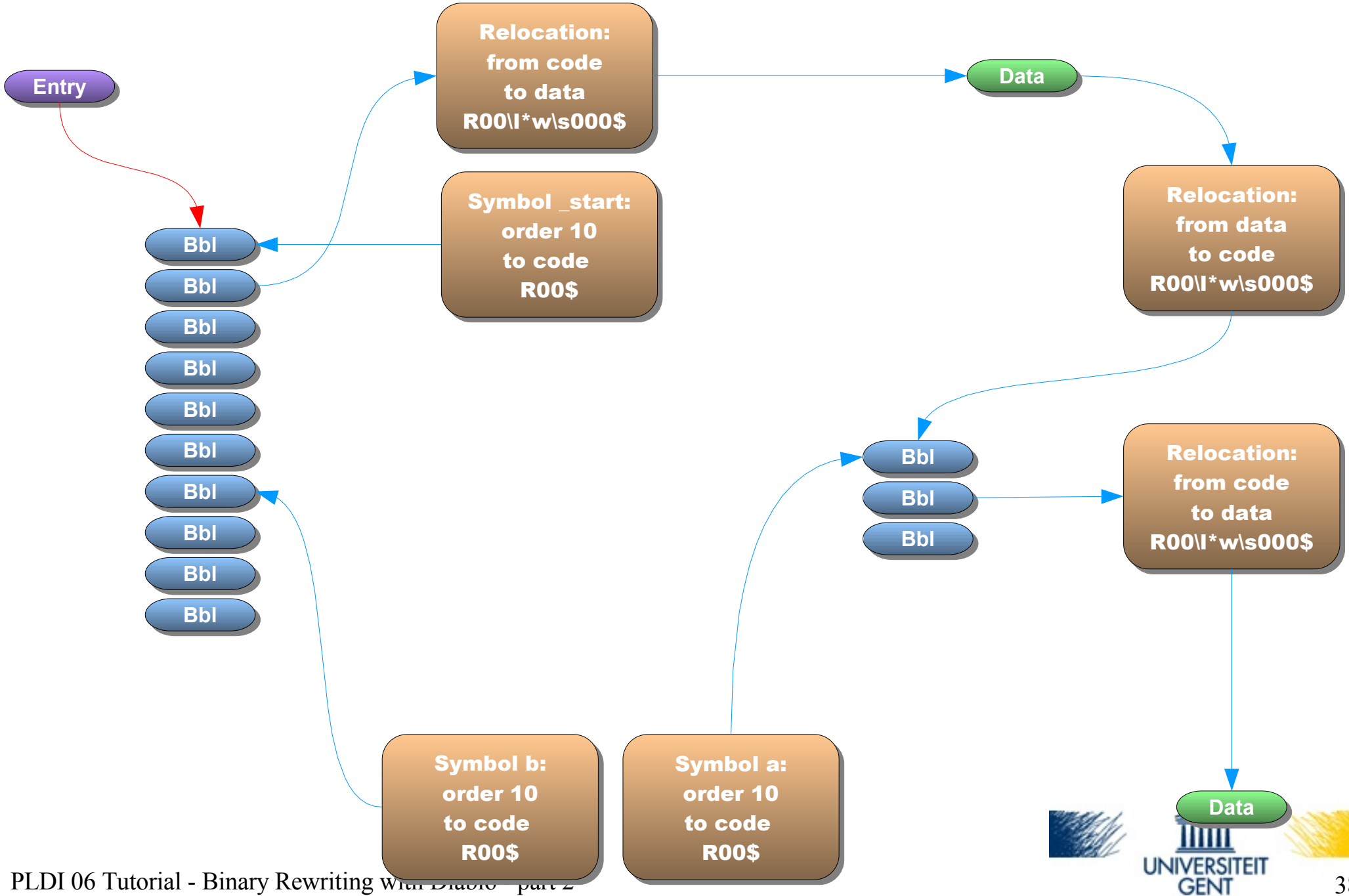
  - Architecture independent analyses and optimizations work on architecture independent part or use callbacks

Detect basic blocks

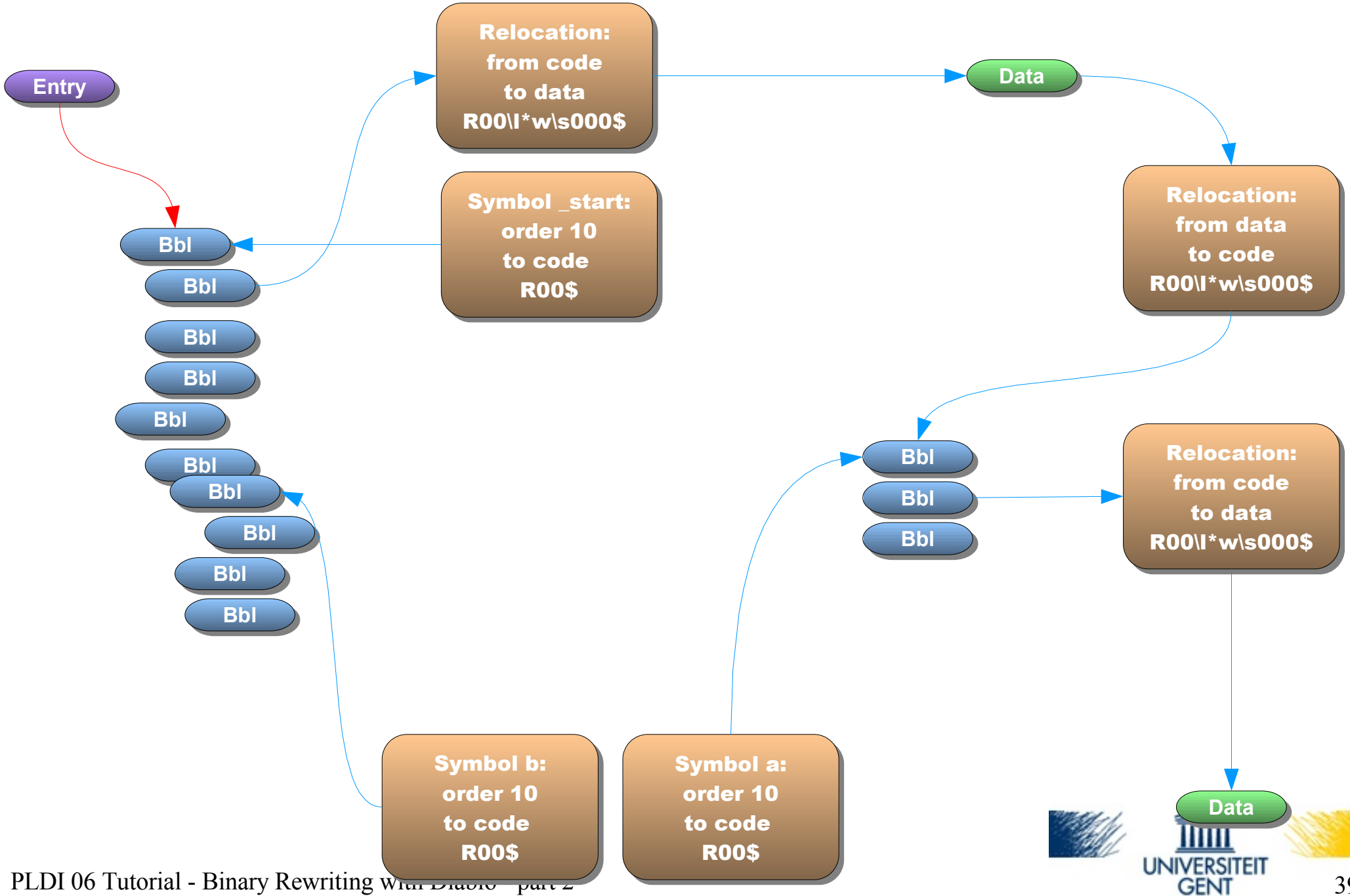
- Disassemble
- Split sections into basic blocks



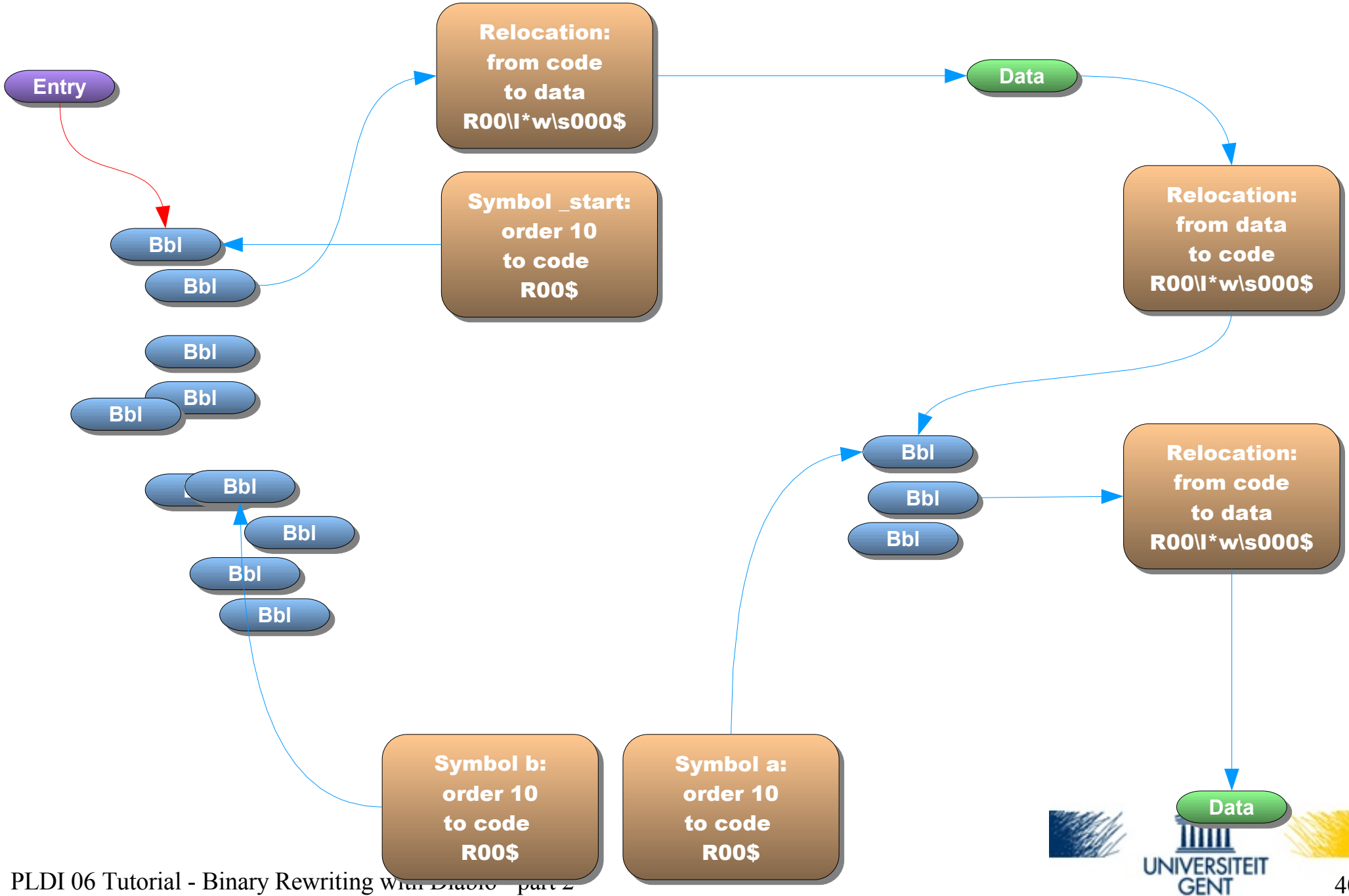
Detect basic blocks



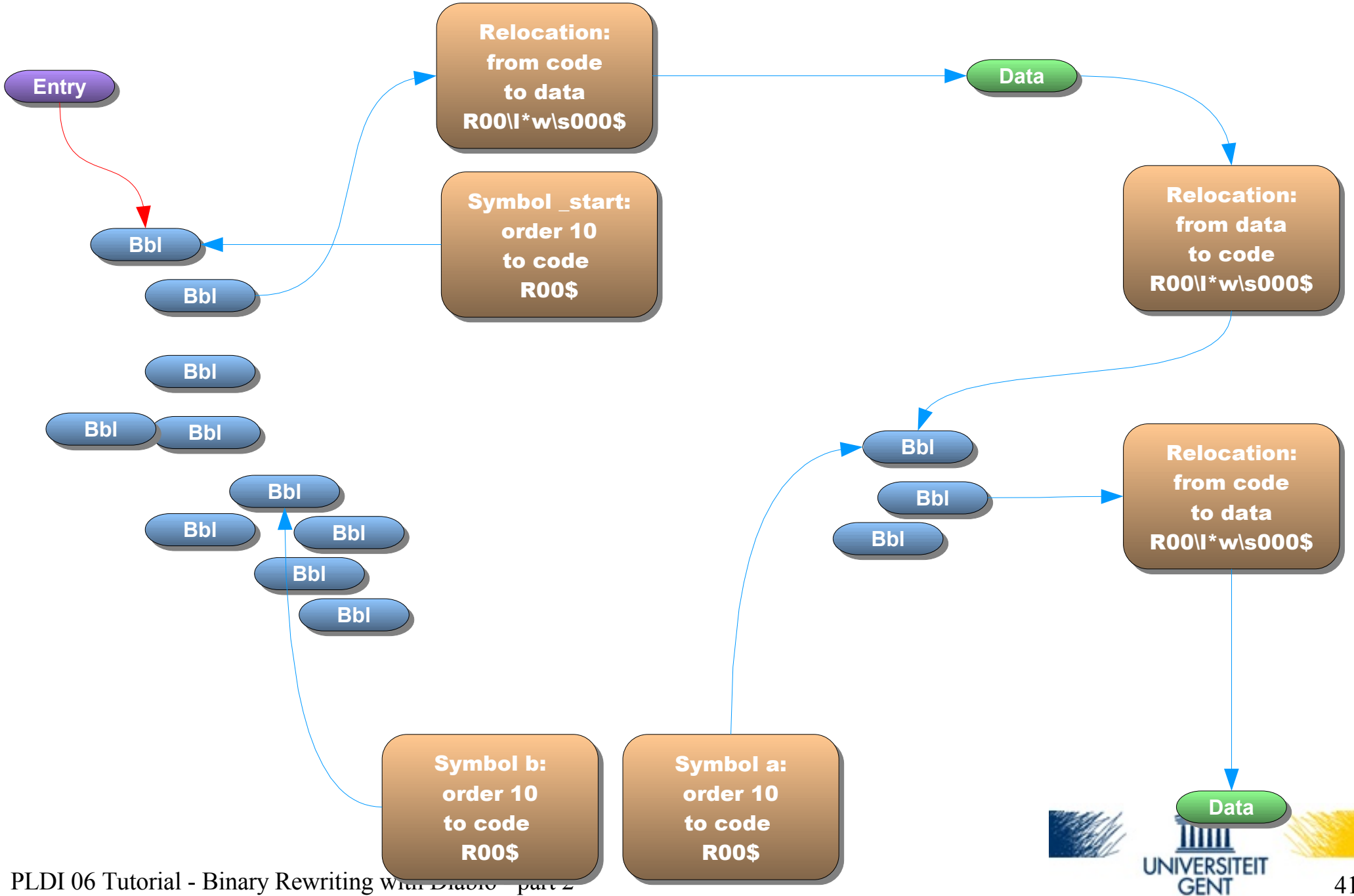
Detect basic blocks



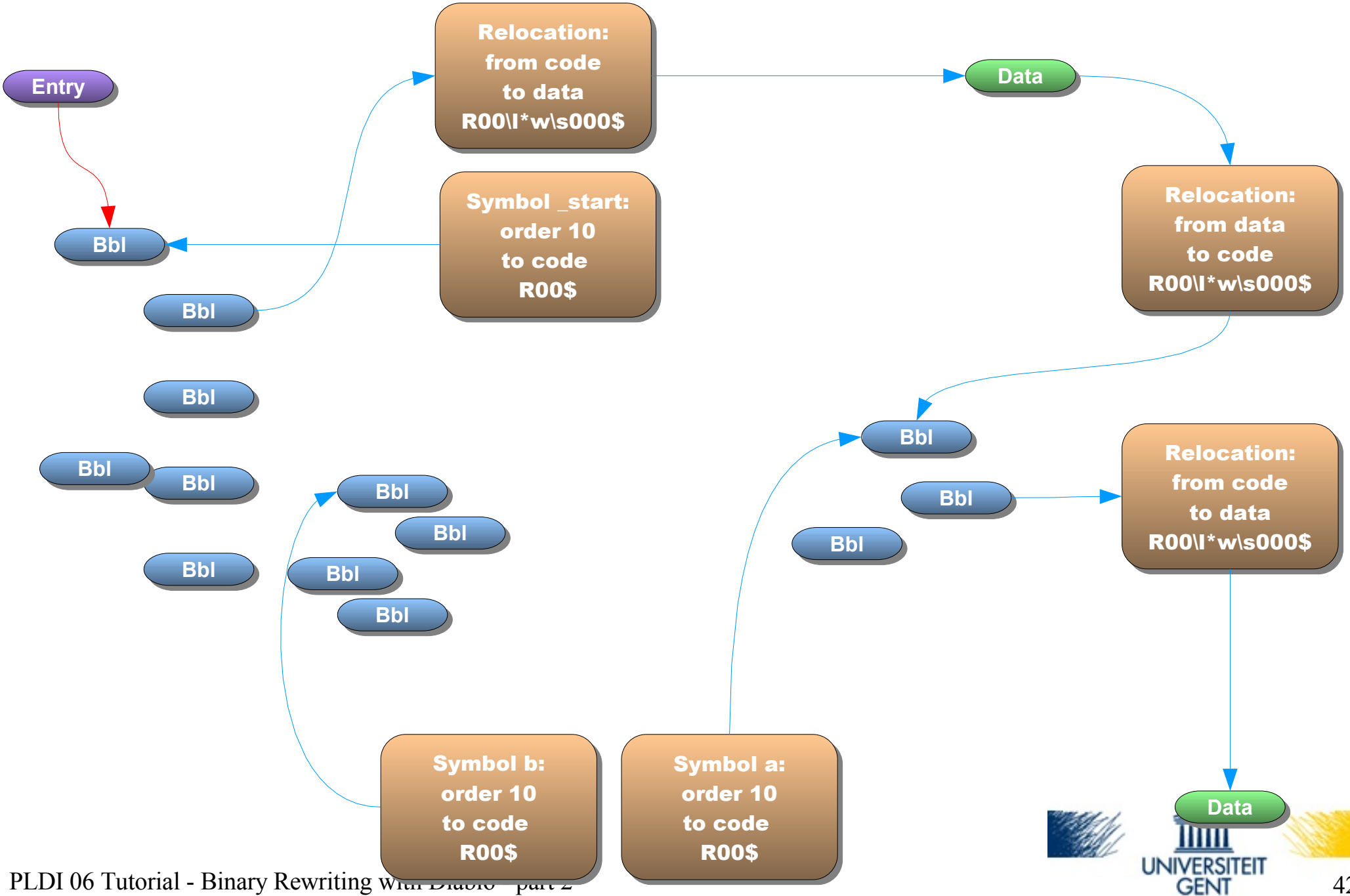
Detect basic blocks



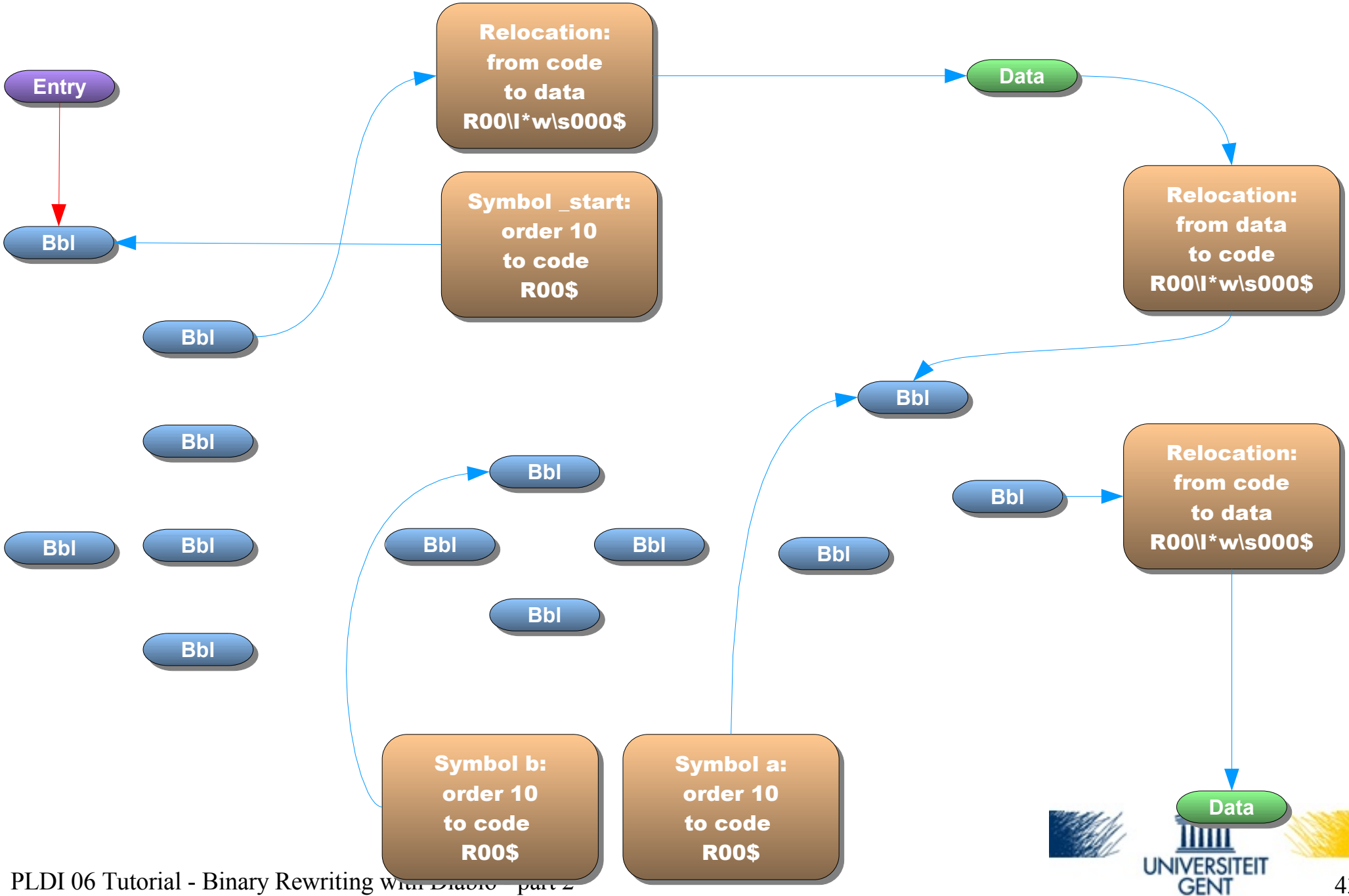
Detect basic blocks



Detect basic blocks

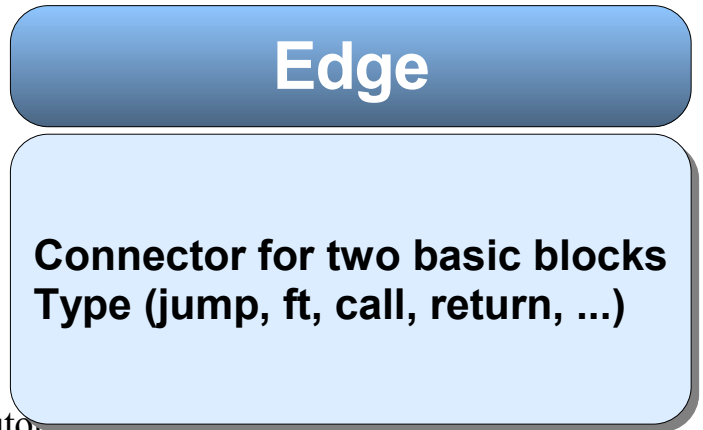


Detect basic blocks

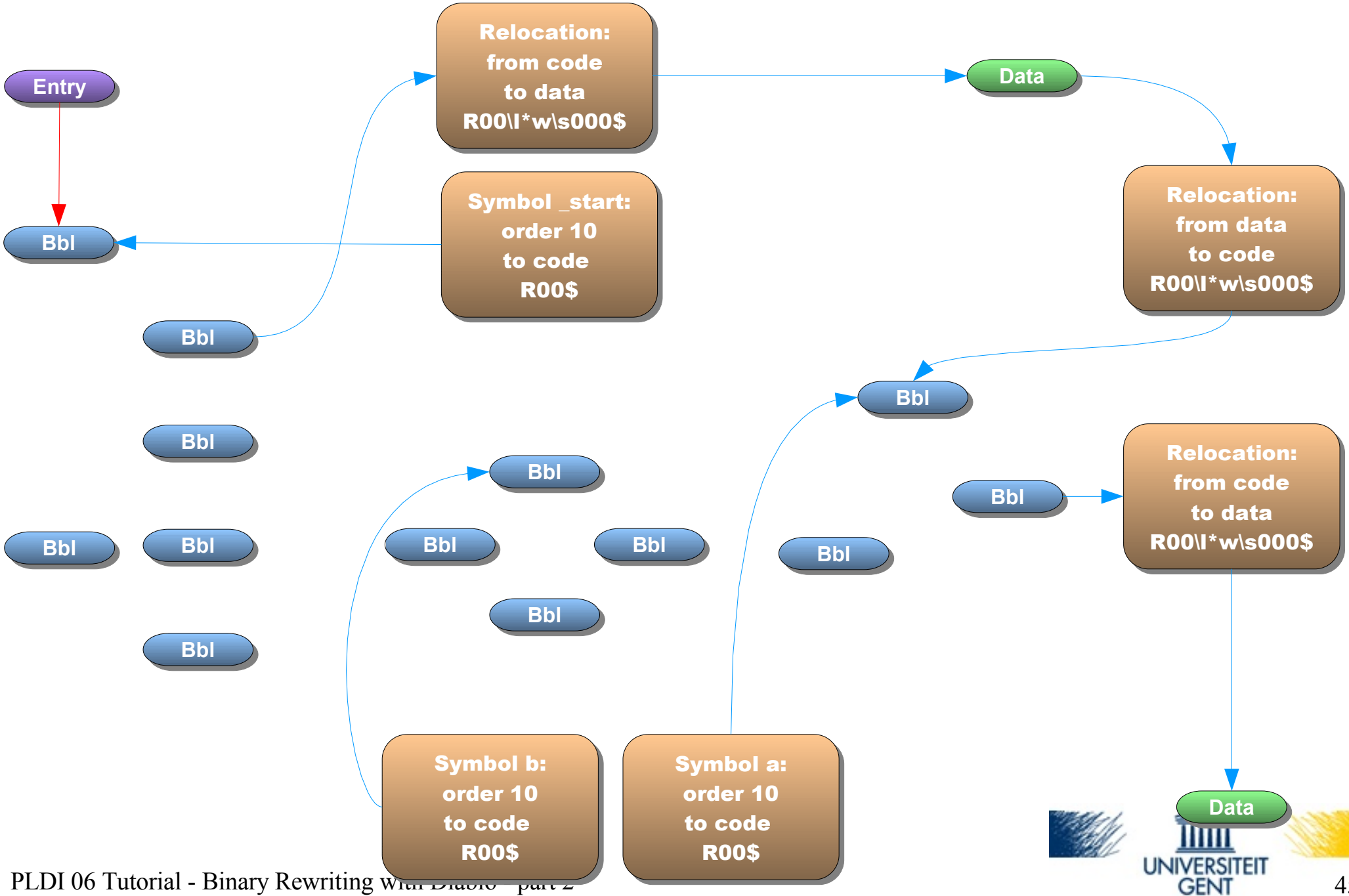


Add edges

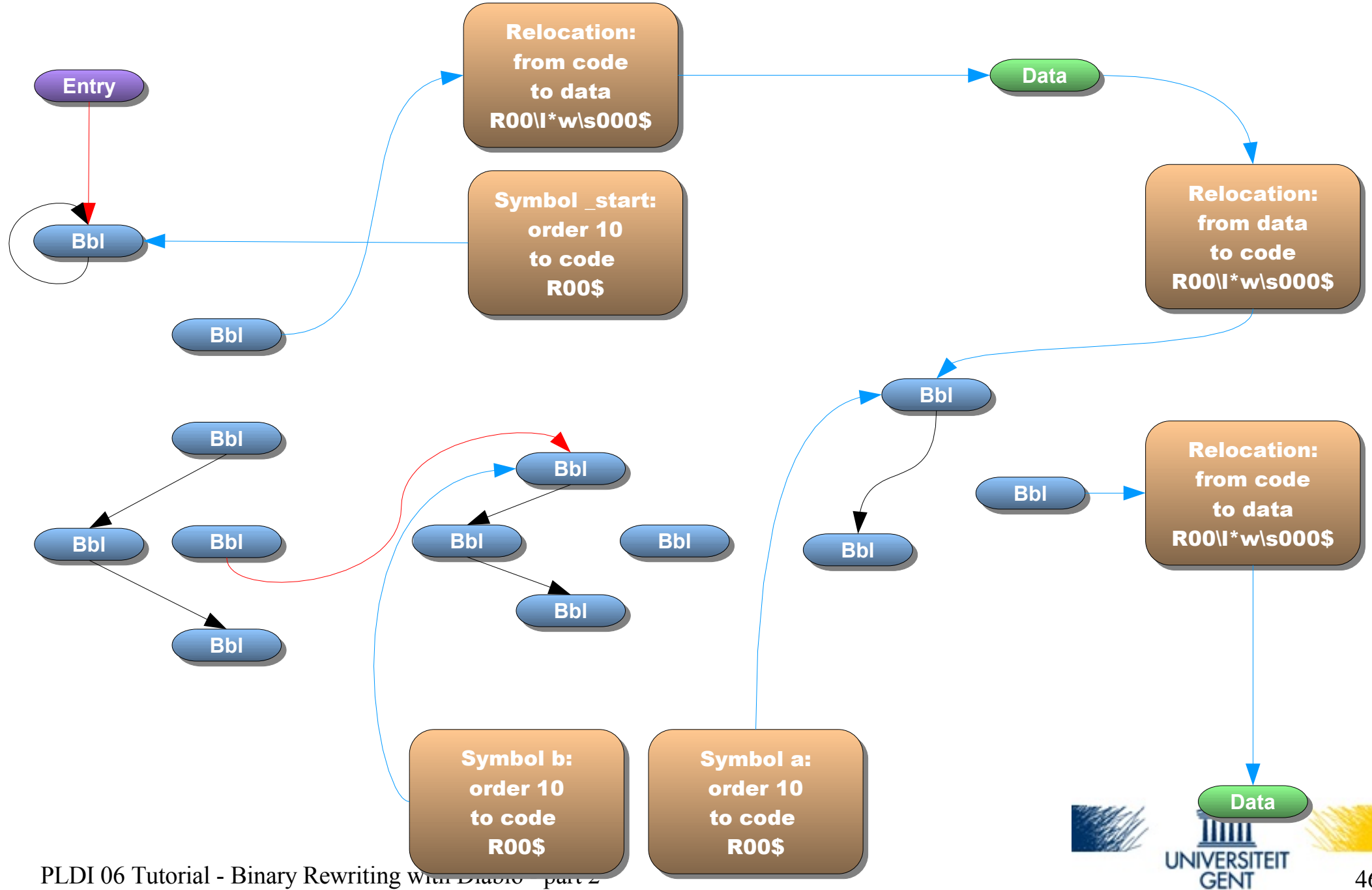
- Disassemble
- Split sections into basic blocks
  - Targets direct jumps + successors conditional jumps
  - To's of relocs
  - analyze switches (computed) to find switch targets
- Add direct control flow edges and switch edges



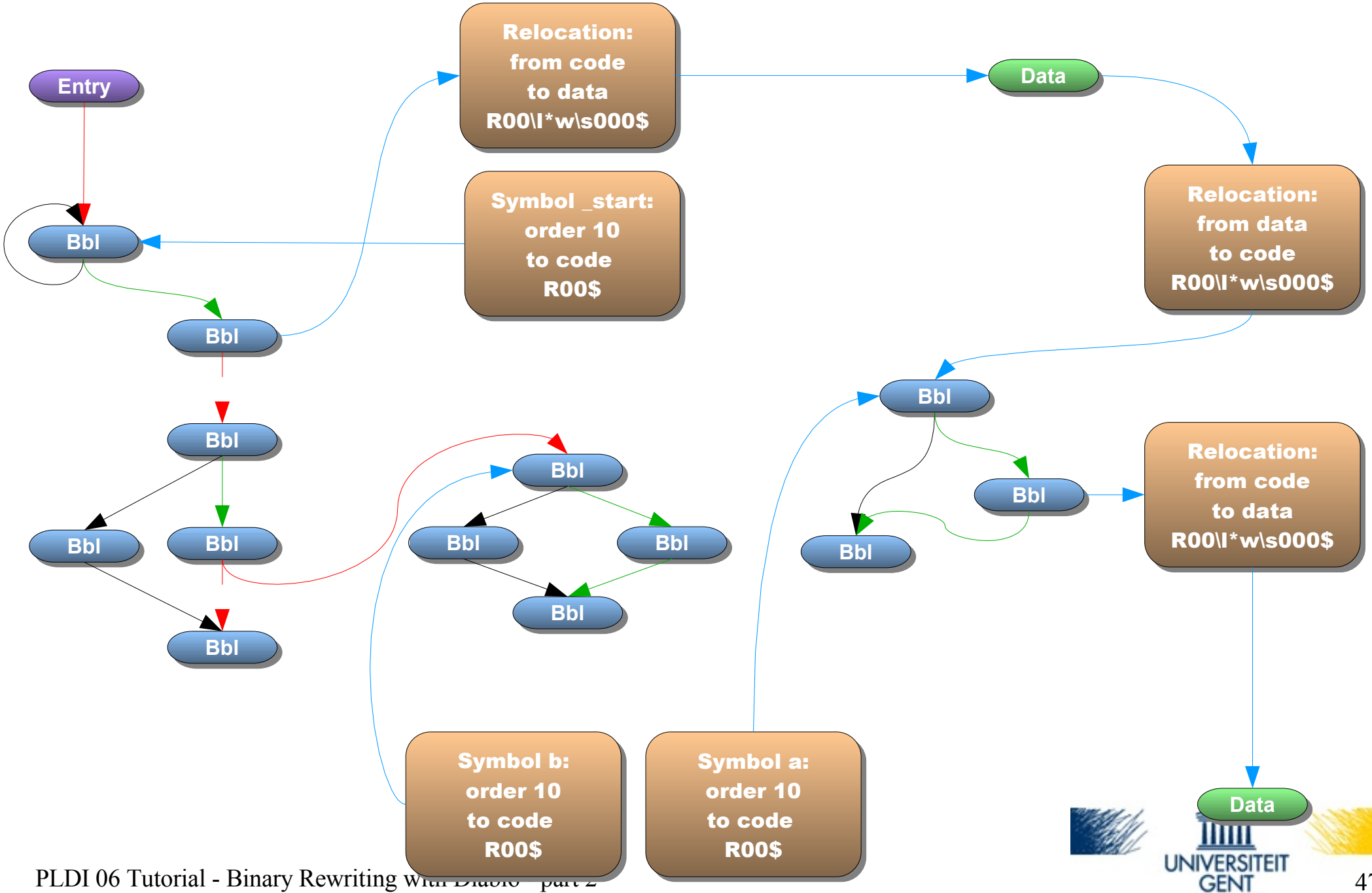
Add edges


Relocation:
from code
to data
R00\l*w\s000$
Data
Entry
Symbol _start:
order 10
to code
R00$
Relocation:
from data
to code
R00\l*w\s000$
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Relocation:
from code
to data
R00\l*w\s000$
Symbol b:
order 10
to code
R00$
Symbol a:
order 10
to code
R00$
Data

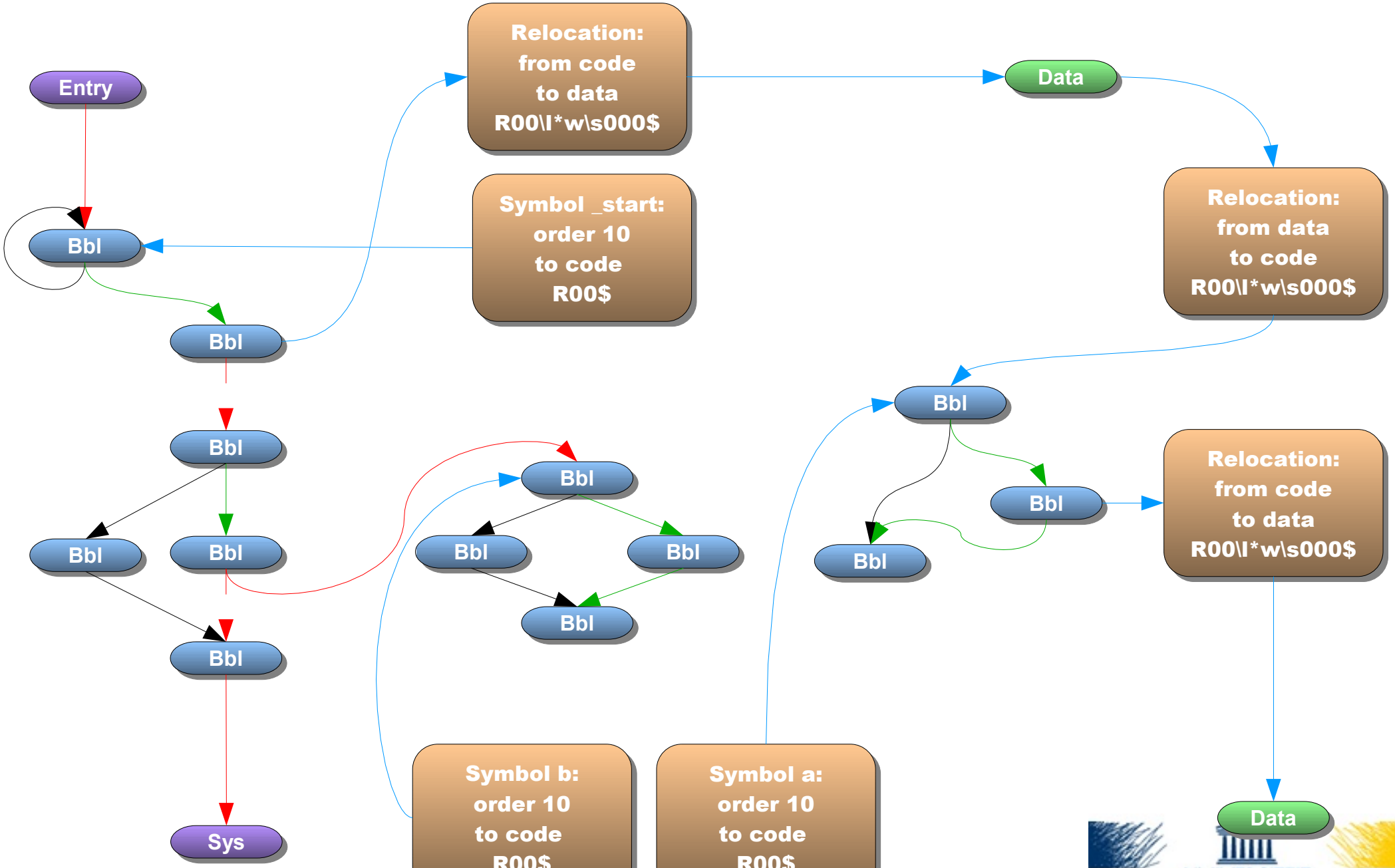
Add edges: direct jumps



Add edges: fall-through paths



Add edges: system calls

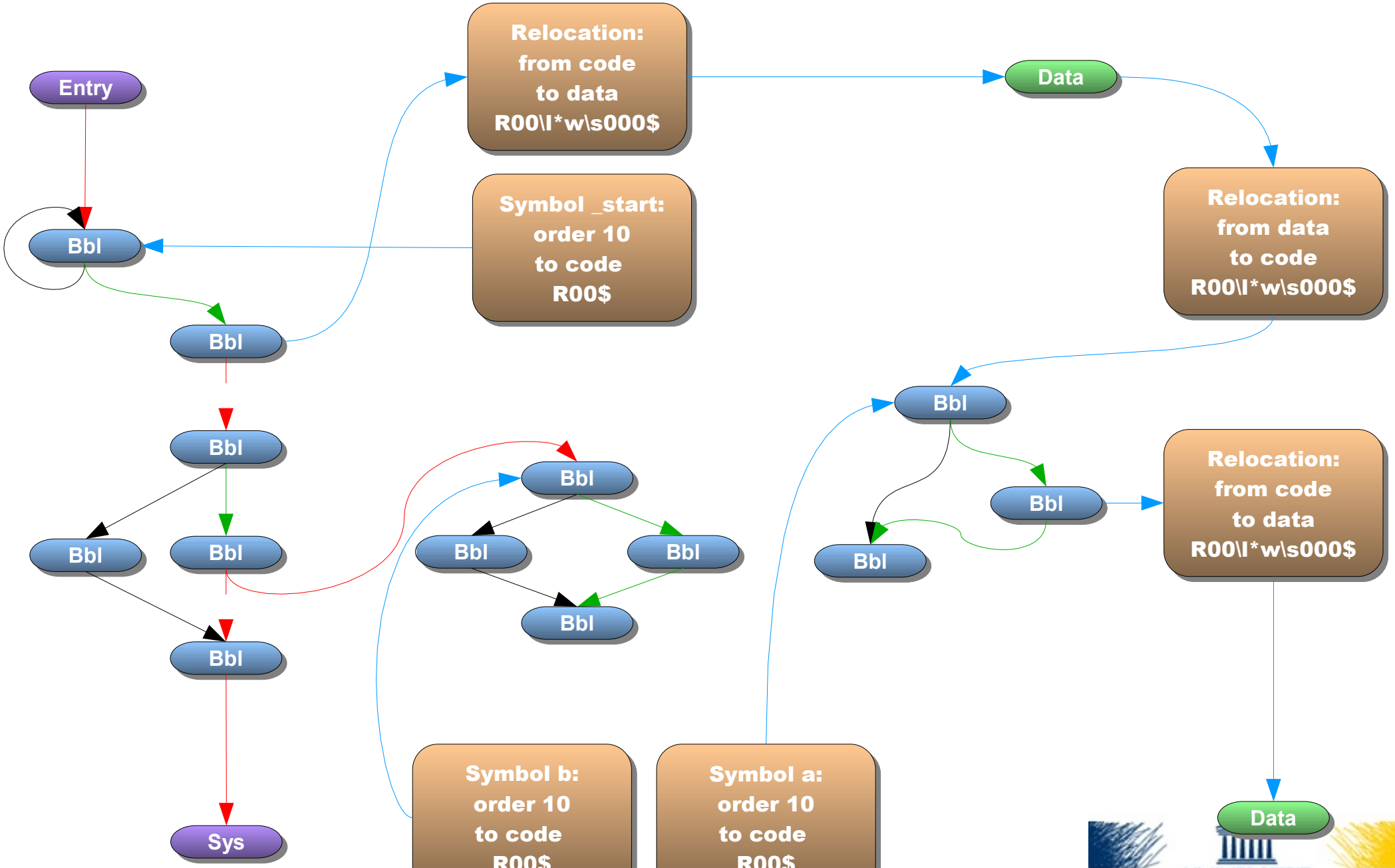


Partition the code into functions

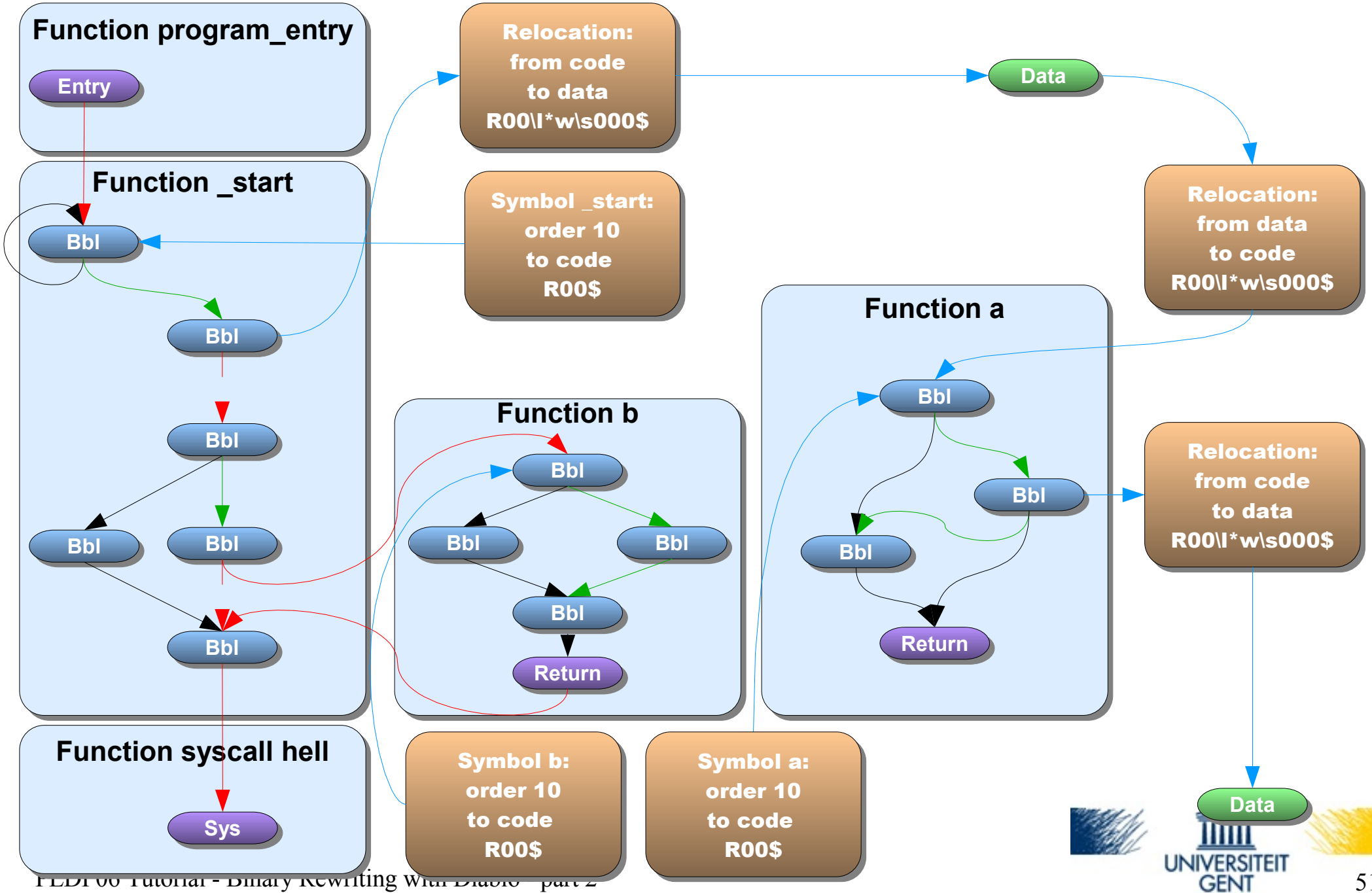
Function

- **Name**
- list of basic blocks
- register lists (used, defined, ...)

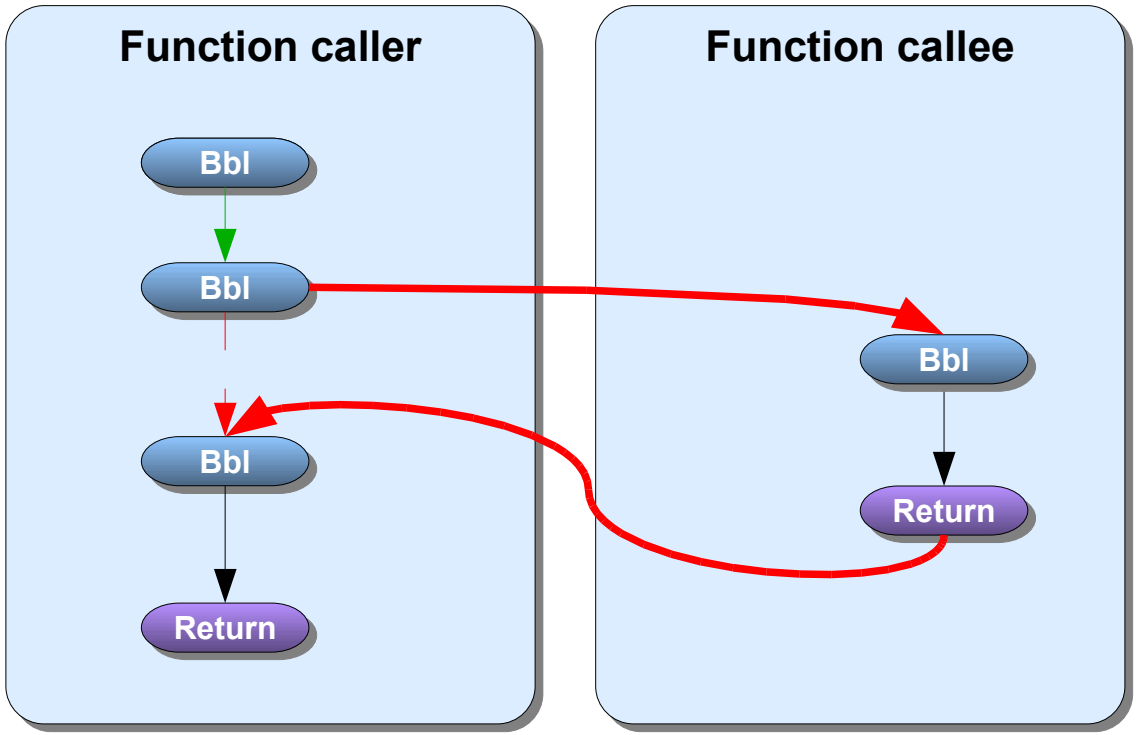
Partition the code into functions



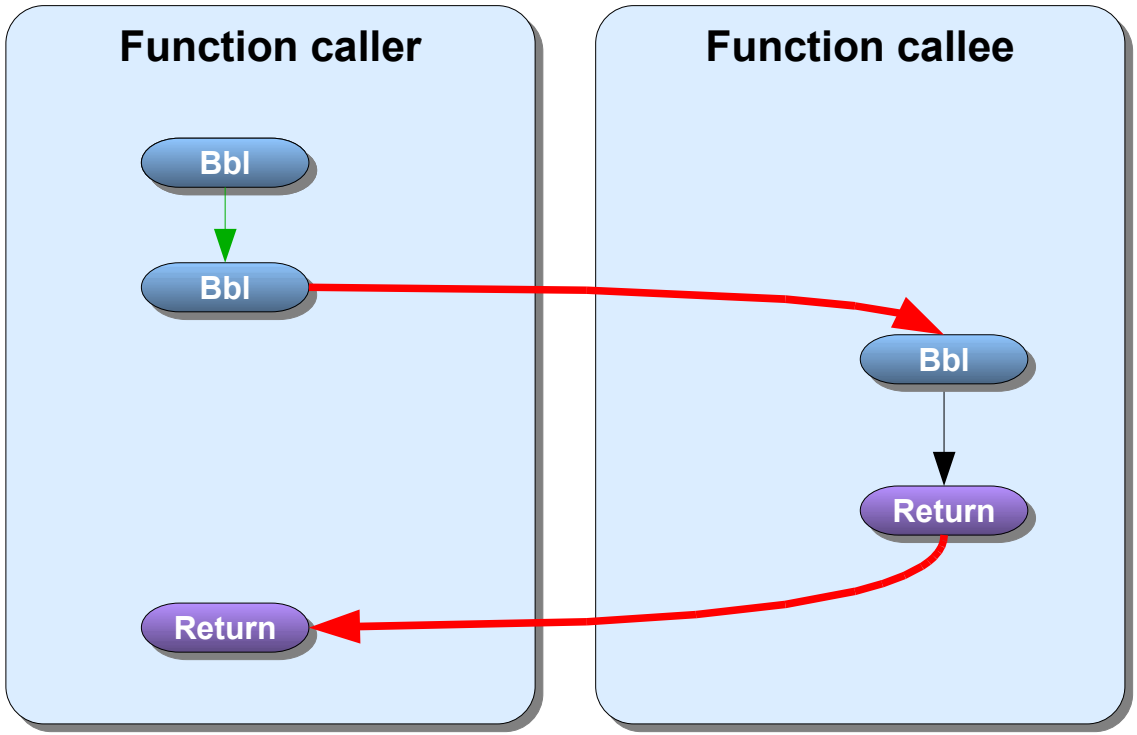
Partition the code into functions



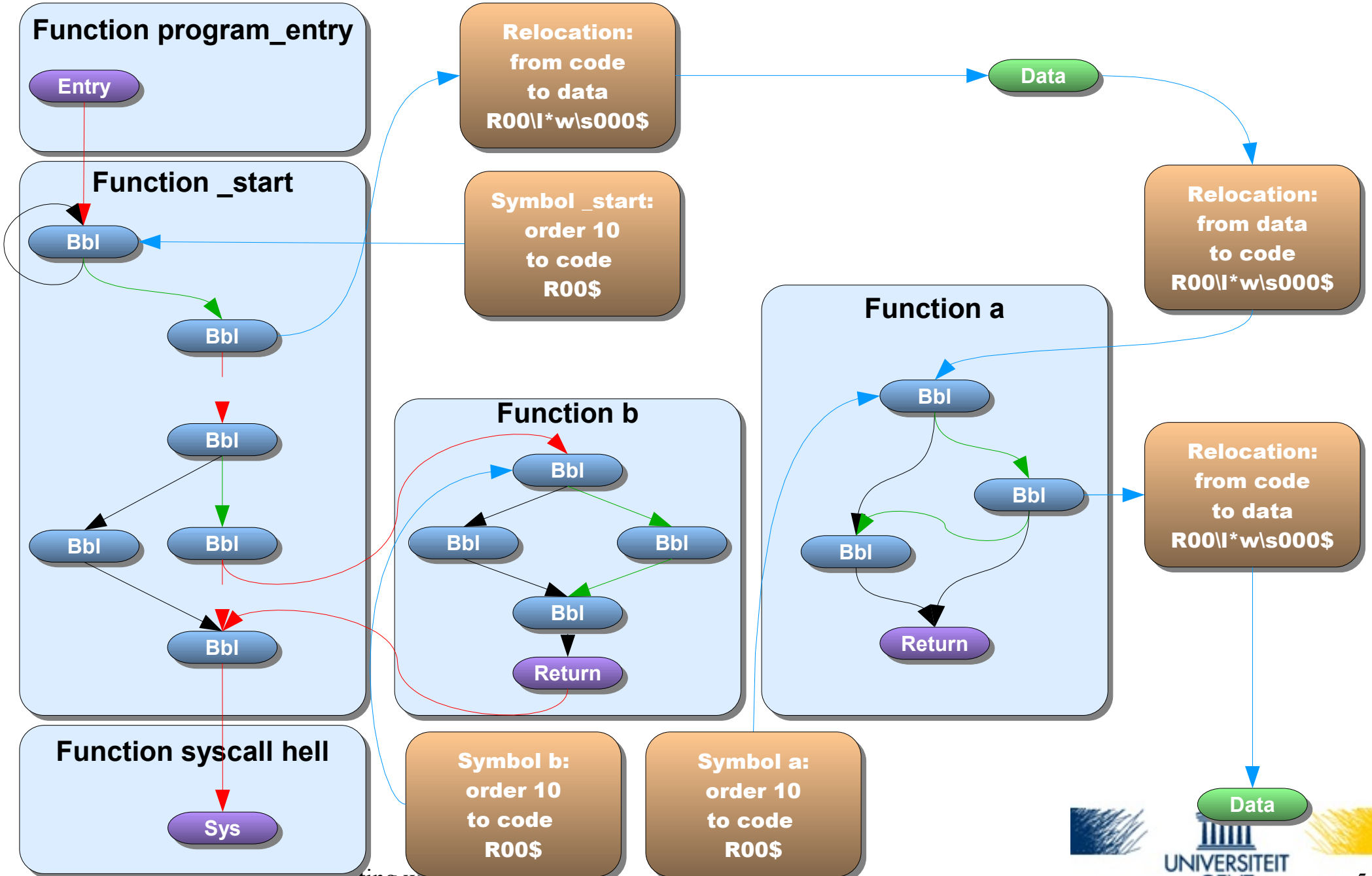
Function Calls



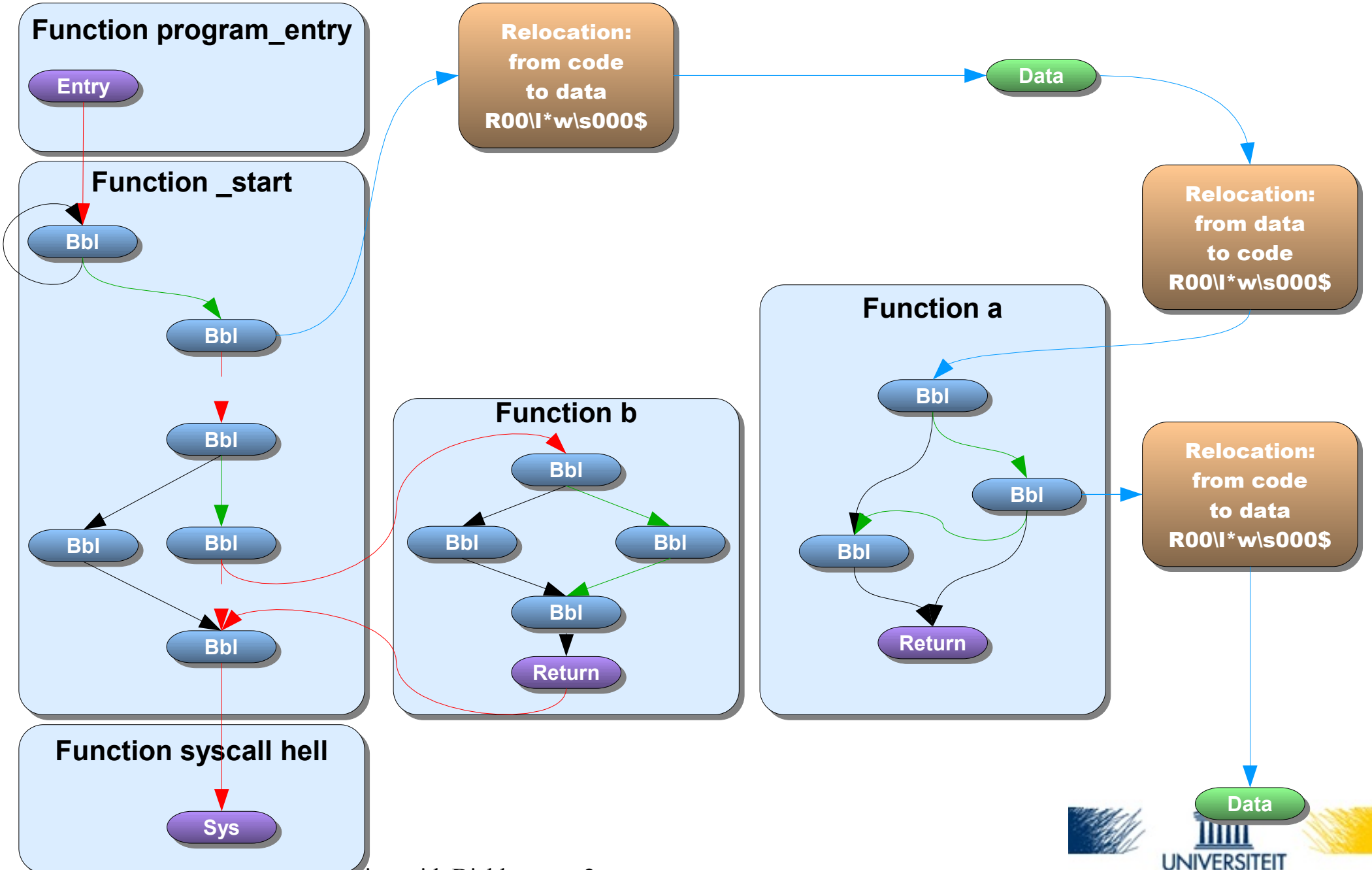
Interprocedural Goto's



DCFRAG


Function program_entry
Entry
Function _start
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Function syscall hell
Sys
Relocation:
from code
to data
R00\l*w\s000$
Symbol _start:
order 10
to code
R00$
Function b
Bbl
Bbl
Bbl
Bbl
Return
Symbol b:
order 10
to code
R00$
Data
Relocation:
from data
to code
R00\l*w\s000$
Function a
Bbl
Bbl
Bbl
Return
Relocation:
from code
to data
R00\l*w\s000$
Data
Symbol a:
order 10
to code
R00$

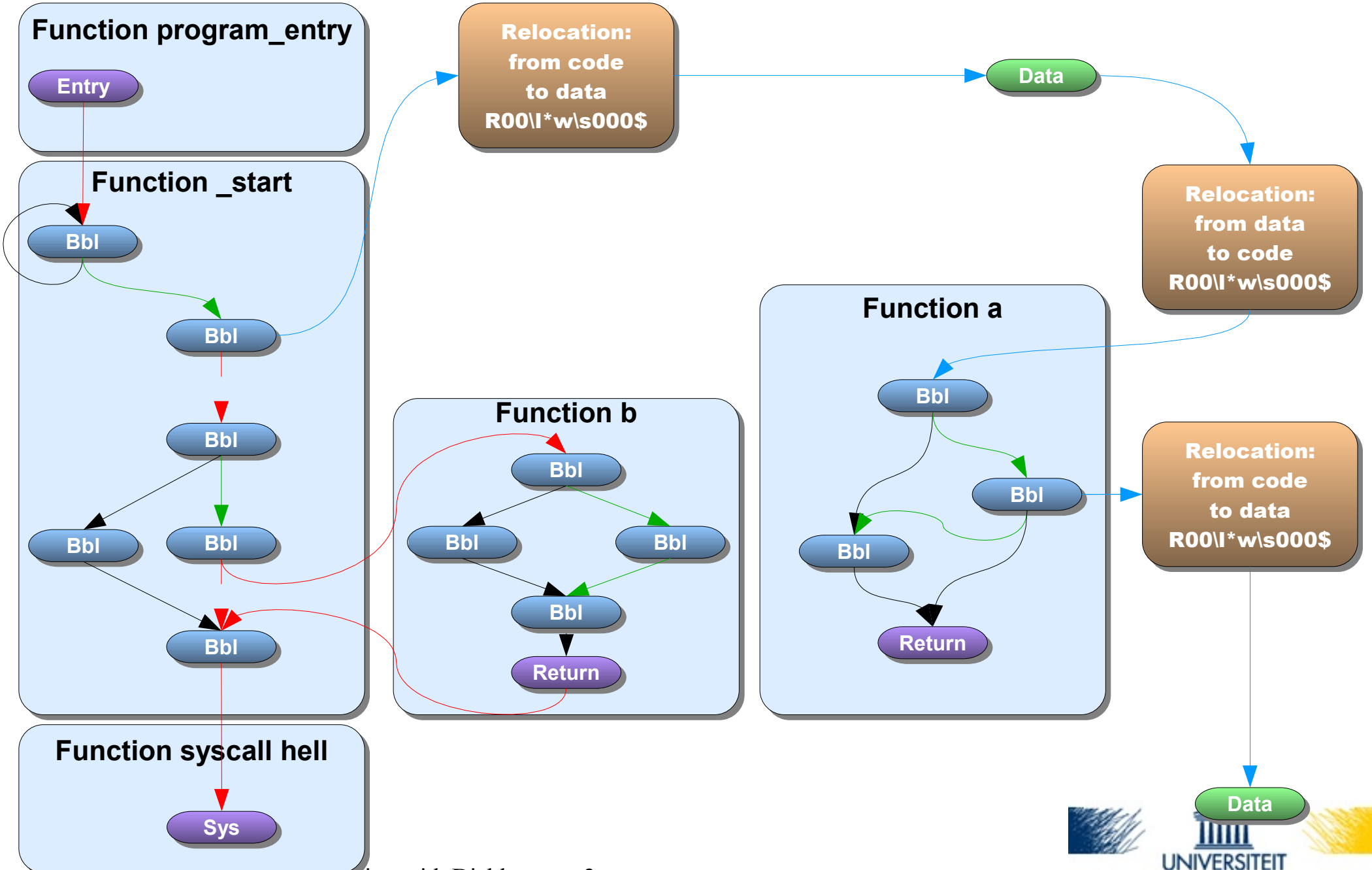
DCFRAG


Function program_entry
Entry
Function _start
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Function syscall hell
Sys
Relocation:
from code
to data
R00\l*w\s000$
Data
Relocation:
from data
to code
R00\l*w\s000$
Function b
Bbl
Bbl
Bbl
Bbl
Return
Function a
Bbl
Bbl
Bbl
Return
Relocation:
from code
to data
R00\l*w\s000$
Data

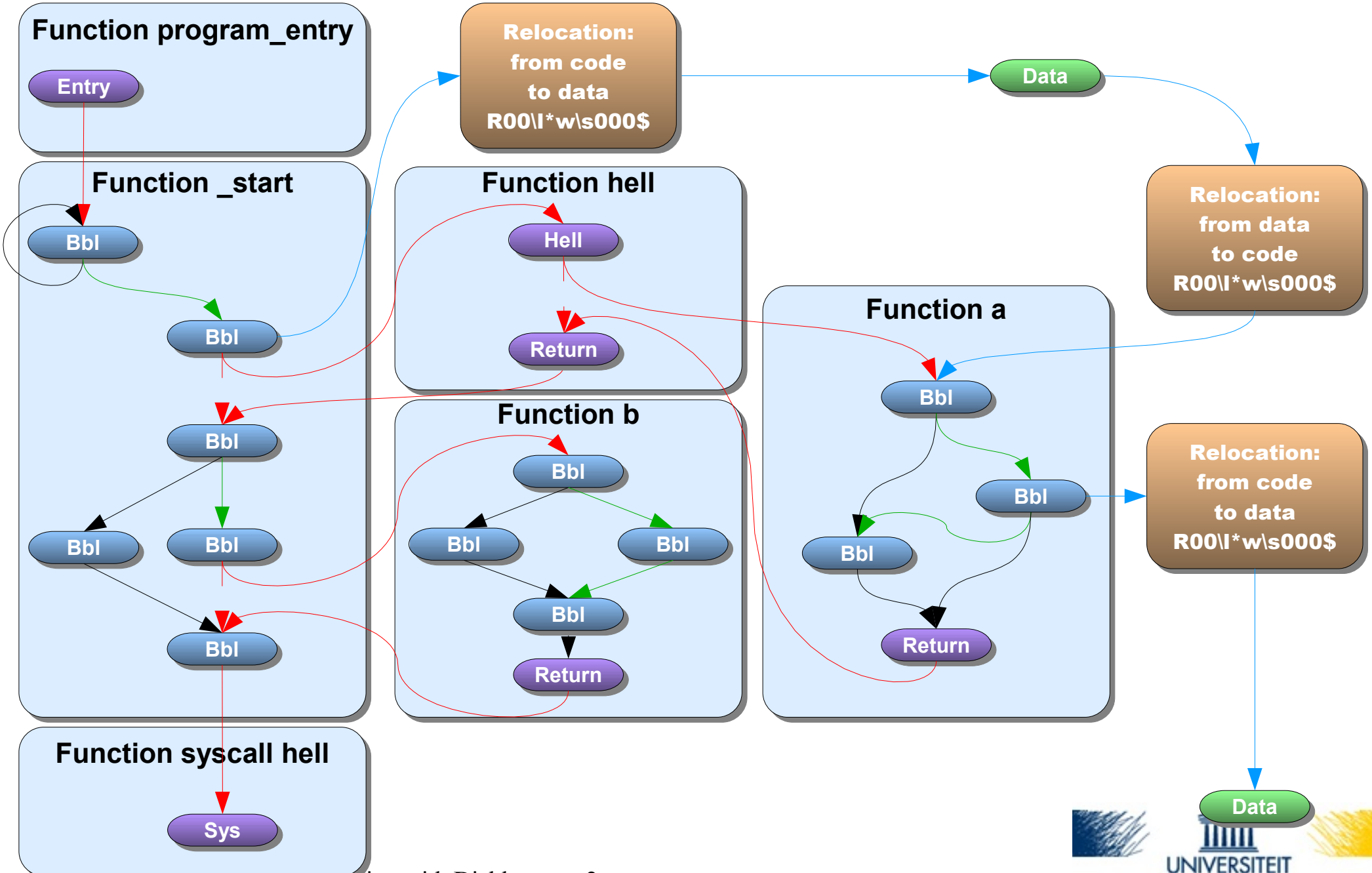
Uses of the DCFRAG

- ✔ Fine grained removal of unused code and data
- ✗ Dataflow
- ➔ We need a graph for data flow analyses (ICFG)

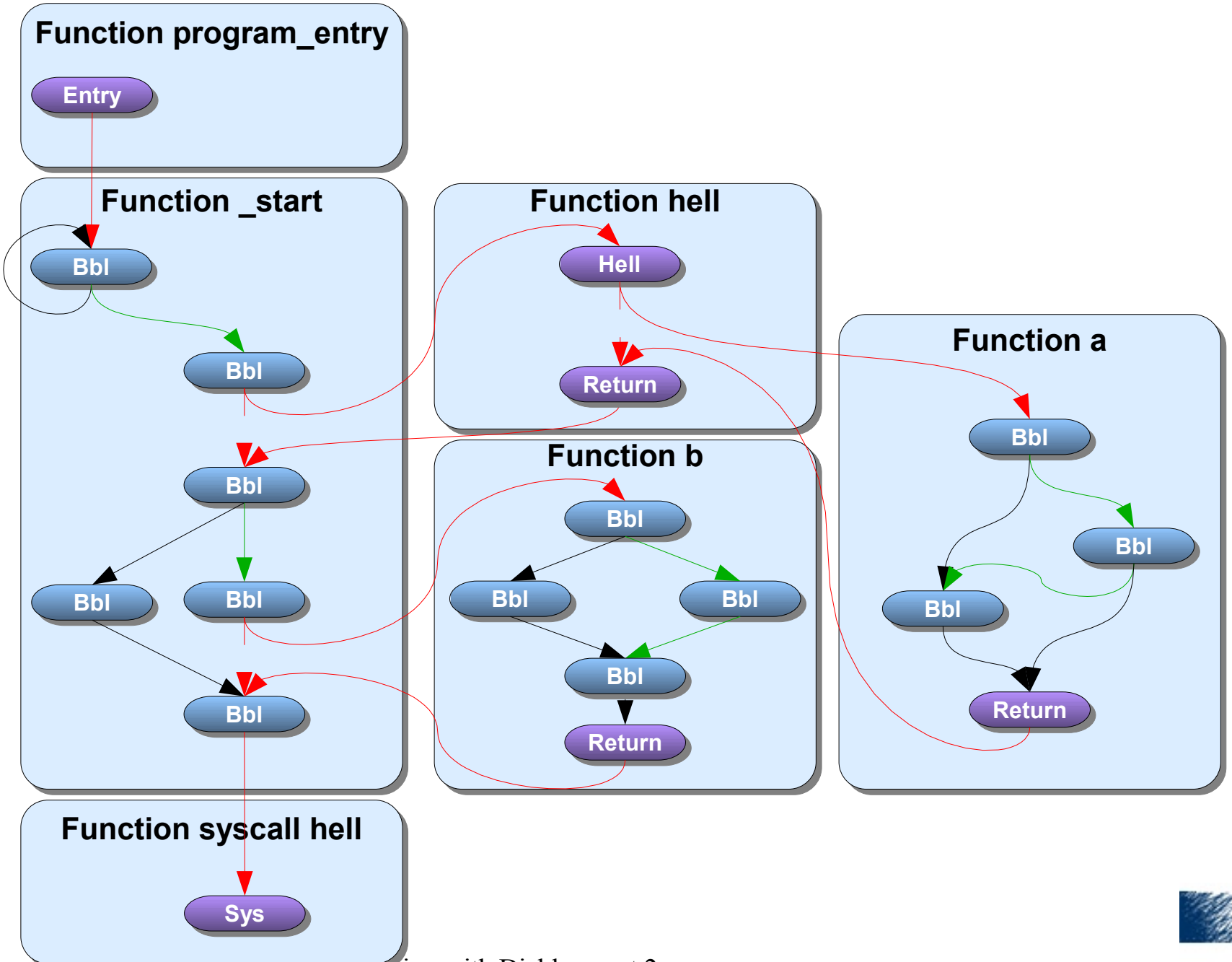
DCFRAG


Function program_entry
Entry
Function _start
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Function syscall hell
Sys
Relocation:
from code
to data
R00\l*w\s000$
Data
Relocation:
from data
to code
R00\l*w\s000$
Function b
Bbl
Bbl
Bbl
Bbl
Return
Function a
Bbl
Bbl
Bbl
Return
Relocation:
from code
to data
R00\l*w\s000$
Data

Augmented Whole-Program CFG



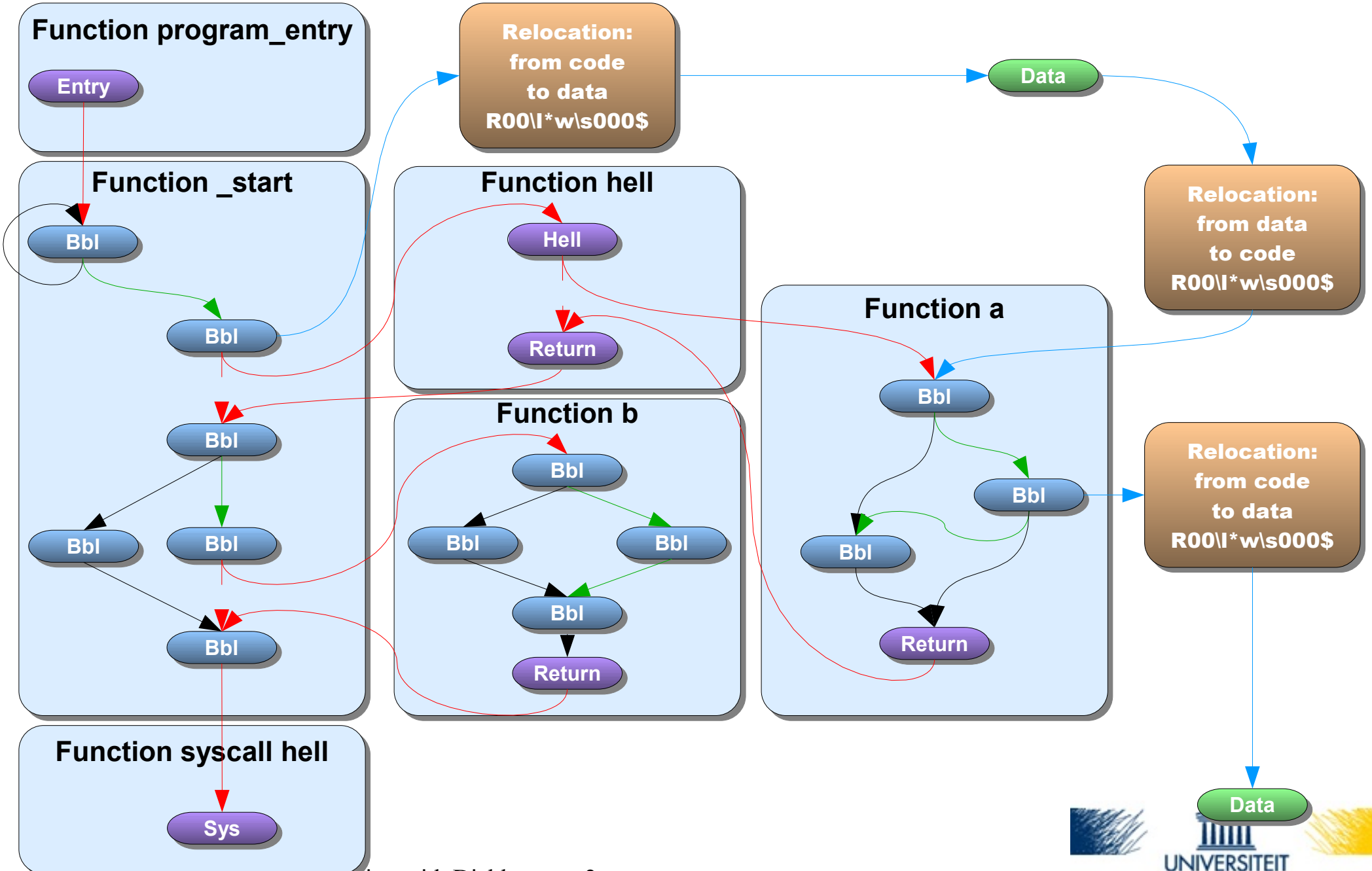
ICFG


Function program_entry
Entry
Function _start
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Function hell
Hell
Return
Function b
Bbl
Bbl
Bbl
Bbl
Return
Function a
Bbl
Bbl
Bbl
Return
Function syscall hell
Sys

Uses of the ICFG

- ✔ Good for analysis and transformations
- ✗ Bad for writing out the program
- ➔ Use the combined ICFG + DCFRAG = AWPCFG

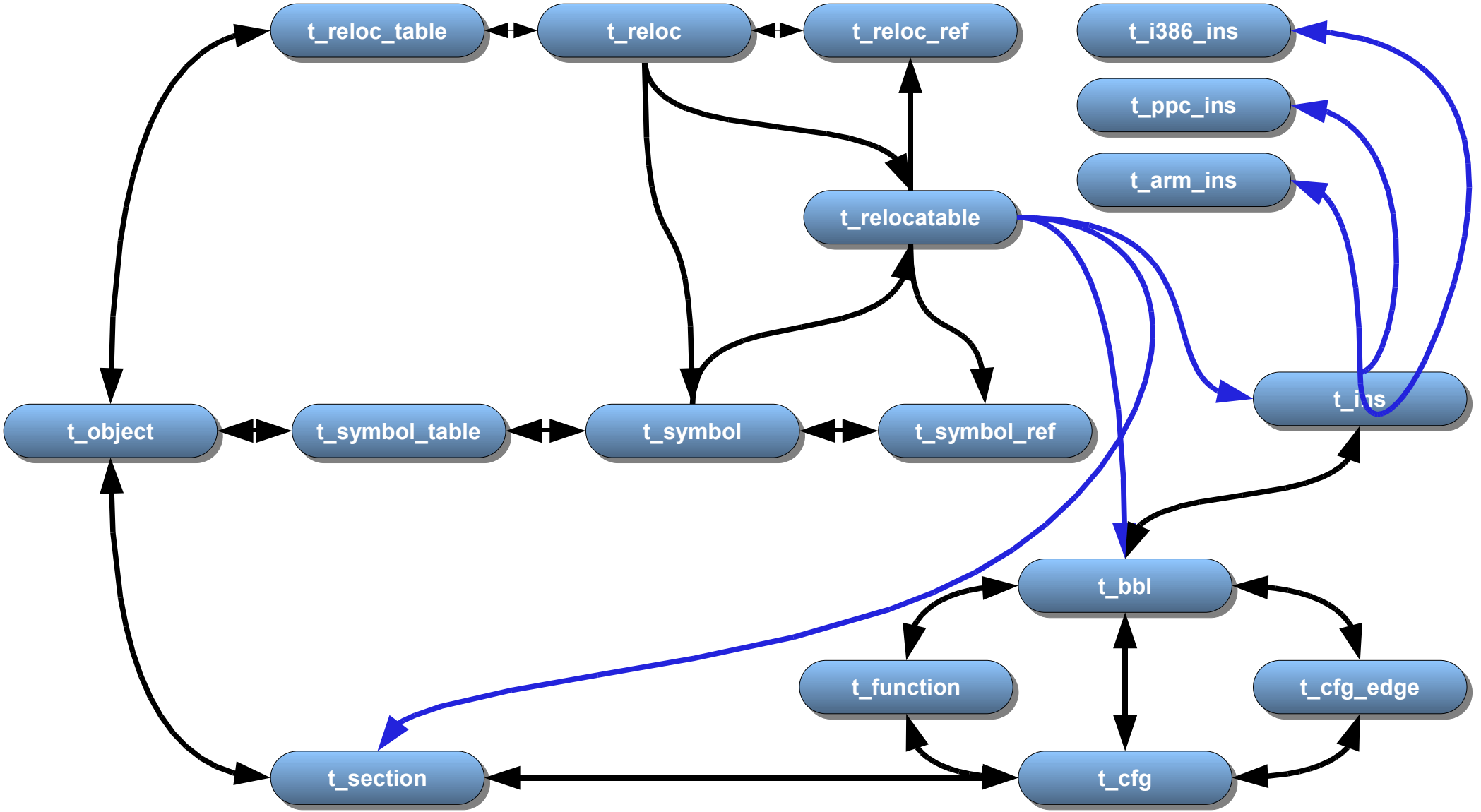
AWPCFG


Function program_entry
Entry
Relocation: from code to data R00\l*w\s000$
Data
Relocation: from data to code R00\l*w\s000$
Function _start
Bbl
Bbl
Bbl
Bbl
Bbl
Bbl
Function hell
Hell
Return
Function b
Bbl
Bbl
Bbl
Bbl
Return
Function a
Bbl
Bbl
Bbl
Return
Relocation: from code to data R00\l*w\s000$
Data
Function syscall hell
Sys

Part 2:
Diablo Data Structures

- Goals
- Linker data structures
- Internal representation
- Construction of graphs
- Concrete Data Structures
- Manipulation
- Dynamic Members

Concrete Data Structures



Accessing fields

- With getters and setters

- `t_bbl * head = CFG_EDGE_HEAD(cfg_edge)`
- `ARM_INS_SET_REGA(arm_ins, reg)`

- Reason: Dynamic Members

Iterating the ICFG

- t_cfg * cfg; t_function * fun; t_bbl * bbl; t_ins * ins; t_cfg_edge * edge;
- CFG_FOREACH_FUNCTION(cfg, fun)
  - FUNCTION_FOREACH_BBL(fun, bbl)
- CFG_FOREACH_BBL(cfg, bbl)
  - BBL_FOREACH_INS(bbl, ins)
  - BBL_FOREACH_SUCC_EDGE(bbl, edge)
  - BBL_FOREACH_PRED_EDGE(bbl, edge)

Part 2:
Diablo Data Structures

- Goals
- Linker data structures
- Internal representation
- Construction of graphs
- Concrete Data Structures
- Manipulation
- Dynamic Members

Primitive Transformations

- Relocations
  - Add: `RelocTableAddRelocToRelocatable`
  - Remove: `RelocTableRemoveReloc`
  - Modify: `RelocSetFrom`, `RelocSetToRelocatable`
- Sections
  - Create: `SectionCreateForObject`
- Functions
  - Create: `FunctionMake`
  - Remove: `FunctionKill`

Primitive Transformations

- Basic blocks
  - Create: `BblNew`
  - Remove: `BblKill`
  - Duplicate: `BblDup`
  - Split: `BblSplitBlock`
- ICFG edges
  - Create: `CfgEdgeCreate`
  - Remove: `CfgEdgeKill`
- Instructions
  - Create: `InsNewForBbl`
  - Remove: `InsKill`

On the DCFRAG

- SECTION_REFED_BY(sec)
- SECTION_REFERS_TO(sec)

- BBL_REFED_BY(bbl)
- BBL_REFED_BY_SYM(bbl)

- INS_REFERS_TO(ins)

Consistency of the AWPCFG

- Manipulate one view, what happens on other?
- Diablo tries to keep things consistent
  - Kill reloc, and its ICFG-edge is also killed
  - Kill ins, and to-relocs are also killed
  - Remove a section, and all to-relocs are also killed
- Makes sure that you do the proper thing
  - Try to kill an object with refers_to relocs, and it fatals

Part 2:
Diablo Data Structures

- Goals
- Linker data structures
- Internal representation
- Construction of graphs
- Concrete Data Structures
- Manipulation
- Dynamic Members

Dynamic Members

- Diablo needs to be extensible
- Many analyses compute different kinds of information on very large data sets
- We cannot include all of them in the main libraries, or even store all information together
- Dynamic members augment basic data structures with members that can be allocated on the fly

Dynamic Members

- Example: member for reachability
  - t_bool BBL_REACHABLE(t_bbl)
  - BBL_SET_REACHABLE(t_bbl, t_bool)
  - BblInitReachable(t_cfg *)
    - Allocates space for this field and calls init callback for each bbl
  - BblFiniReachable(t_cfg *)
    - Calls fini callback and deallocates space

Dynamic Members

To instantiate the member:

```
DYNAMIC_MEMBER(
   bbl,                    /* data structure to extend */
   t_cfg *,                /* manager type */
   bbl_reachable_array,    /* array to hold members */
   t_bool,                 /* the type of the member */
   reachable,              /* lowercase name */
   REACHABLE,              /* UPPERCASE name */
   Reachable,              /* CamelCase name*/
   CFG_FOREACH_BBL,        /* iterator */
   BblReachableInitCb,     /* init callback*/
   BblReachableFiniCb,     /* fini callback */
   BblReachableDupCb,      /* dup callback */
);
```