Tutorial on Essential Abstractions in GCC

Module Binding Mechanisms and Control Flow in GCC

Uday Khedker

(www.cse.iitb.ac.in/grc)

GCC Resource Center,
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

April 2011

Outline

- Motivation
- Plugins in GCC
- GCC Control Flow
- Conclusions

Part 1

Motivation

Module Binding Mechanisms

- The need for adding, removing, and maintaining modules relatively independently
- The mechanism for supporting this is called by many names:
  - Plugin, hook, callback, ...
  - Sometimes it remains unnamed (eg. compilers in gcc driver)
- It may involve
  - Minor changes in the main source
    Requires static linking
  - No changes in the main source
    Requires dynamic linking

Module Binding Mechanisms

- The need for adding, removing, and maintaining modules relatively independently
- The mechanism for supporting this is called by many names:
  - Plugin, hook, callback, ...
  - Sometimes it remains unnamed (eg. compilers in gcc driver)
- It may involve
  - Minor changes in the main source
    Requires static linking
    We call this a static plugin
  - No changes in the main source
    Requires dynamic linking
    We call this a dynamic plugin

Plugin as a Module Binding Mechanisms

- We view plugin at a more general level than the conventional view Adjectives "static" and "dynamic" create a good contrast
- Most often a plugin in a C based software is a data structure containing function pointers and other related information

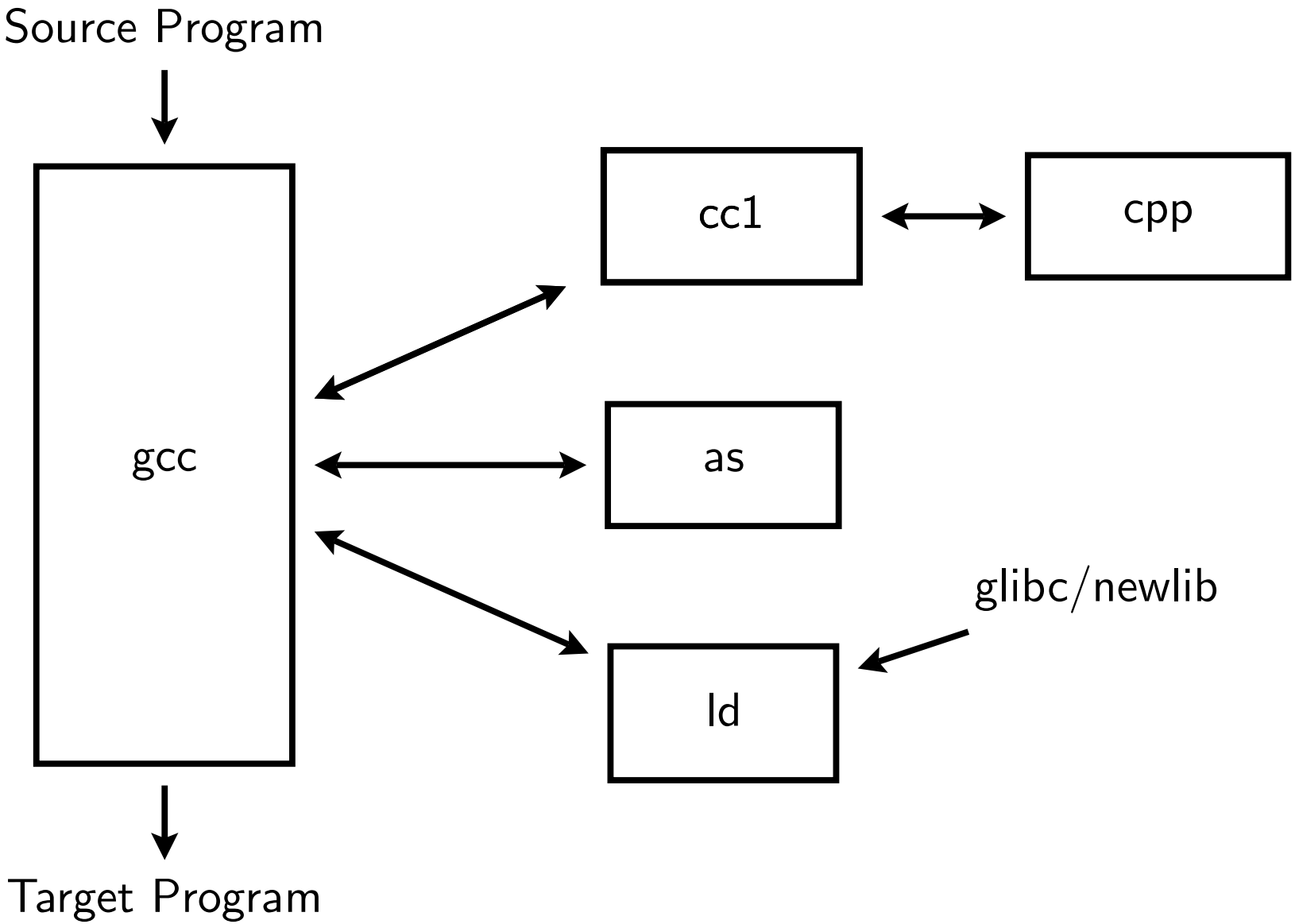
Static Vs. Dynamic Plugins

- Static plugin requires static linking
  - Changes required in `gcc/Makefile.in`, some header and source files
  - At least `cc1` may have to be rebuild
    All files that include the changed headers will have to be recompiled
- Dynamic plugin uses dynamic linking
  - Supported on platforms that support `-ldl -rdynamic`
  - Loaded using `dlopen` and invoked at pre-determined locations in the compilation process
  - Command line option

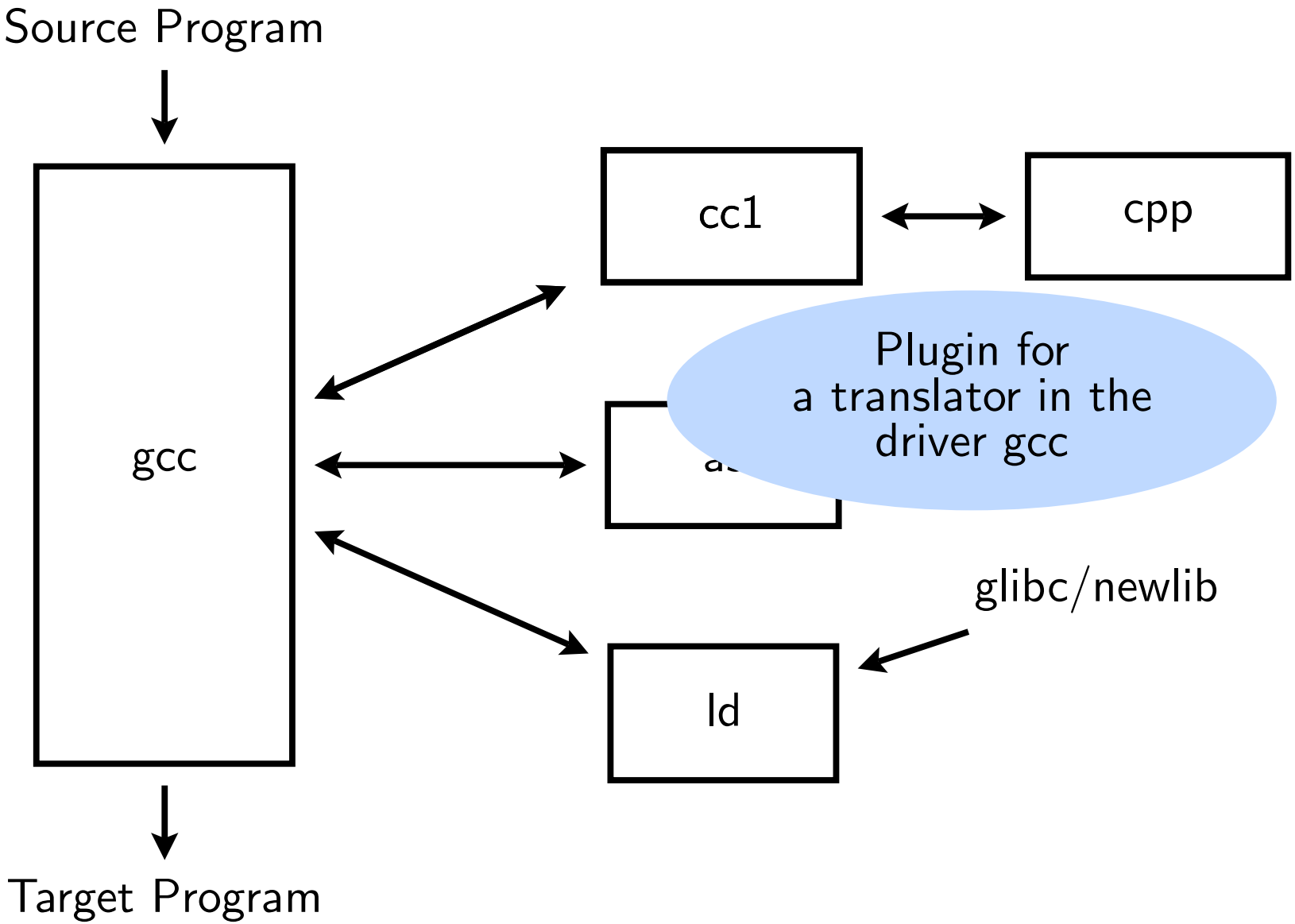
    `-fplugin=/path/to/name.so`

    Arguments required can be supplied as name-value pairs

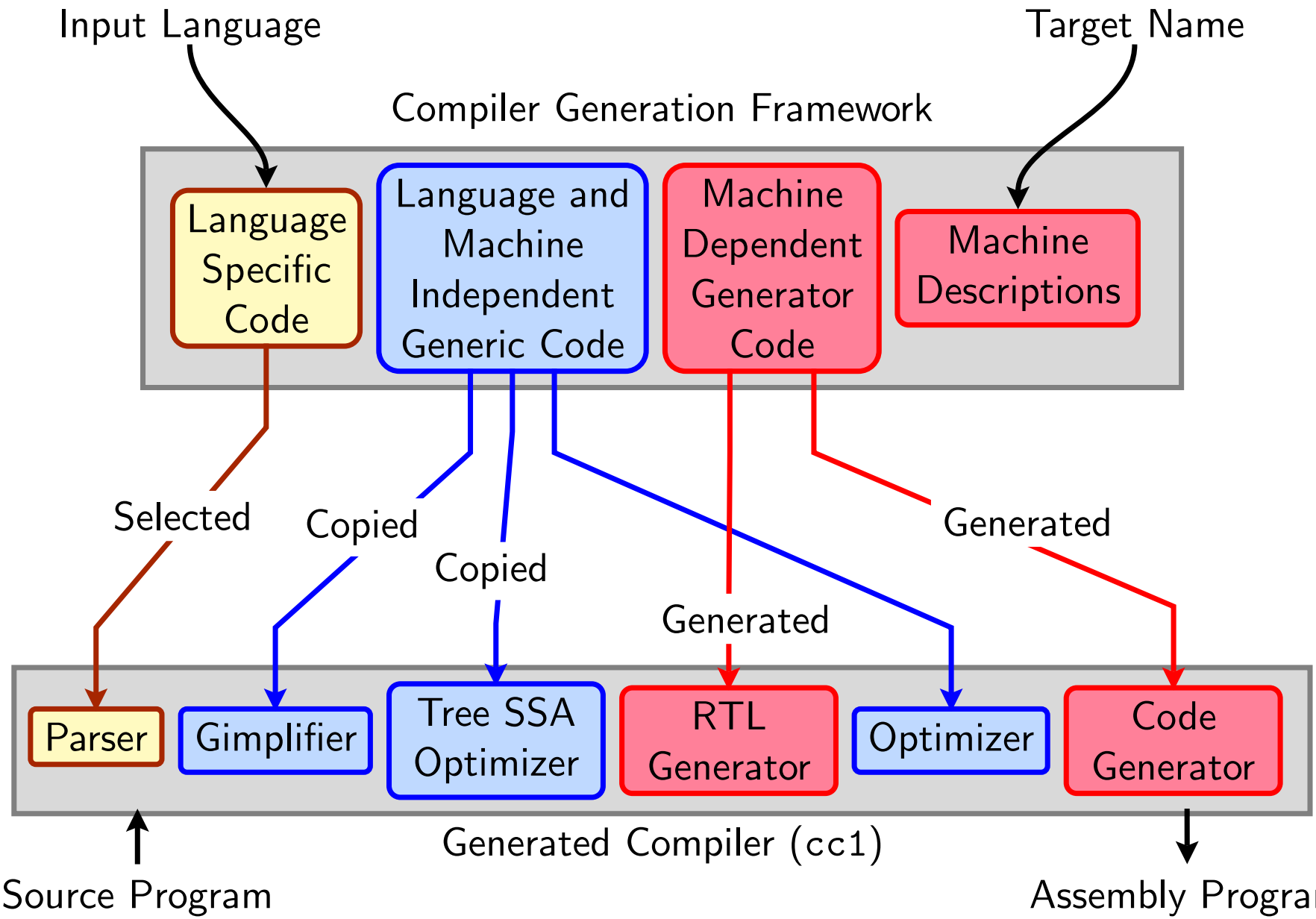
Static Plugins in the GCC Driver



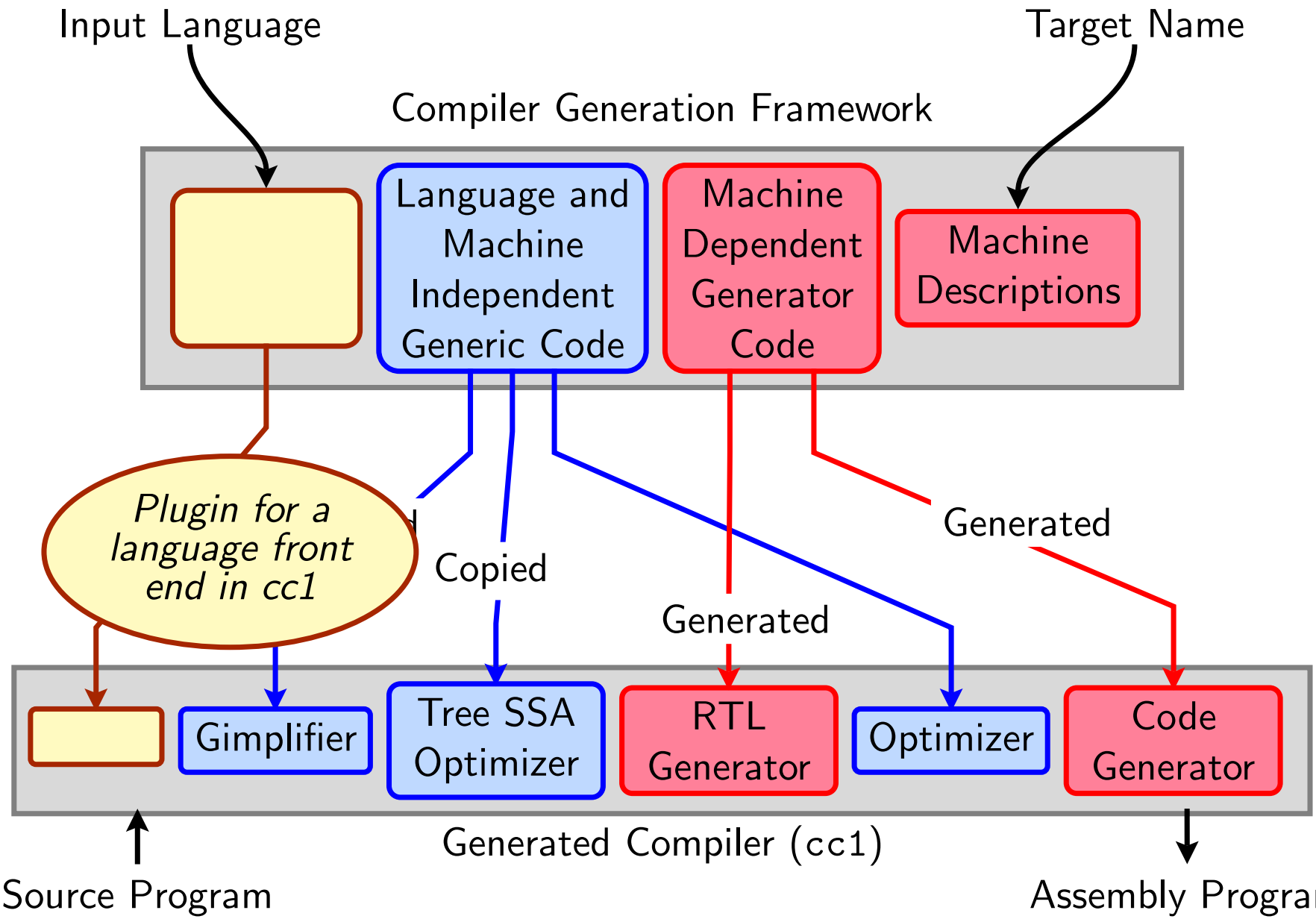
Static Plugins in the GCC Driver



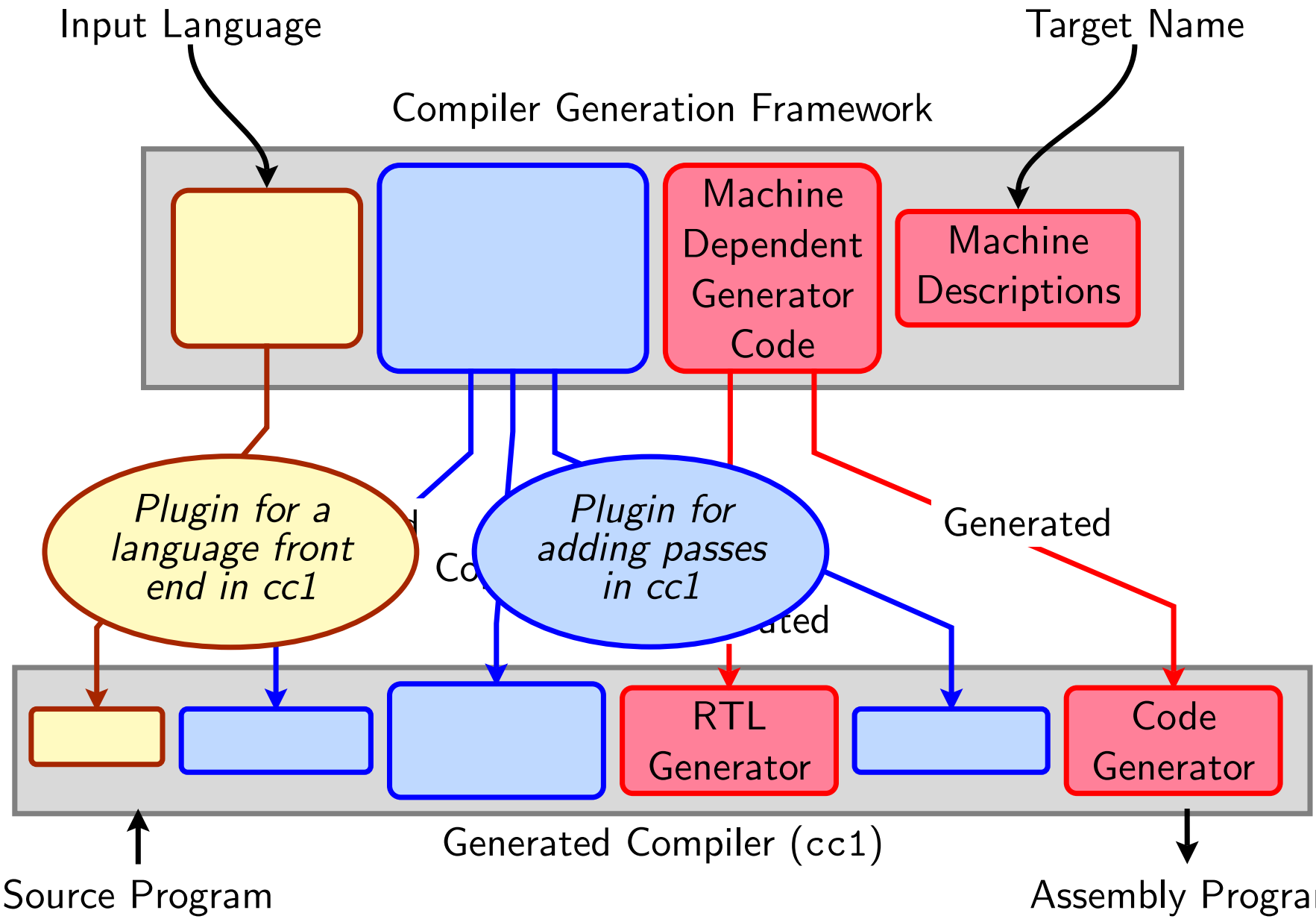
Static Plugins in the Generated Compiler



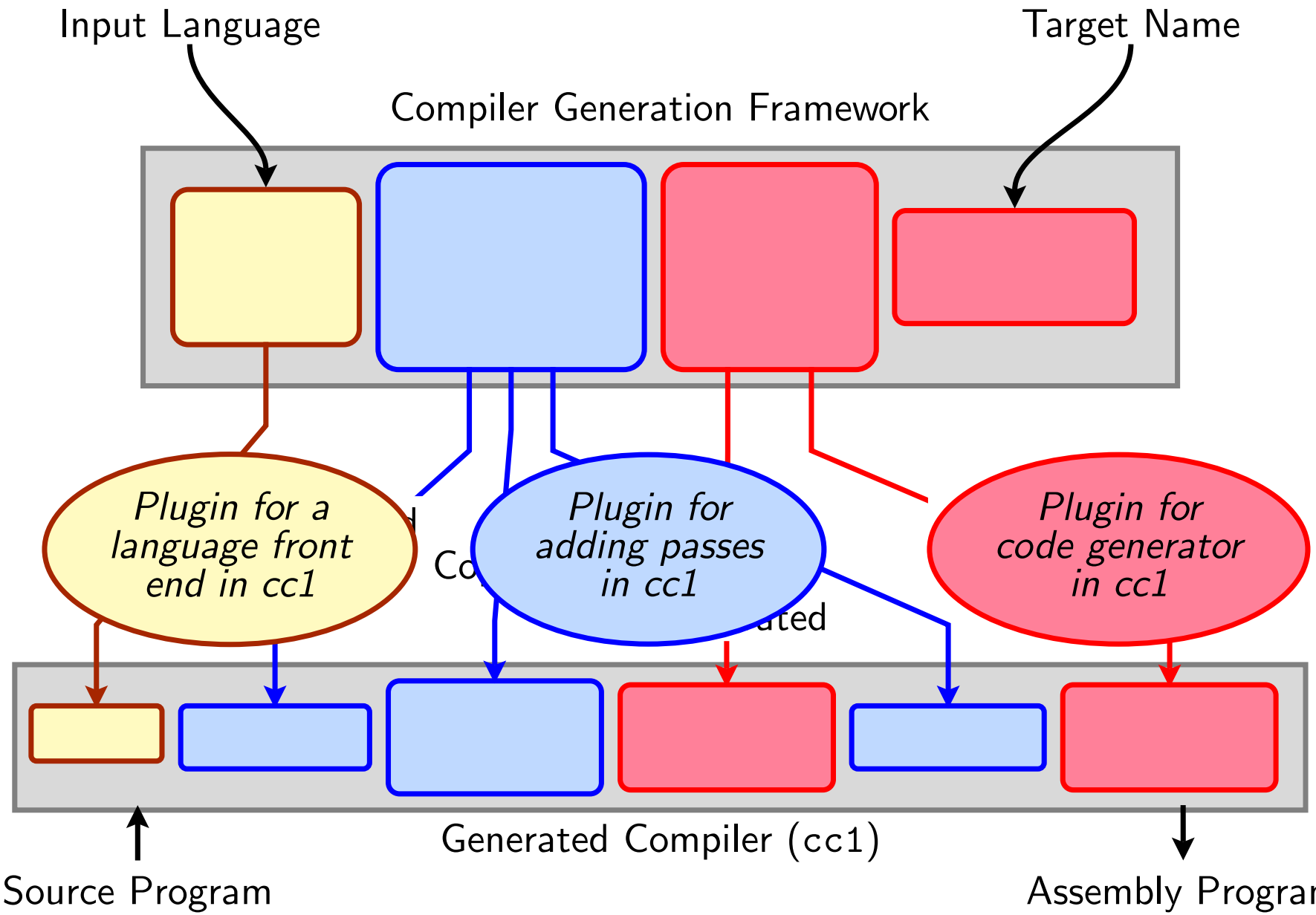
Static Plugins in the Generated Compiler



Static Plugins in the Generated Compiler



Static Plugins in the Generated Compiler



Part 2

Static Plugins in GCC

GCC's Solution

Plugin	Implementation	
	Data Structure	Initialization
Translator in gcc	Array of C structures	Development time
Front end in cc1	C structure	Build time
Passes in cc1	Linked list of C structures	Development time
Back end in cc1	Arrays of structures	Build time

Plugin Data Structure in the GCC Driver

```
struct compiler
{
  const char *suffix;      /* Use this compiler for input files
                              whose names end in this suffix.  */

  const char *spec;        /* To use this compiler, run this spec. *

  const char *cpp_spec;    /* If non-NULL, substitute this spec
                              for '%C', rather than the usual
                              cpp_spec.  */
  const int combinable;    /* If nonzero, compiler can deal with
                              multiple source files at once (IMA).
  const int needs_preprocessing;
                           /* If nonzero, source files need to
                              be run through a preprocessor.  */
};
```

Default Specs in the Plugin Data Structure in gcc.c

All entries of Objective C/C++ and some entries of Fortran removed.

```
static const struct compiler default_compilers[] =
{
  {".cc", "#C++", 0, 0, 0},          {".cxx", "#C++", 0, 0, 0},
  {".cpp", "#C++", 0, 0, 0},         {".cp", "#C++", 0, 0, 0},
  {".c++", "#C++", 0, 0, 0},         {".C", "#C++", 0, 0, 0},
  {".CPP", "#C++", 0, 0, 0},         {".ii", "#C++", 0, 0, 0},
  {".ads", "#Ada", 0, 0, 0},         {".adb", "#Ada", 0, 0, 0},
  {".f", "#Fortran", 0, 0, 0},       {".F", "#Fortran", 0, 0, 0},
  {".for", "#Fortran", 0, 0, 0},     {".FOR", "#Fortran", 0, 0, 0},
  {".f90", "#Fortran", 0, 0, 0},     {".F90", "#Fortran", 0, 0, 0},
  {".p", "#Pascal", 0, 0, 0},        {".pas", "#Pascal", 0, 0, 0},
  {".java", "#Java", 0, 0, 0},       {".class", "#Java", 0, 0, 0},
  {".c", "@c", 0, 1, 1},
  {".h", "@c-header", 0, 0, 0},
  {".i", "@cpp-output", 0, 1, 0},
  {".s", "@assembler", 0, 1, 0}
}
```

Default Specs in the Plugin Data Structure in gcc.c

All entries of Objective C/C++ and some entries of Fortran removed.

```
static const struct compiler default_compilers[] =
{
  {".cc", "#C++", 0, 0, 0},          {".cxx", "#C++", 0, 0, 0},
  {".cpp", "#C++", 0, 0, 0},         {".cp", "#C++", 0, 0, 0},
  {".c++", "#C++", 0, 0, 0},         {".C", "#C++", 0, 0, 0},
  {".CPP", "#C++", 0, 0, 0},         {".ii", "#C++", 0, 0, 0},
  {".ads", "#Ada", 0, 0, 0},         {".adb", "#Ada", 0, 0, 0},
  {".f", "#Fortran", 0, 0, 0},       {".F", "#Fortran", 0, 0, 0},
  {".for", "#Fortran", 0, 0, 0},     {".FOR", "#Fortran", 0, 0, 0},
  {".f90", "#Fortran", 0, 0, 0},     {".F90", "#Fortran", 0, 0, 0},
  {".p", "#Pascal", 0, 0, 0},        {".pas", "#Pascal", 0, 0, 0},
  {".java", "#Java", 0, 0, 0},       {".class", "#Java", 0, 0, 0},
  {".c", "@c", 0, 1, 1},
  {".h", "@c-header", 0, 0, 0},
  {".i", "@cpp-output", 0, 1, 0},
  {".s", "@assembler", 0, 1, 0}
}
```

- @: Aliased entry

Default Specs in the Plugin Data Structure in gcc.c

All entries of Objective C/C++ and some entries of Fortran removed.

```
static const struct compiler default_compilers[] =
{
  {".cc", "#C++", 0, 0, 0},          {".cxx", "#C++", 0, 0, 0},
  {".cpp", "#C++", 0, 0, 0},         {".cp", "#C++", 0, 0, 0},
  {".c++", "#C++", 0, 0, 0},         {".C", "#C++", 0, 0, 0},
  {".CPP", "#C++", 0, 0, 0},         {".ii", "#C++", 0, 0, 0},
  {".ads", "#Ada", 0, 0, 0},         {".adb", "#Ada", 0, 0, 0},
  {".f", "#Fortran", 0, 0, 0},       {".F", "#Fortran", 0, 0, 0},
  {".for", "#Fortran", 0, 0, 0},     {".FOR", "#Fortran", 0, 0, 0},
  {".f90", "#Fortran", 0, 0, 0},     {".F90", "#Fortran", 0, 0, 0},
  {".p", "#Pascal", 0, 0, 0},        {".pas", "#Pascal", 0, 0, 0},
  {".java", "#Java", 0, 0, 0},       {".class", "#Java", 0, 0, 0},
  {".c", "@c", 0, 1, 1},
  {".h", "@c-header", 0, 0, 0},
  {".i", "@cpp-output", 0, 1, 0},
  {".s", "@assembler", 0, 1, 0}
}
```

- @: Aliased entry
- #: Default specs not available

Complete Entry for C in gcc.c

```
  {"@c",
   /* cc1 has an integrated ISO C preprocessor.  We should invoke the
      external preprocessor if -save-temps is given.  */
     "%{E|M|MM:%(trad_capable_cpp) %(cpp_options) %(cpp_debug_options)}\
      %{!E:%{!M:%{!MM:\
          %{traditional|ftraditional:\
%eGNU C no longer supports -traditional without -E}\
       %{!combine:\
  %{save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \
%(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i} \n\
    cc1 -fpreprocessed %{save-temps:%b.i} %{!save-temps:%g.i} \
%(cc1_options)}\
  %{!save-temps:%{!traditional-cpp:%{!no-integrated-cpp:\
cc1 %(cpp_unique_options) %(cc1_options)}}}\
          %{!fsyntax-only:%(invoke_as)}} \
      %{combine:\
  %{save-temps|traditional-cpp|no-integrated-cpp:%(trad_capable_cpp) \
%(cpp_options) -o %{save-temps:%b.i} %{!save-temps:%g.i}}\
  %{!save-temps:%{!traditional-cpp:%{!no-integrated-cpp:\
cc1 %(cpp_unique_options) %(cc1_options)}}\
                %{!fsyntax-only:%(invoke_as)}}}}}}", 0, 1, 1},
```


Populated Plugin Data Structure for C++:

gcc/cp/lang-specs.h

```
{".cc",  "@c++", 0, 0, 0},
{".cp",  "@c++", 0, 0, 0},
{".cxx", "@c++", 0, 0, 0},
{".cpp", "@c++", 0, 0, 0},
{".c++", "@c++", 0, 0, 0},
{".C",   "@c++", 0, 0, 0},
{".CPP", "@c++", 0, 0, 0},
{".H",   "@c++-header", 0, 0, 0},
{".hpp", "@c++-header", 0, 0, 0},
{".hp",  "@c++-header", 0, 0, 0},
{".hxx", "@c++-header", 0, 0, 0},
{".h++", "@c++-header", 0, 0, 0},
{".HPP", "@c++-header", 0, 0, 0},
{".tcc", "@c++-header", 0, 0, 0},
{".hh",  "@c++-header", 0, 0, 0},
{"@c++-header",
  "%{E|M|MM:cc1plus -E %(cpp_options) %2 %(cpp_debug_options)}\
   %{!E:%{!M:%{!MM:\
     %{save-temps|no-integrated-cpp:cc1plus -E\
%(cpp_options) %2 -o %{save-temps:%b.ii} %{!save-temps:%g.ii} \n}\
```


Populated Plugin Data Structure for LTO:

gcc/lto/lang-specs.h

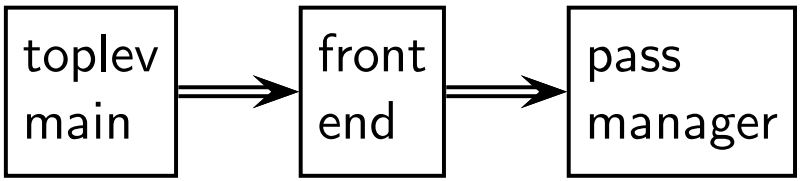
```
/* LTO contributions to the "compilers" array in gcc.c.  */

  {"@lto", "lto1 %(cc1_options) %i %{!fsyntax-only:%(invoke_as)}",
   /*cpp_spec=*/NULL, /*combinable=*/1, /*needs_preprocessing=*/0},
```

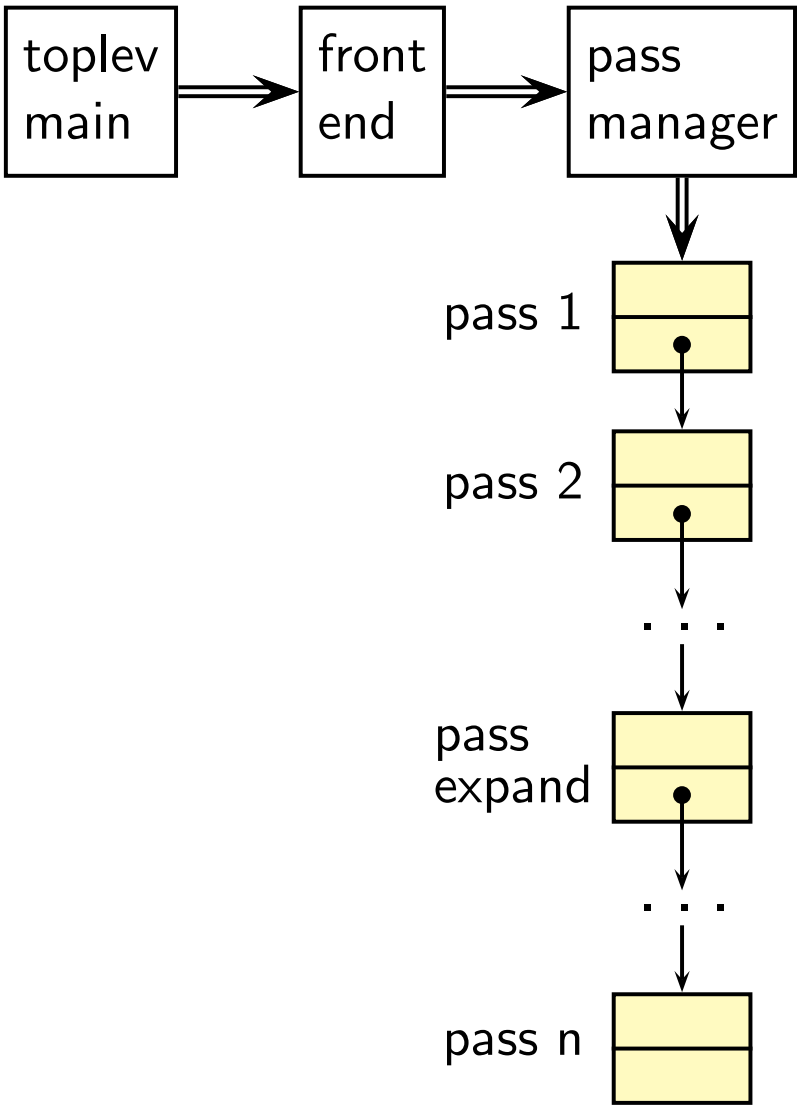

What about the Files to be Procecced by the Linker?

- Linking is the last step
- Every file is passed on to linker unless it is suppressed
- If a translator is not found, input file is assumed to be a file for linker

Plugin Structure in `cc1`

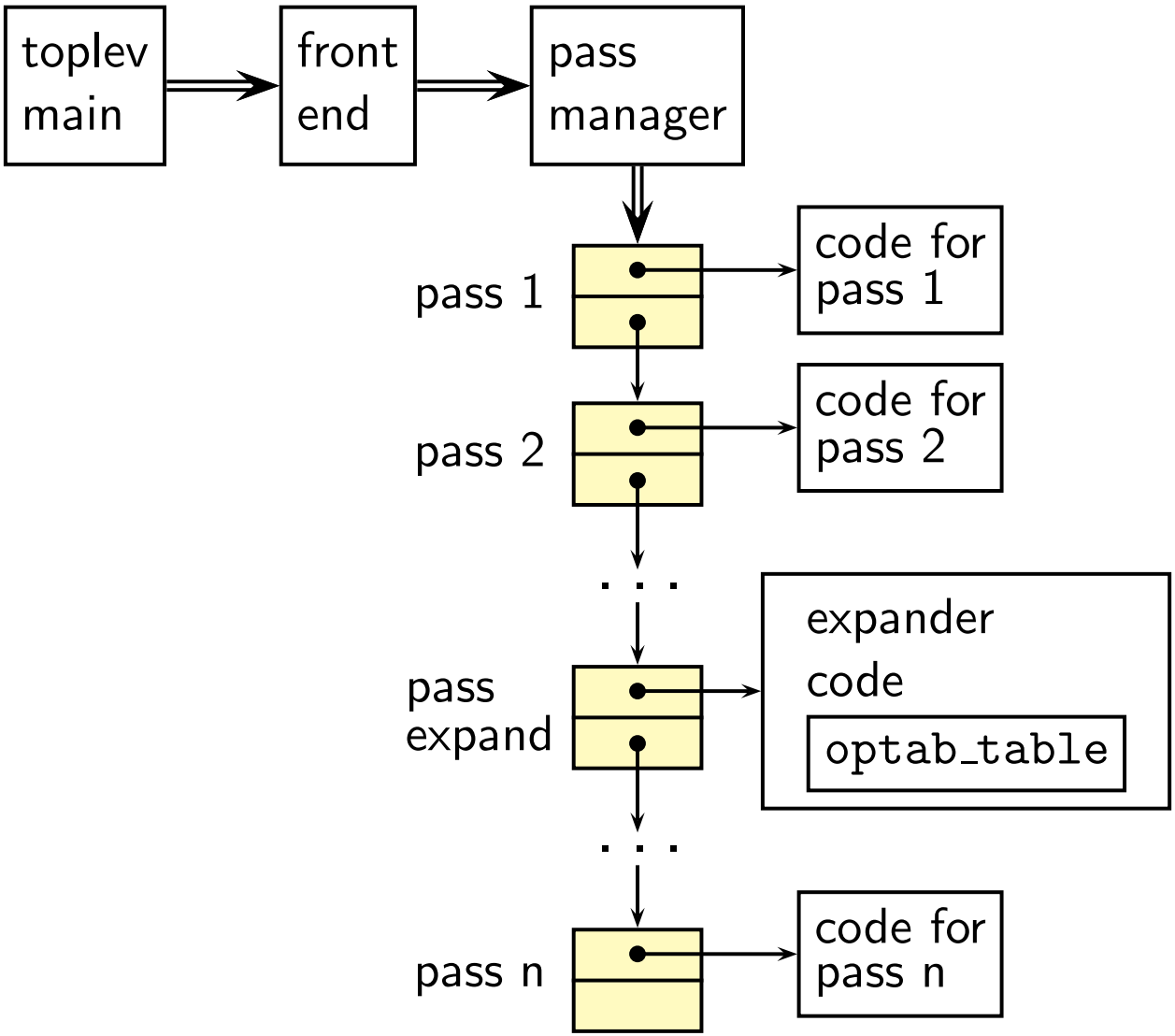


Plugin Structure in `cc1`

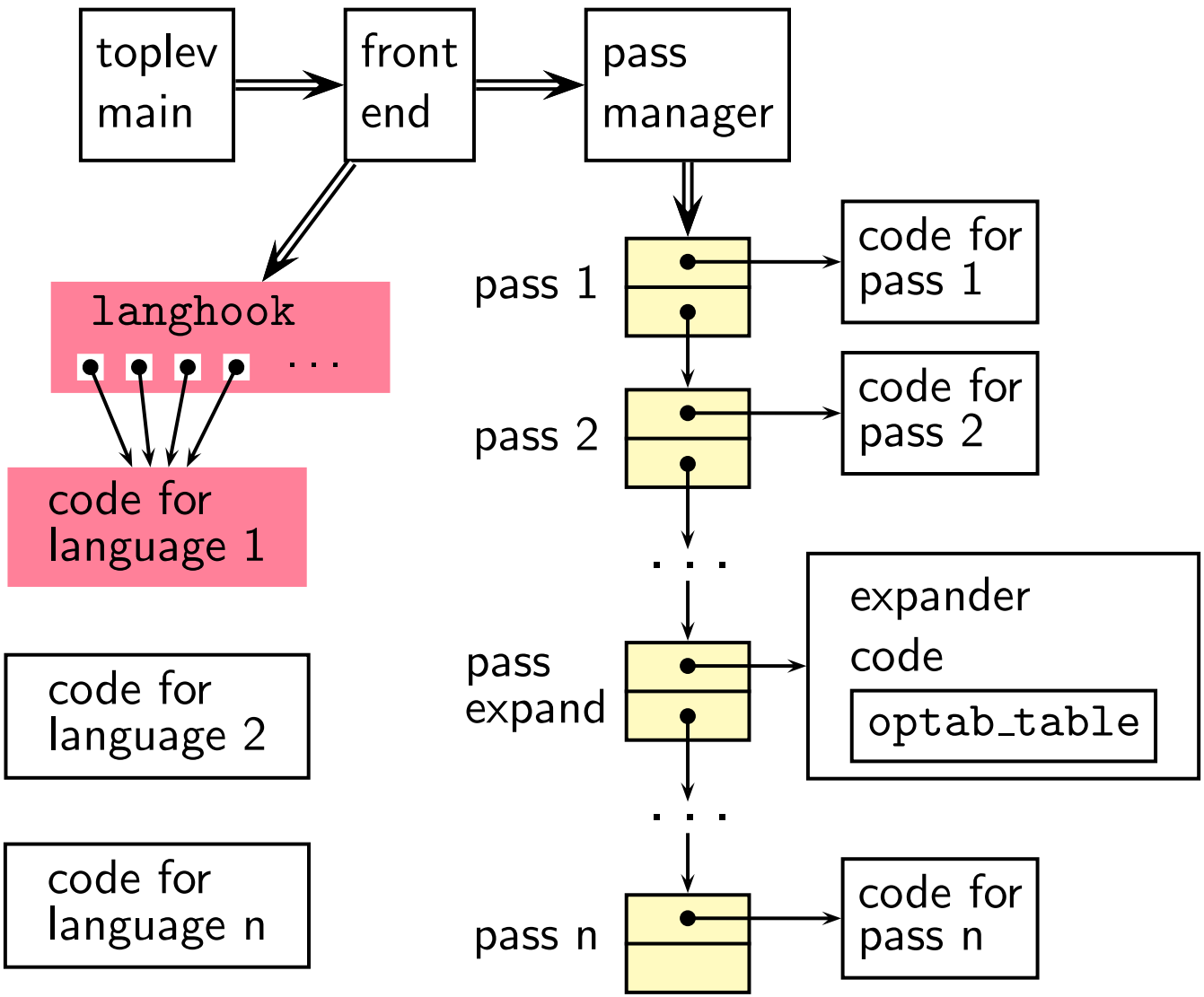


double arrow represents control flow whereas single arrow represents pointer or index

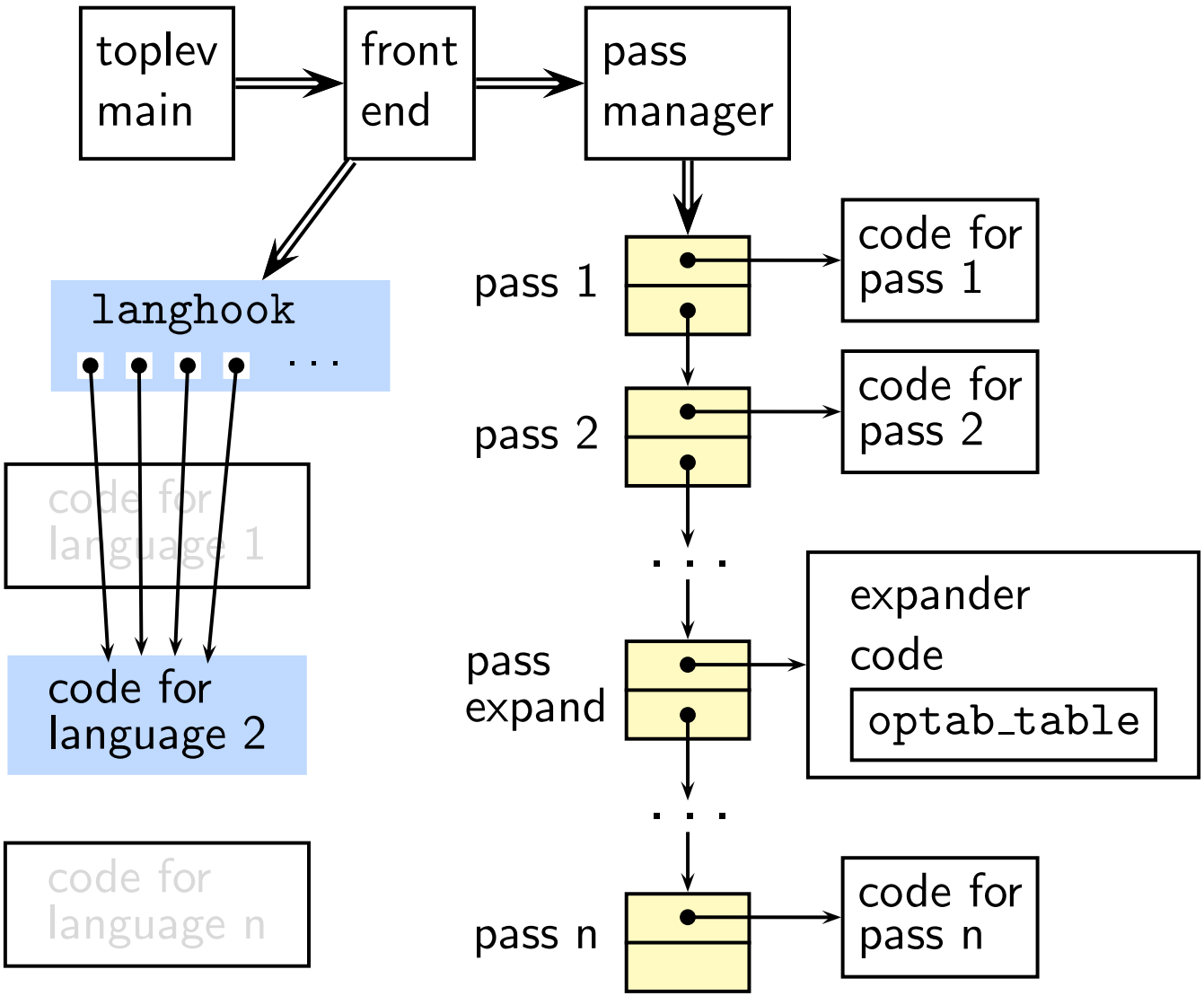
Plugin Structure in `cc1`



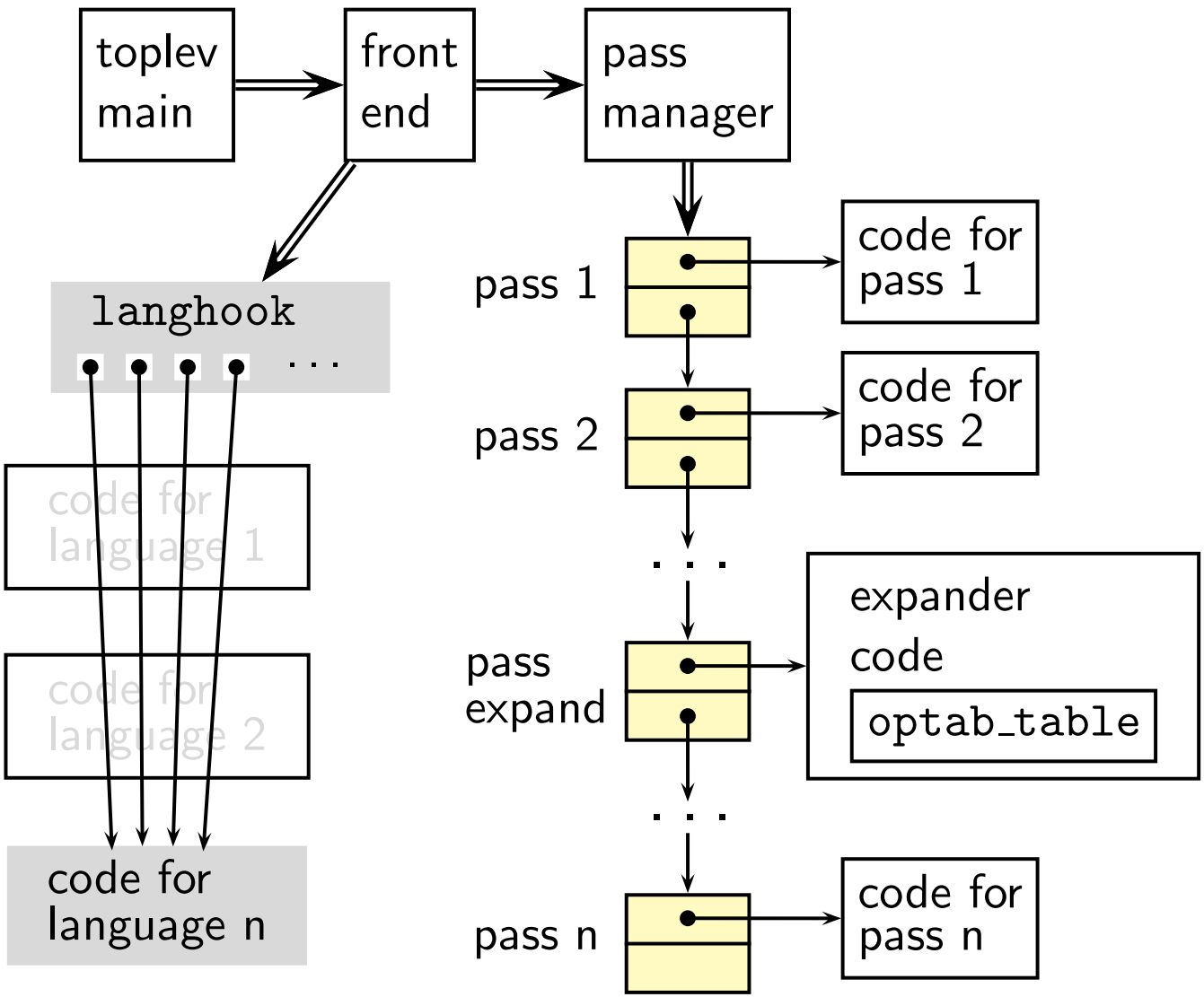
Plugin Structure in `cc1`



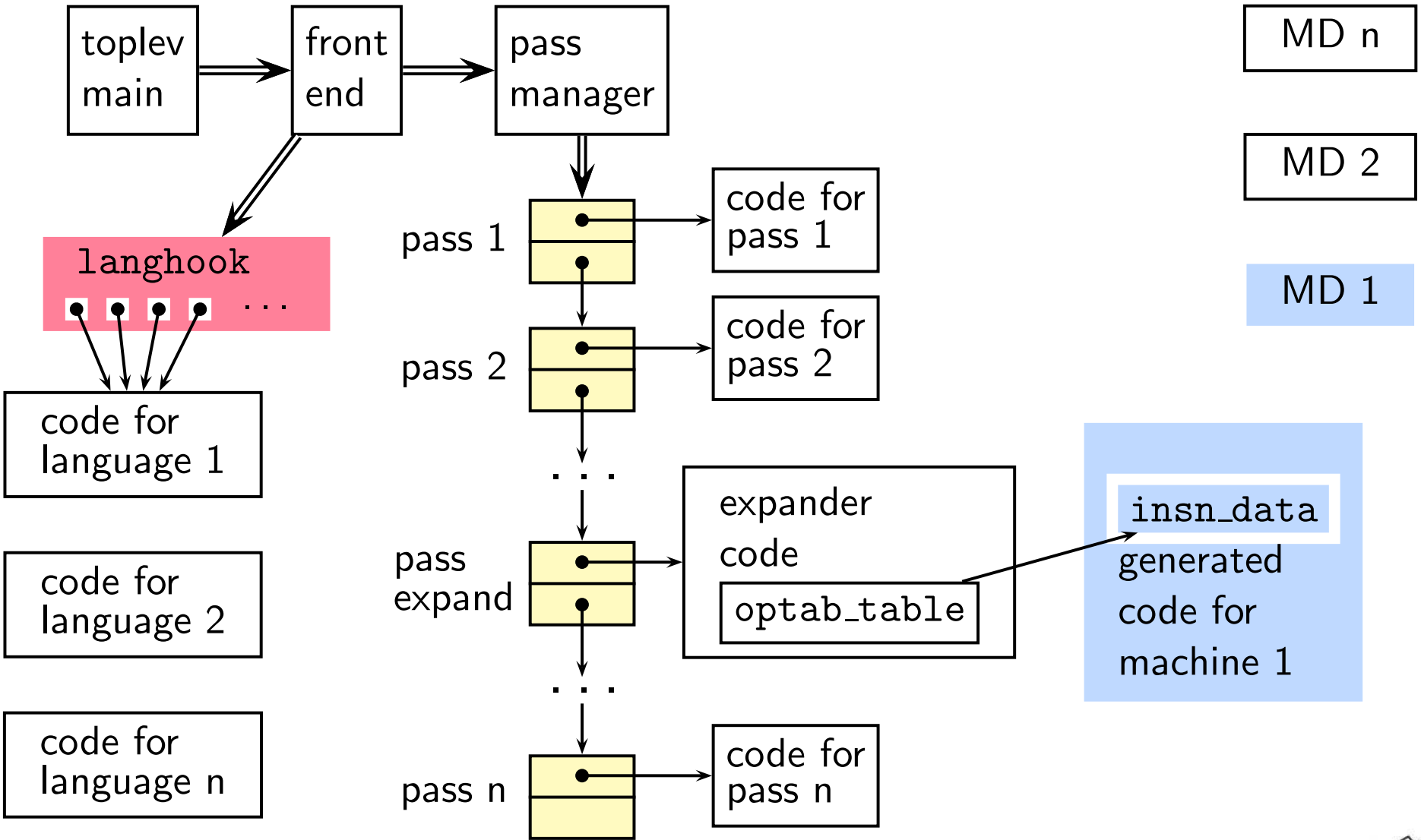
Plugin Structure in `cc1`



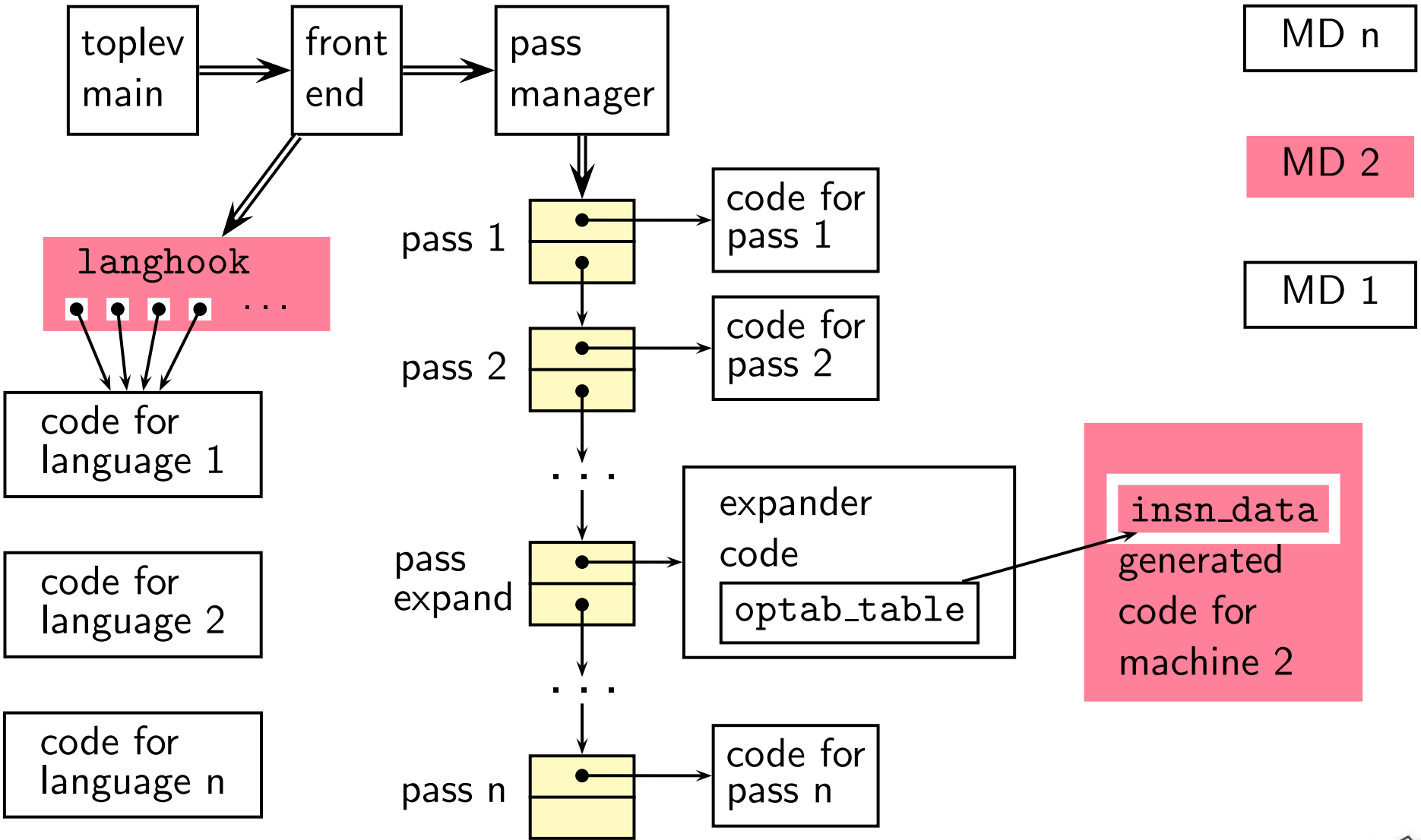
Plugin Structure in `cc1`



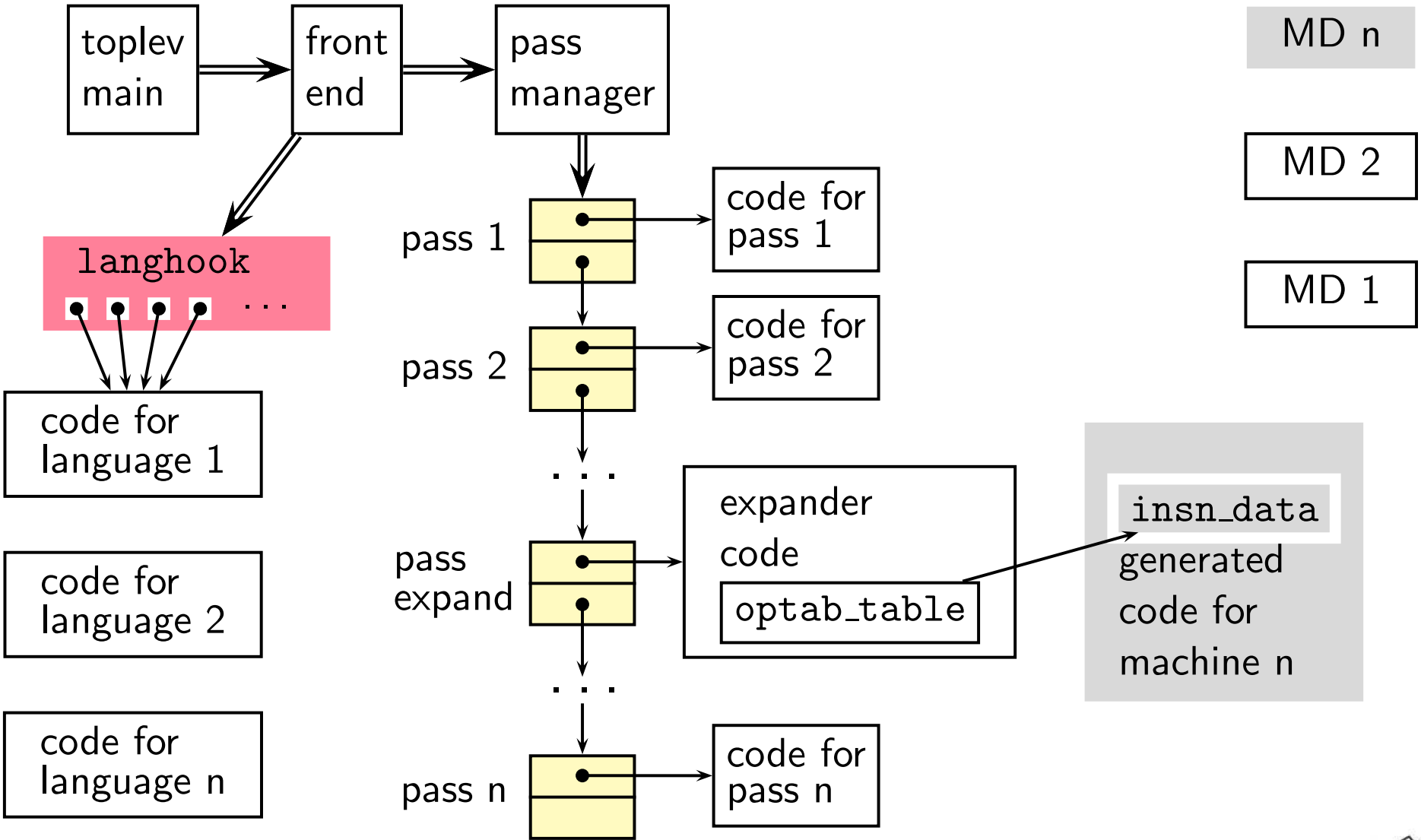
Plugin Structure in `cc1`



Plugin Structure in `cc1`



Plugin Structure in `cc1`



Front End Plugin

Important fields of `struct lang_hooks` instantiated for C

```
#define LANG_HOOKS_FINISH c_common_finish
#define LANG_HOOKS_EXPAND_EXPR c_expand_expr
#define LANG_HOOKS_PARSE_FILE c_common_parse_file
#define LANG_HOOKS_WRITE_GLOBALS c_write_global_declarations
```


Plugins for Intraprocedural Passes

```
struct opt_pass
{
  enum opt_pass_type type;
  const char *name;
  bool (*gate) (void);
  unsigned int (*execute) (void);
  struct opt_pass *sub;
  struct opt_pass *next;
  int static_pass_number;
  timevar_id_t tv_id;
  unsigned int properties_required;
  unsigned int properties_provided;
  unsigned int properties_destroyed;
  unsigned int todo_flags_start;
  unsigned int todo_flags_finish;
};
```

```
struct gimple_opt_pass
{
  struct opt_pass pass;
};

struct rtl_opt_pass
{
  struct opt_pass pass;
};
```


Plugins for Interprocedural Passes

```
struct ipa_opt_pass_d
{
  struct opt_pass pass;
  void (*generate_summary) (void);
  void (*write_summary) (struct cgraph_node_set_def *);
  void (*read_summary) (void);
  void (*function_read_summary) (struct cgraph_node *);
  void (*stmt_fixup) (struct cgraph_node *, gimple *);
  unsigned int function_transform_todo_flags_start;
  unsigned int (*function_transform) (struct cgraph_node *);
  void (*variable_transform) (struct varpool_node *);
};

struct simple_ipa_opt_pass
{
  struct opt_pass pass;
};
```


Predefined Pass Lists

Pass Name	Purpose
`all_lowering_passes`	Lowering
`all_small_ipa_passes`	Early optimization passes. Invokes intraprocedural passes over the call graph.
`all_regular_ipa_passes`	
`all_lto_gen_passes`	
`all_passes`	Intraprocedural passes on GIMPLE and RTL

Registering a Pass

1. Write the driver function in your file
2. Declare your pass in file `tree-pass.h`:
   `extern struct gimple_opt_pass your_pass_name;`
3. Add your pass to the appropriate pass list in `init_optimization_passes()` using the macro `NEXT_PASS`
4. Add your file details to `$SOURCE/gcc/Makefile.in`
5. Configure and build gcc
   (For simplicity, you can make `cc1` only)
6. Debug `cc1` using ddd/gdb if need arises

Part 3

Dynamic Plugins in GCC

Dynamic Plugins

- Supported on platforms that support `-ldl -rdynamic`
- Loaded using `dlopen` and invoked at pre-determined locations in the compilation process
- Command line option

  `-fplugin=/path/to/name.so`

  Arguments required can be supplied as name-value pairs

Specifying an Example Pass

```
struct simple_ipa_opt_pass pass_plugin = {
  {
    SIMPLE_IPA_PASS,
    "dynamic_plug",                 /*  name */
    0,                              /*  gate */
    execute_pass_plugin,            /*  execute */
    NULL,                           /*  sub */
    NULL,                           /*  next */
    0,                              /*  static pass number */
    TV_INTEGRATION,                 /*  tv_id */
    0,                              /*  properties required */
    0,                              /*  properties provided */
    0,                              /*  properties destroyed */
    0,                              /*  todo_flags start */
    0                               /*  todo_flags end */
  }
};
```

Registering Our Pass as a Dynamic Plugin

```
struct register_pass_info pass_info = {
  &(pass_plugin.pass),      /* Address of new pass, here, the
                               struct opt_pass field of
                               simple_ipa_opt_pass defined above */
  "pta",                    /* Name of the reference pass for
                               hooking up the new pass.  */
  0,                        /* Insert the pass at the specified
                               instance number of the reference
                               pass. Do it for every instance if
                               it is 0. */
  PASS_POS_INSERT_AFTER     /* how to insert the new pass:
                               before, after, or replace. Here we
                               are inserting our pass the pass
                               named pta */
};
```


Registering Callback for Our Pass for a Dynamic Plugins

```
int plugin_init(struct plugin_name_args *plugin_info,
    struct plugin_gcc_version *version)
{ /*  Plugins are activiated using this callback  */

  register_callback (
      plugin_info->base_name,     /* char *name: Plugin name,
                                     could be any name.
                                     plugin_info->base_name
                                     gives this filename */
      PLUGIN_PASS_MANAGER_SETUP,  /* int event: The event code.
                                     Here, setting up a new
                                     pass */
      NULL,                       /* The function that handles
                                     the event */
      &pass_info);                /* plugin specific data */

  return 0;
}
```


Part 4

Flow of Control Flow in the Generated Compiler

Walking the Maze of a Large Code Base

- If you use conventional editors such as `vi` or `emacs`
  - Use cscope

    ```
    cd $SOURCE
    cscope -R
    ```

  - Use ctags

    ```
    cd $SOURCE
    ctags -R
    ```

    Make sure you use `exeburant-ctags`
- Or use IDE such as `eclipse`

gcc Driver Control Flow

```
main   /* In file gcc.c */
   validate_all_switches
   lookup_compiler
   do_spec
      do_spec_2
         do_spec_1  /* Get the name of the compiler */
      execute
         pex_init
         pex_run
            pex_run_in_environment
               obj->funcs->exec_child
```


gcc Driver Control Flow

```
main   /* In file gcc.c */
   validate_all_switches
   lookup_compiler
   do_spec
      do_spec_2
         do_spec_1  /*
      execute
         pex_init
         pex_run
            pex_run_in
               obj->fu
```

Observations

- All compilers are invoked by this driver
- Assembler is also invoked by this driver
- Linker is invoked in the end by default

cc1 Top Level Control Flow

```
main
   toplev_main   /* In file toplev.c */
     decode_options
     do_compile
        compile_file
          lang_hooks.parse_file => c_common_parse_file
          lang_hooks.decls.final_write_globals =>
                                        c_write_global_declarations
          targetm.asm_out.file_end
     finalize
```


cc1 Top Level Control Flow

```
main
   toplev_main   /* In file toplev.c */
     decode_options
     do_compile
        compile_file
          lang_hooks.p
          lang_hooks.d
                                                    larations
          targetm.asm_
     finalize
```

Observations

- The entire compilation is driven by functions specified in language hooks
- Not a good design!

cc1 Control Flow: Parsing for C

```
lang_hooks.parse_file => c_common_parse_file
   c_parse_file
         c_parser_translation_unit
            c_parser_external_declaration
               c_parser_declaration_or_fndef
                  c_parser_declspecs /* parse declarations */
                  c_parser_compound_statement
                  finish_function    /* finish parsing */
                     c_genericize
                     cgraph_finalize_function
                     /* finalize AST of a function */
```


cc1 Control Flow: Parsing for C

```
lang_hooks.parse_file => c_common_parse_file
   c_parse_file
         c_parser_translation_unit
            c_parser_e
               c_parse
                  c_pa                                    ions */
                  c_pa
                  fini                                    */
                     c
                     c
                     /
```

Observations

- GCC has moved to a recursive descent parser from version 4.1.0
- Earlier parser was generated using Bison specification

cc1 Control Flow: Lowering Passes for C

```
lang_hooks.decls.final_write_globals =>
                                    c_write_global_declarations
    cgraph_finalize_compilation_unit
        cgraph_analyze_functions      /* Create GIMPLE */
           cgraph_analyze_function
                gimplify_function_tree
                   gimplify_body
                      gimplify_stmt
                         gimplify_expr
           cgraph_lower_function     /* Intraprocedural */
              tree_lowering_passes
                 execute_pass_list (all_lowering_passes)
```


cc1 Control Flow: Lowering Passes for C

```
lang_hooks.decls.final_write_globals =>
                                    c_write_global_declarations
    cgraph_finalize_compilation_unit
        cgraph_analyze                                          */
            cgraph_anal
                gimpli
                    gim


            cgraph_lowe                                         */
                tree_low
                    execu                                       s)
```

Observations

- Lowering passes are language independent
- Yet they are being called from a function in language hooks
- Not a good design!

Organization of Passes

Order	Task	IR	Level	Pass data structure
1	Lowering	GIMPLE	Intra	`gimple_opt_pass`
2	Optimizations	GIMPLE	Inter	`ipa_opt_pass`
3	Optimizations	GIMPLE	Intra	`gimple_opt_pass`
4	RTL Generation	GIMPLE	Intra	`rtl_opt_pass`
5	Optimization	RTL	Intra	`rtl_opt_pass`

cc1 Control Flow: Optimization and Code Generation Passes

```
cgraph_analyze_function          /* Create GIMPLE */
...                              /* previous slide */
cgraph_optimize
   ipa_passes
      execute_ipa_pass_list(all_small_ipa_passes)  /*!in_lto_p*/
      execute_ipa_summary_passes(all_regular_ipa_passes)
      execute_ipa_summary_passes(all_lto_gen_passes)
      ipa_write_summaries
   cgraph_expand_all_functions
       cgraph_expand_function
       /* Intraprocedural passes on GIMPLE,  */
       /* expansion pass, and passes on RTL. */
            tree_rest_of_compilation
               execute_pass_list (all_passes)
```


cc1 Control Flow: Optimization and Code Generation Passes

```
cgraph_analyze_function          /* Create GIMPLE */
...                              /* previous slide */
cgraph_optimize
   ipa_passes
      execute_ipa_pass_list(all_small_ipa_passes)  /*!in_lto_p*/
      execute_ipa_sum                              ses)
      execute_ipa_sum
      ipa_write_summa
   cgraph_expand_all_
       cgraph_expand_
       /* Intraproced
       /* expansion p
            tree_rest
               execut
```

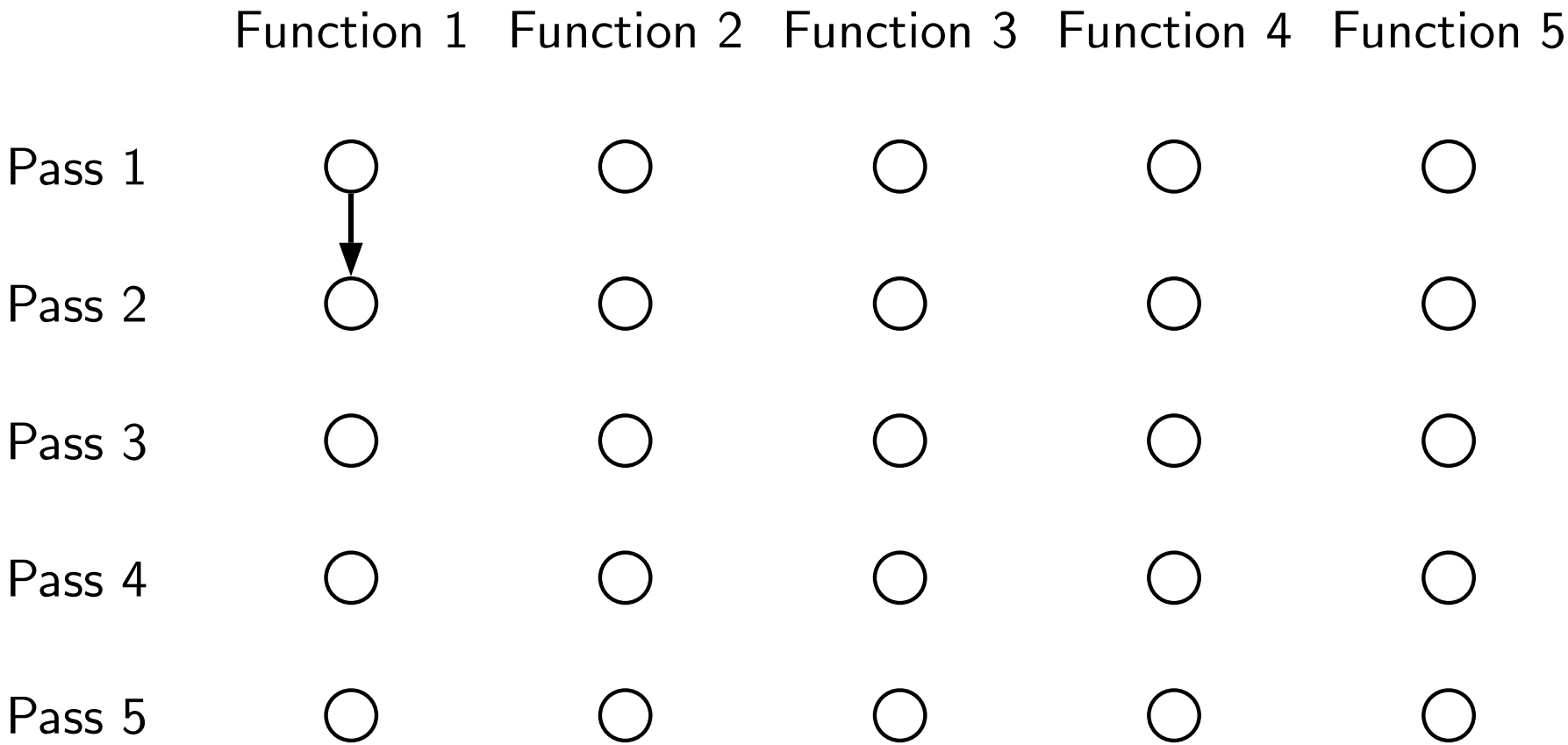
Observations

- Optimization and code generation passes are language independent
- Yet they are being called from a function in language hooks
- Not a good design!

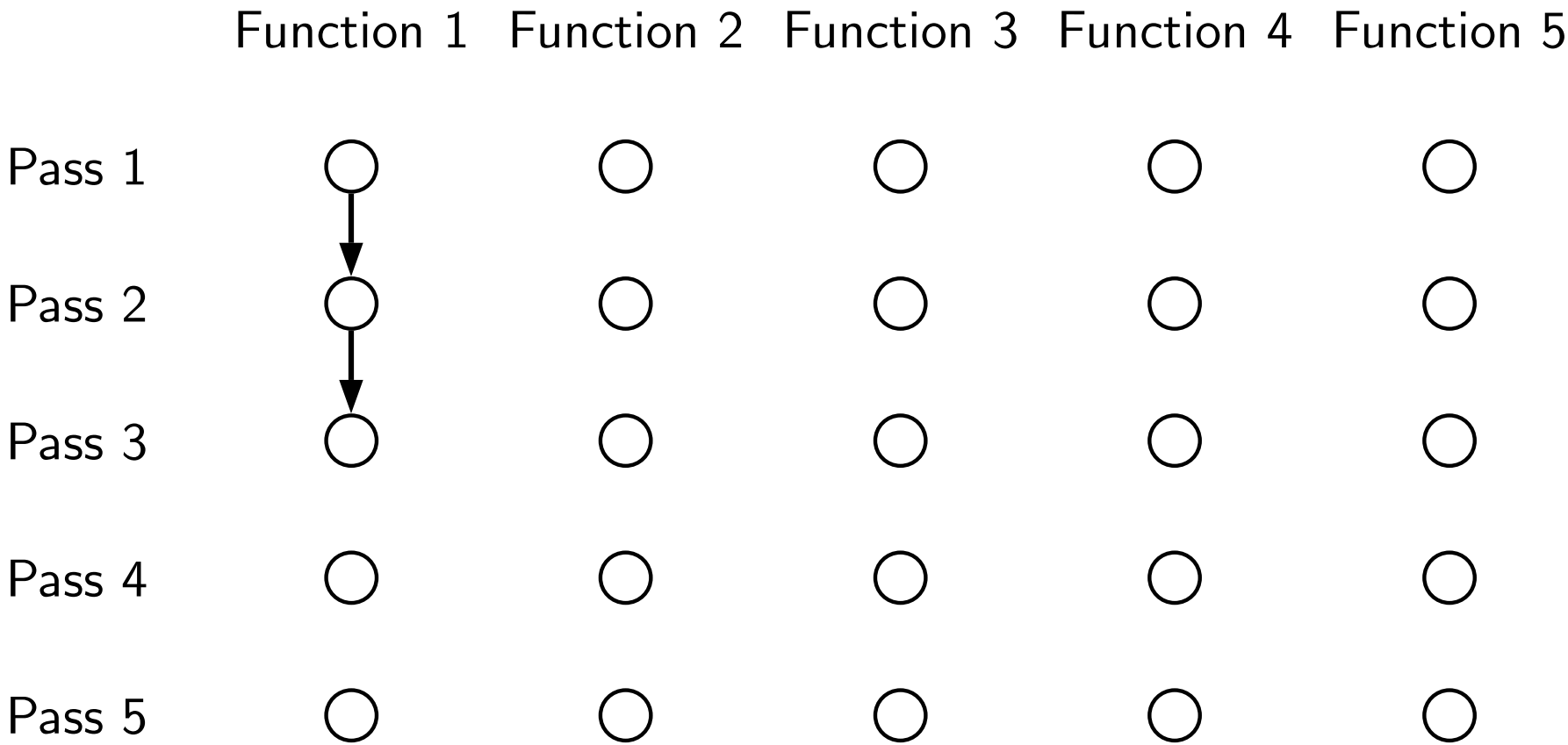
Execution Order in Intraprocedural Passes

	Function 1	Function 2	Function 3	Function 4	Function 5
Pass 1	○	○	○	○	○
Pass 2	○	○	○	○	○
Pass 3	○	○	○	○	○
Pass 4	○	○	○	○	○
Pass 5	○	○	○	○	○

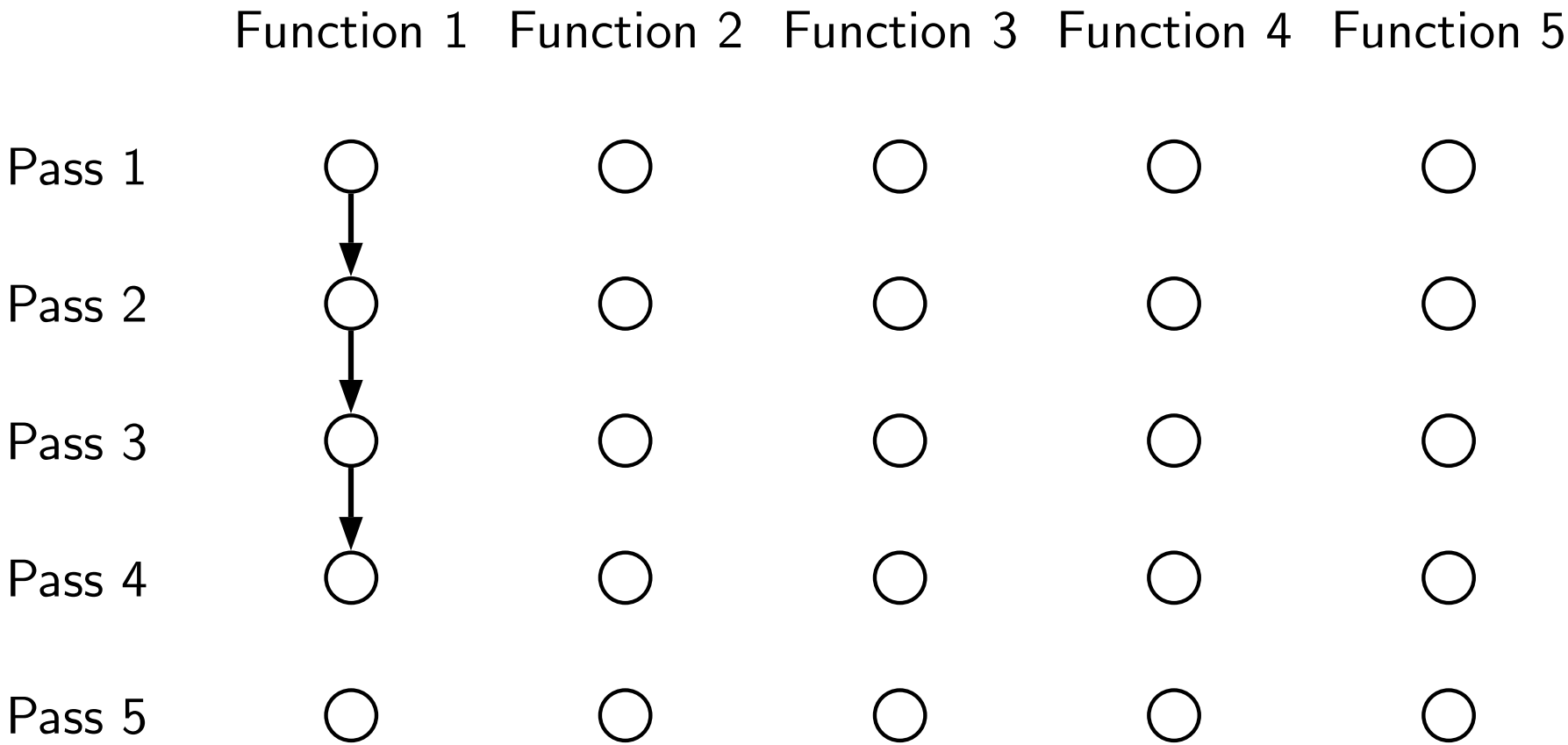
Execution Order in Intraprocedural Passes



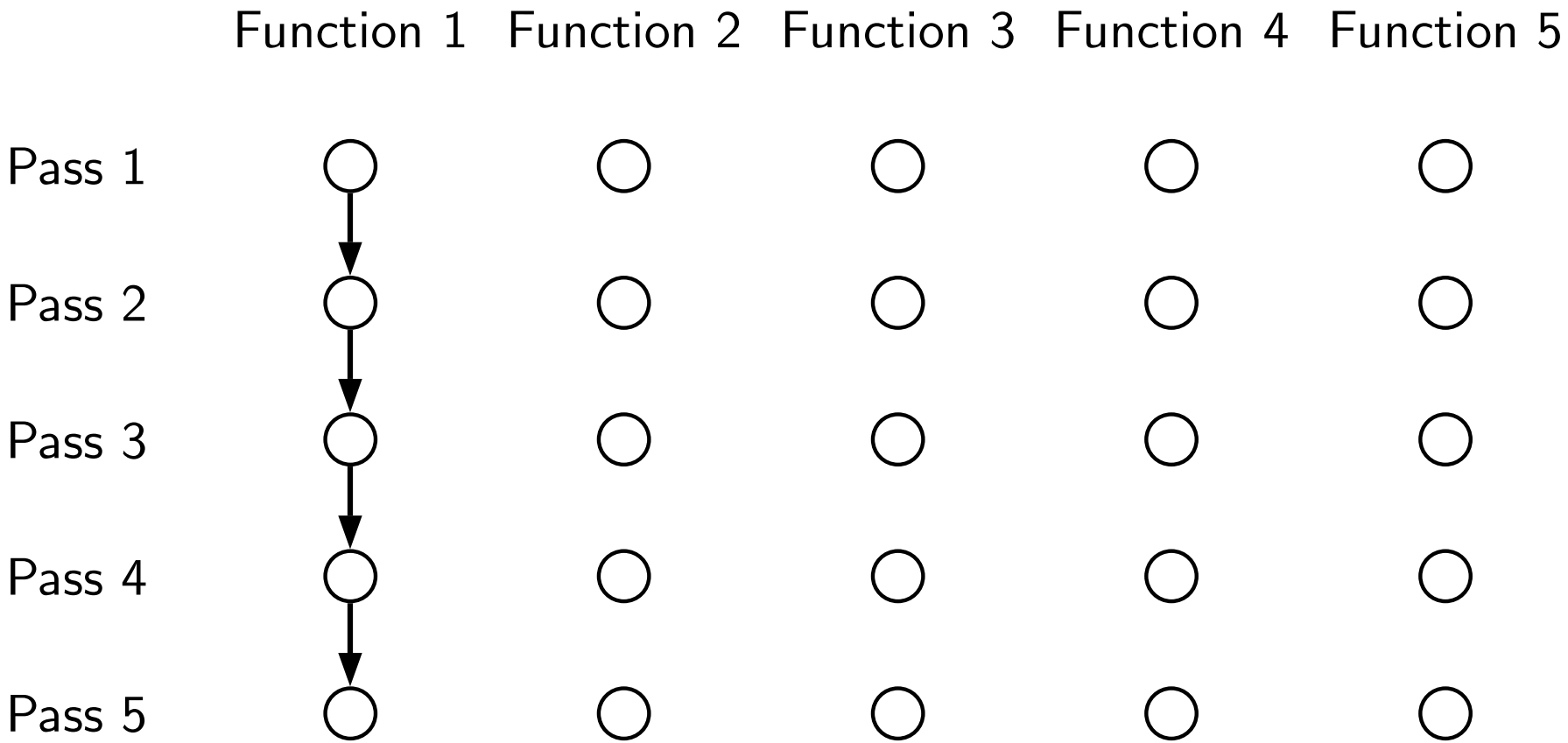
Execution Order in Intraprocedural Passes



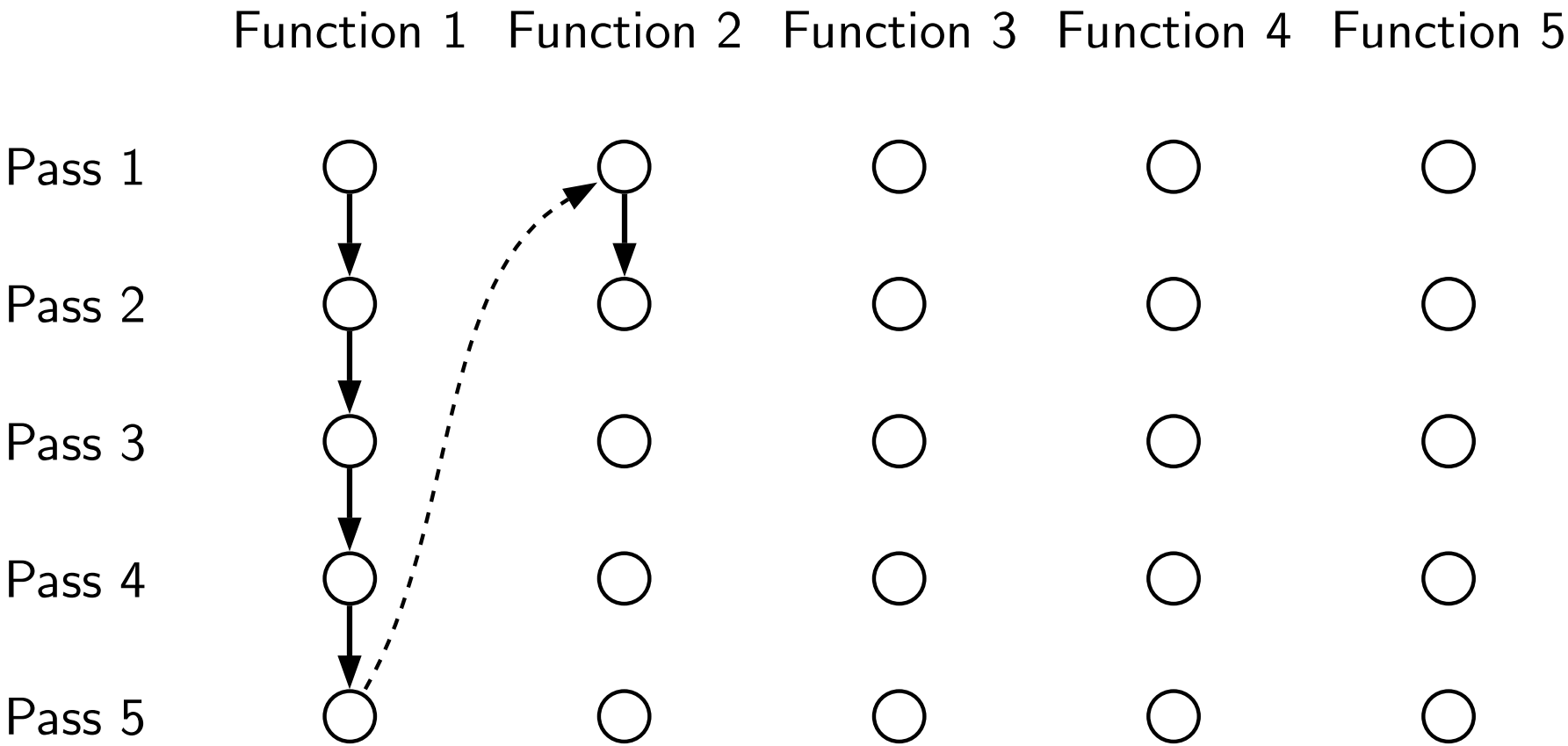
Execution Order in Intraprocedural Passes



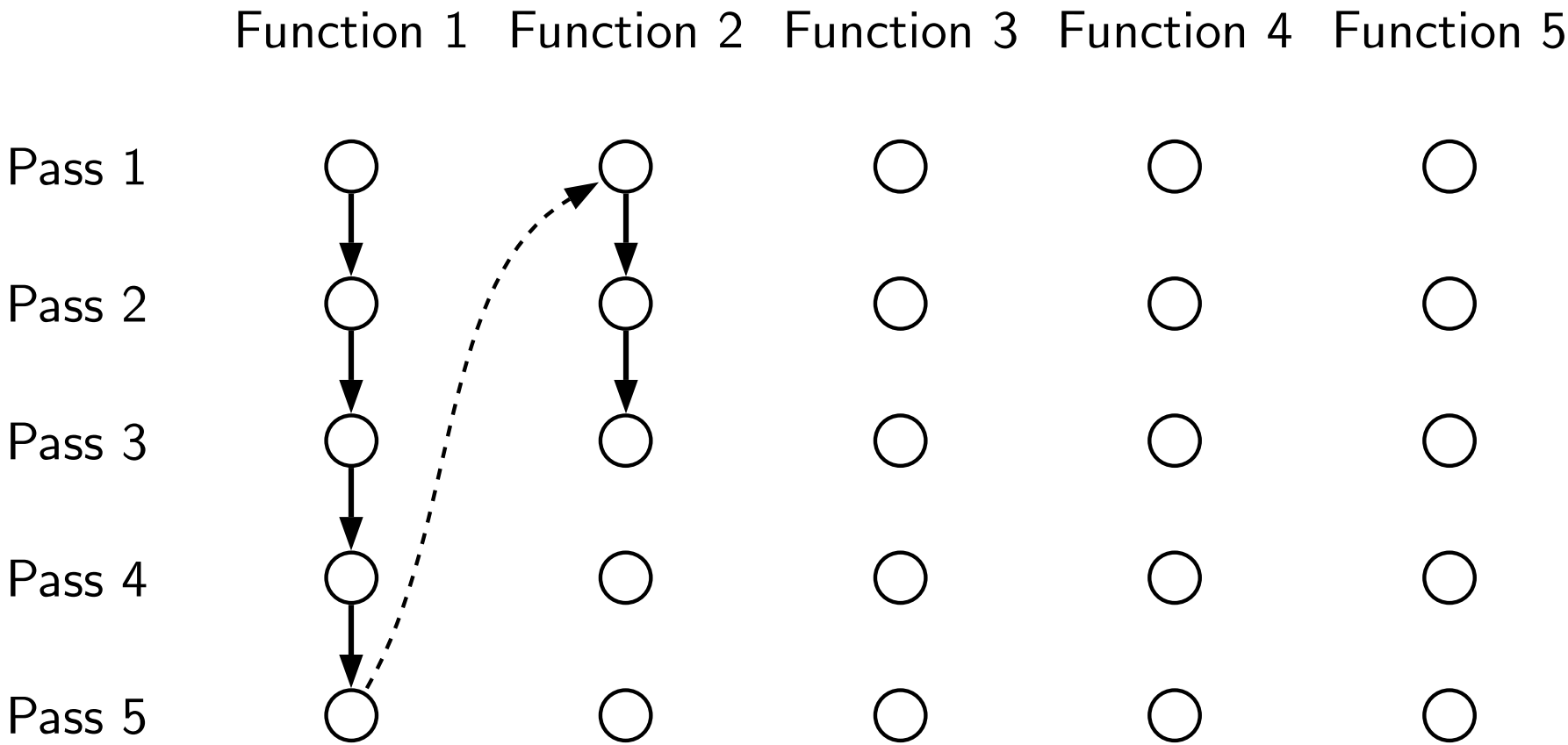
Execution Order in Intraprocedural Passes



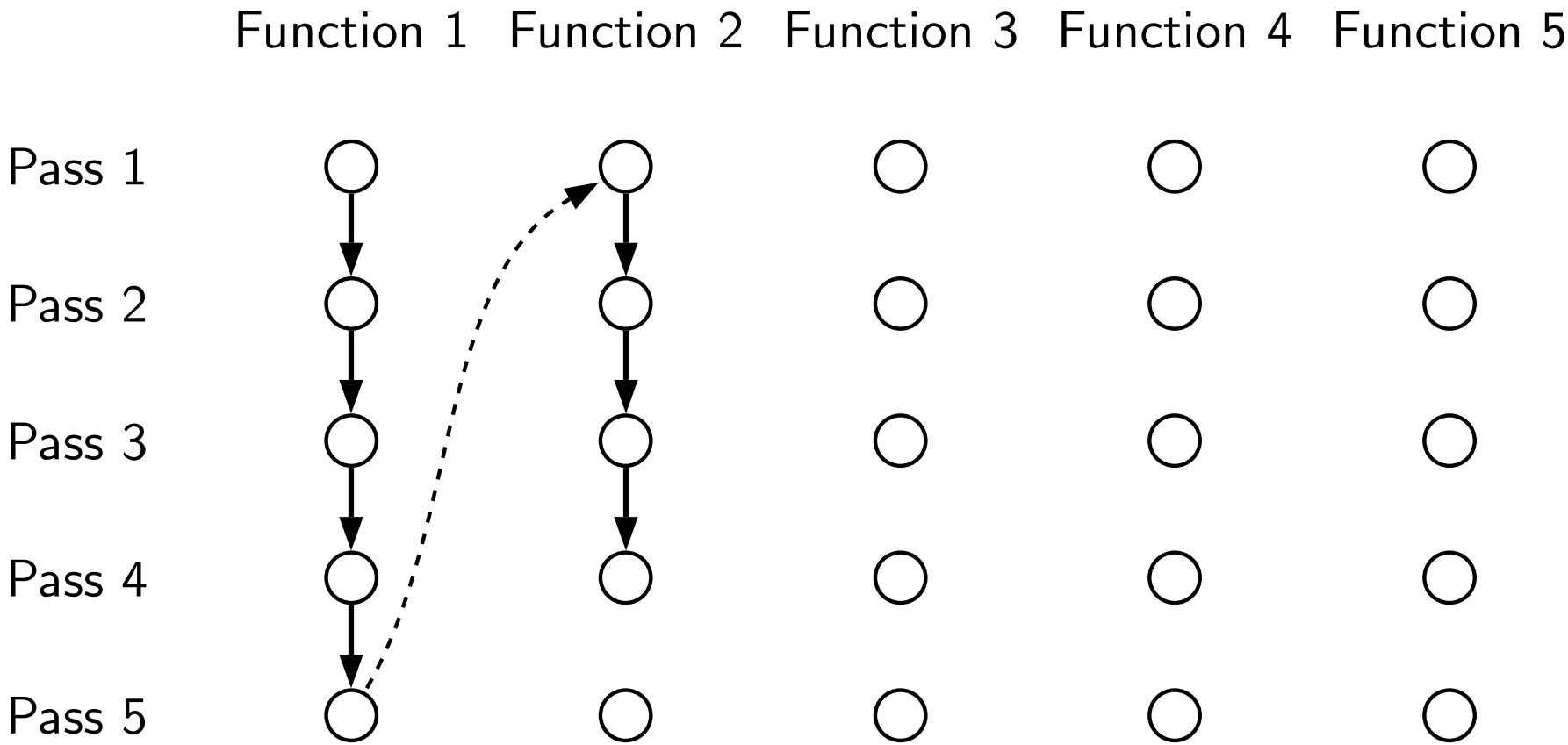
Execution Order in Intraprocedural Passes



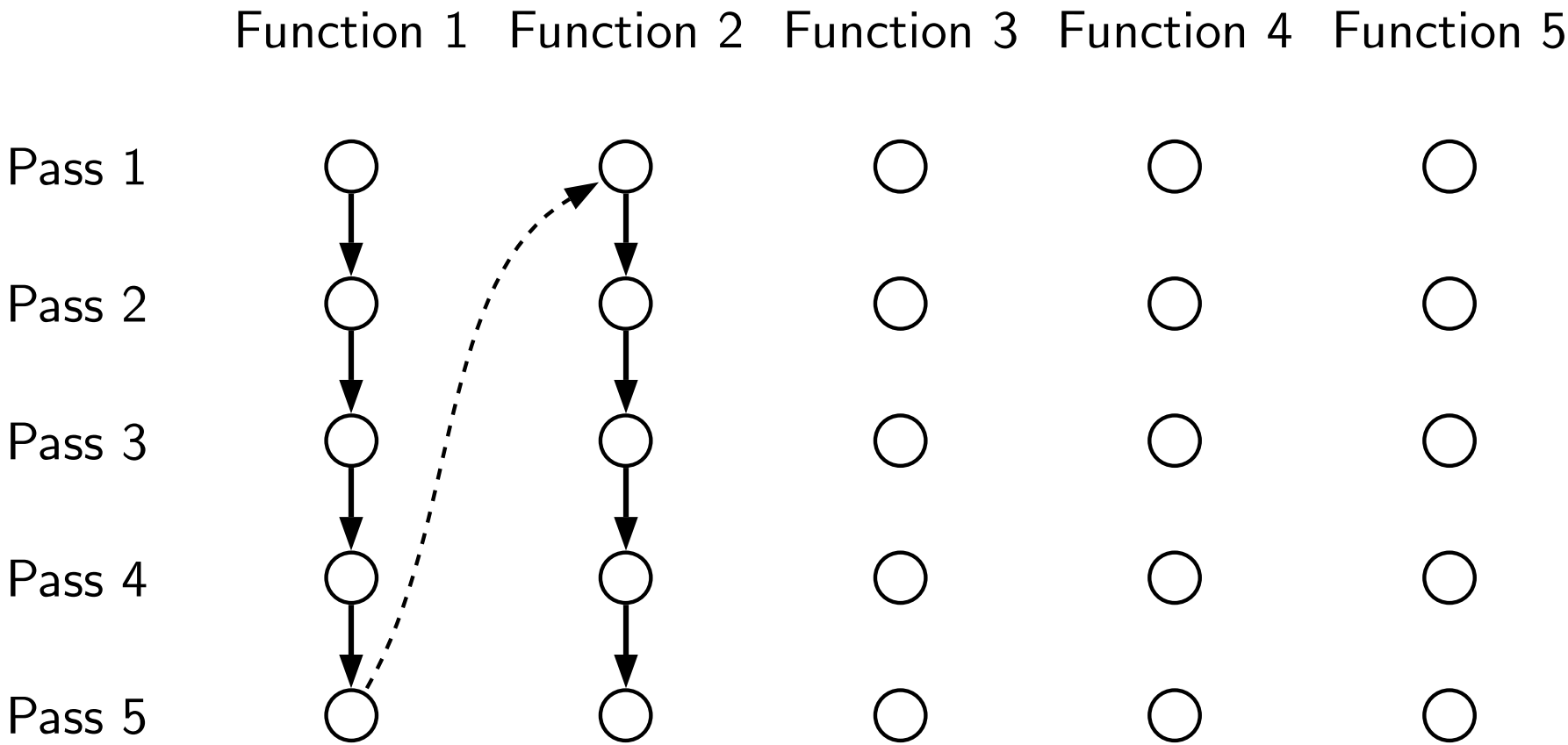
Execution Order in Intraprocedural Passes



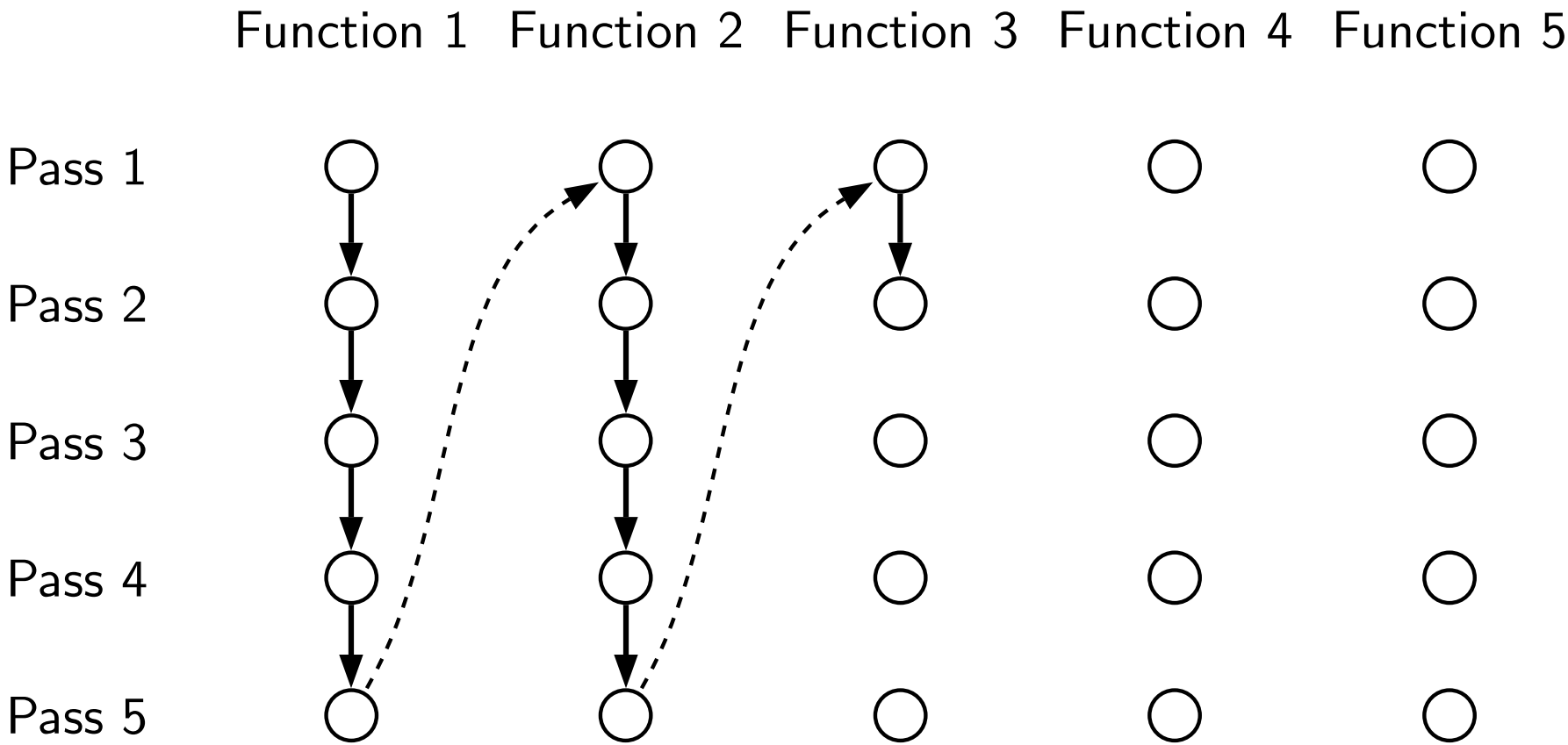
Execution Order in Intraprocedural Passes



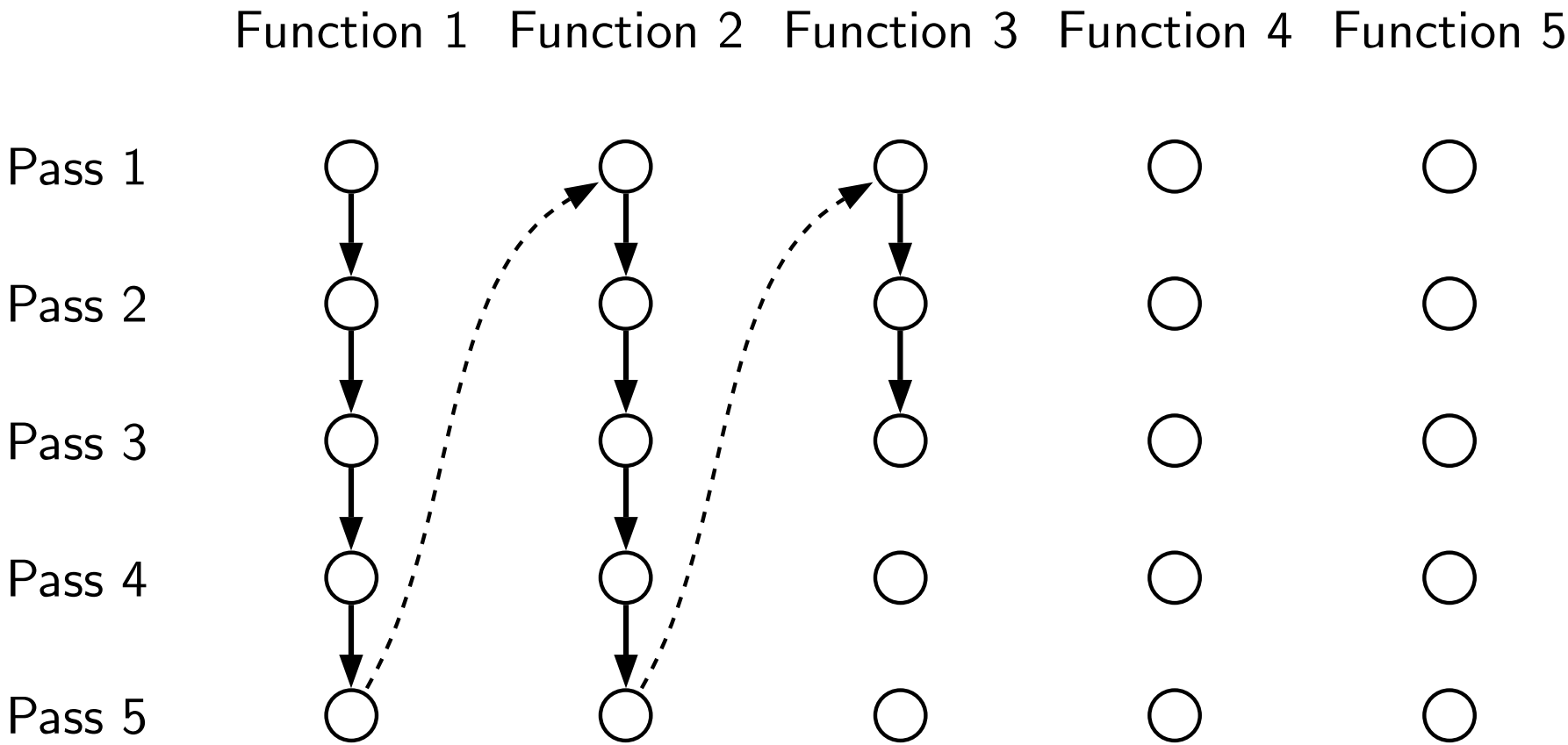
Execution Order in Intraprocedural Passes



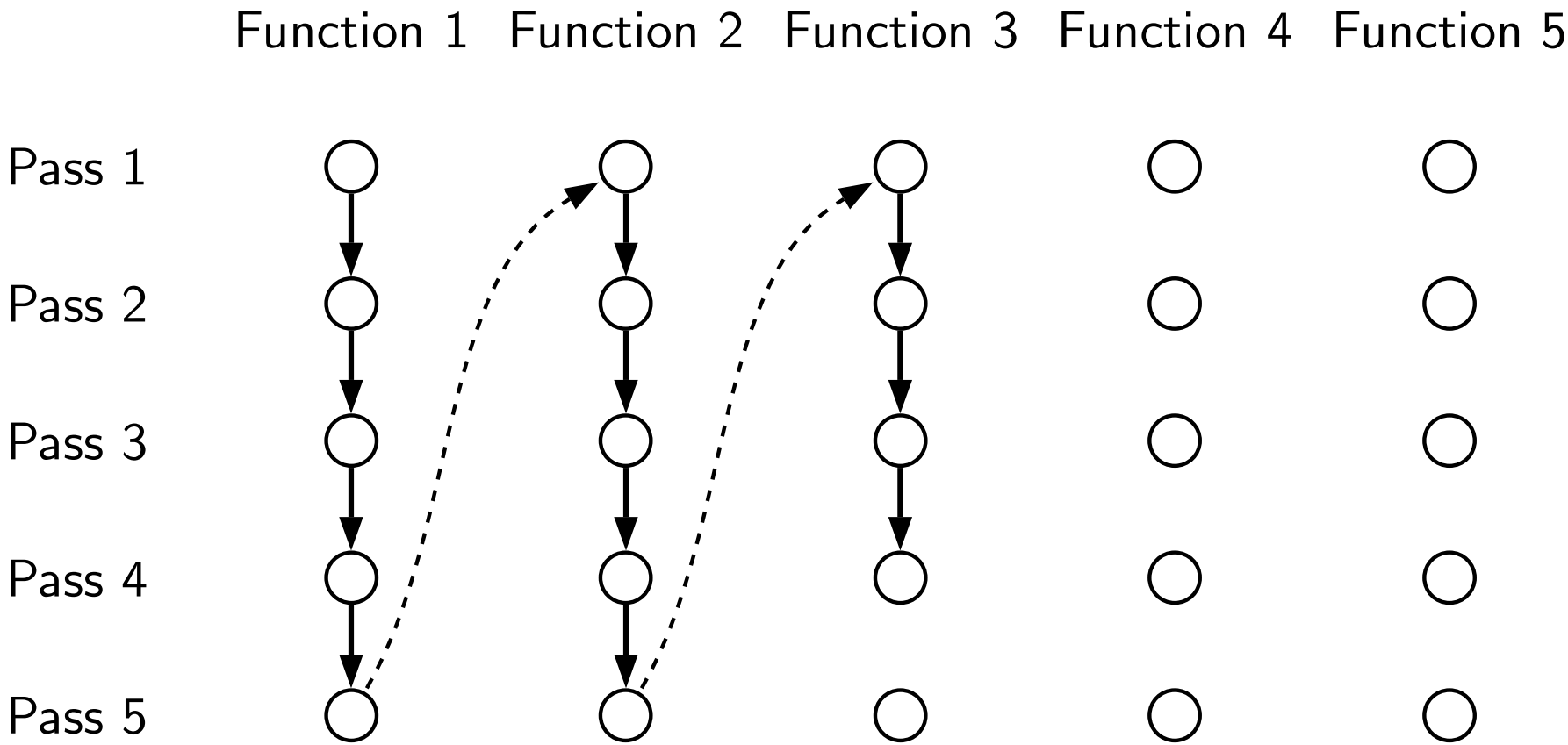
Execution Order in Intraprocedural Passes



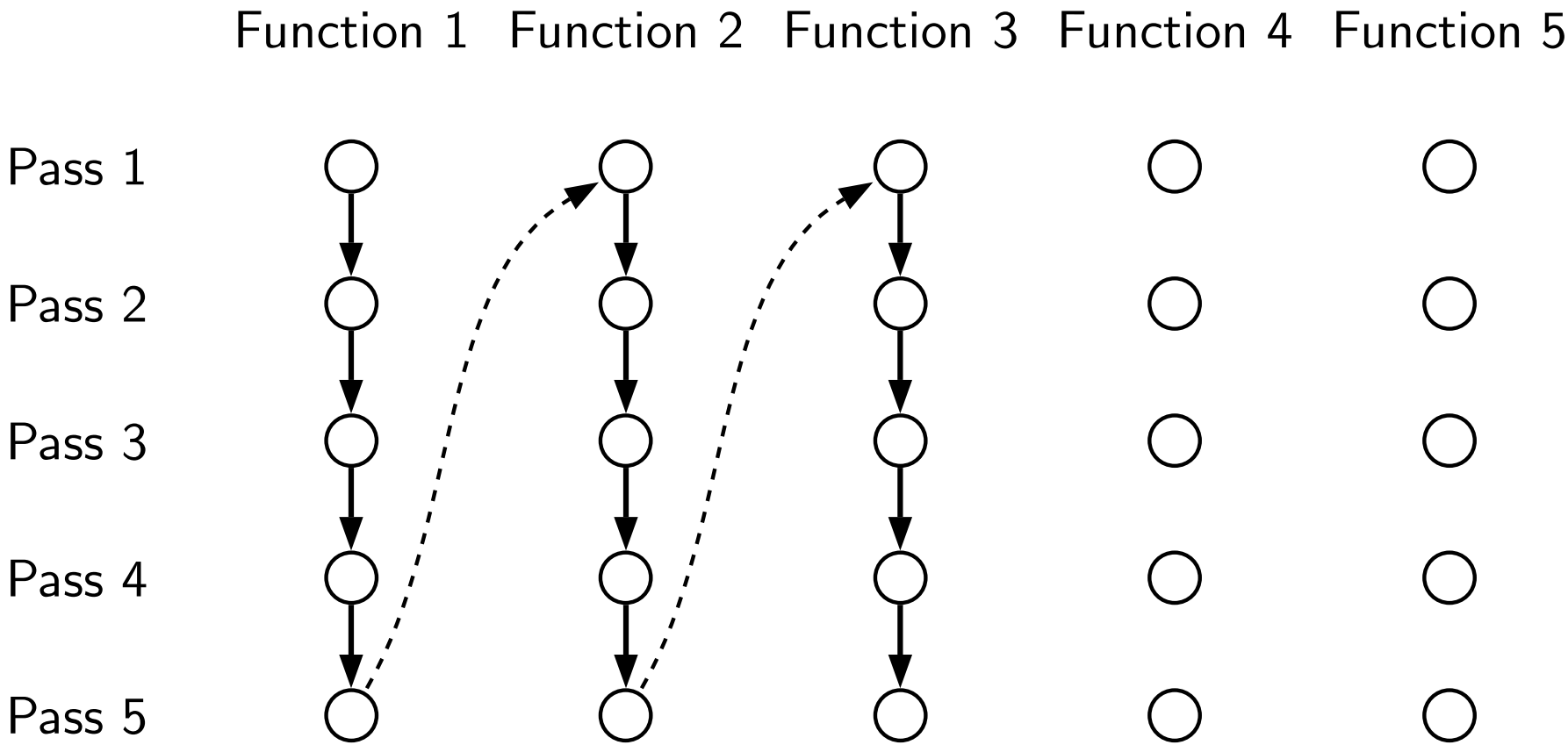
Execution Order in Intraprocedural Passes



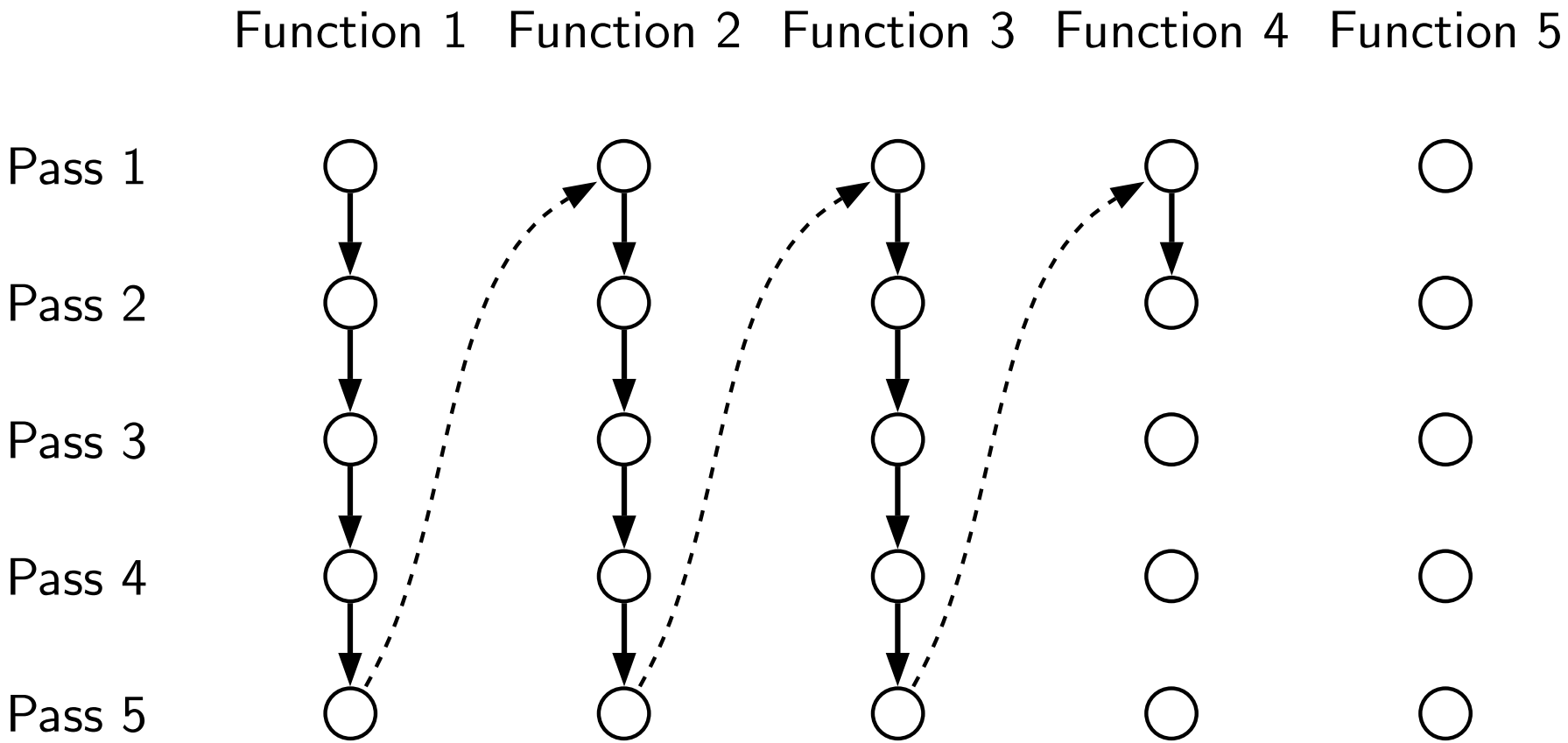
Execution Order in Intraprocedural Passes



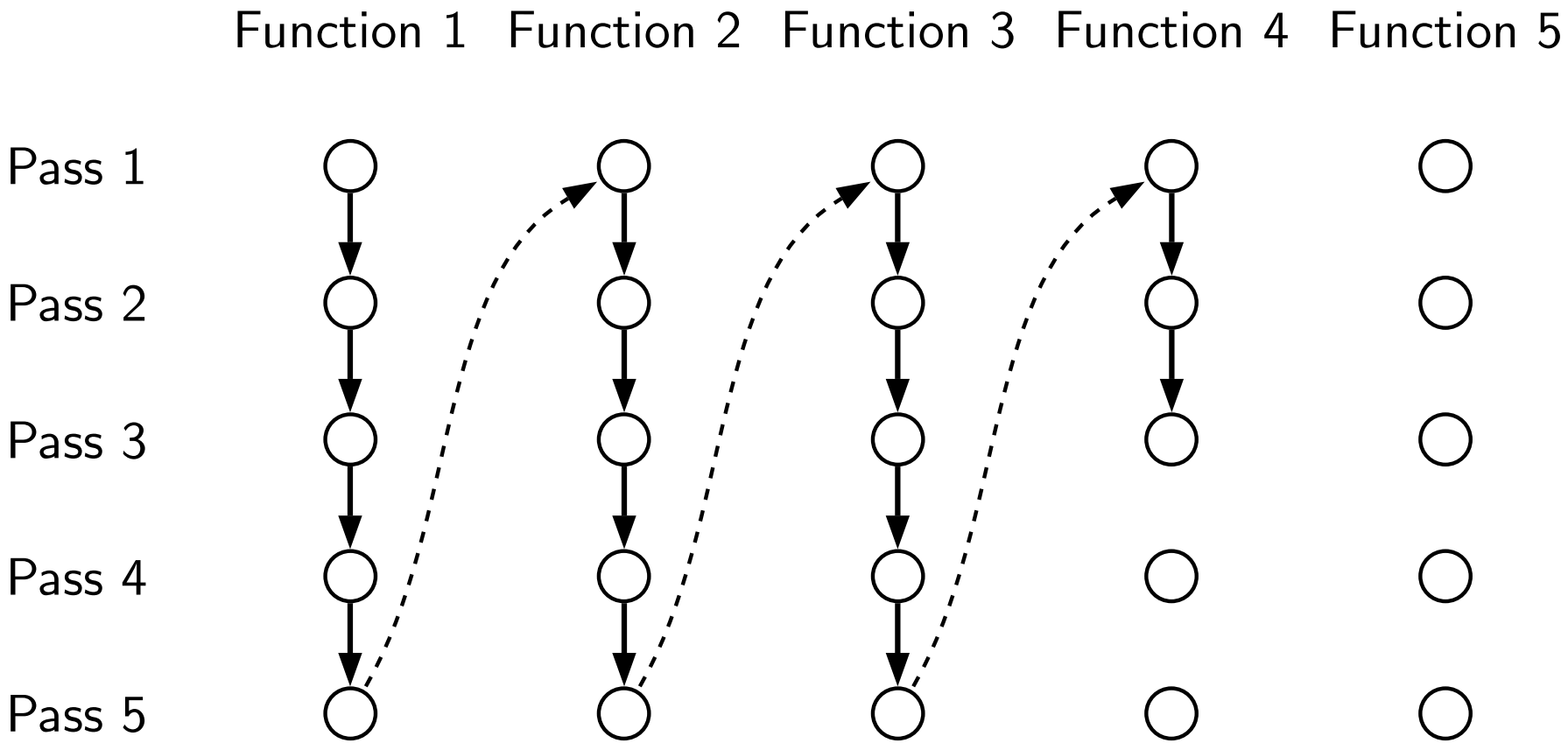
Execution Order in Intraprocedural Passes



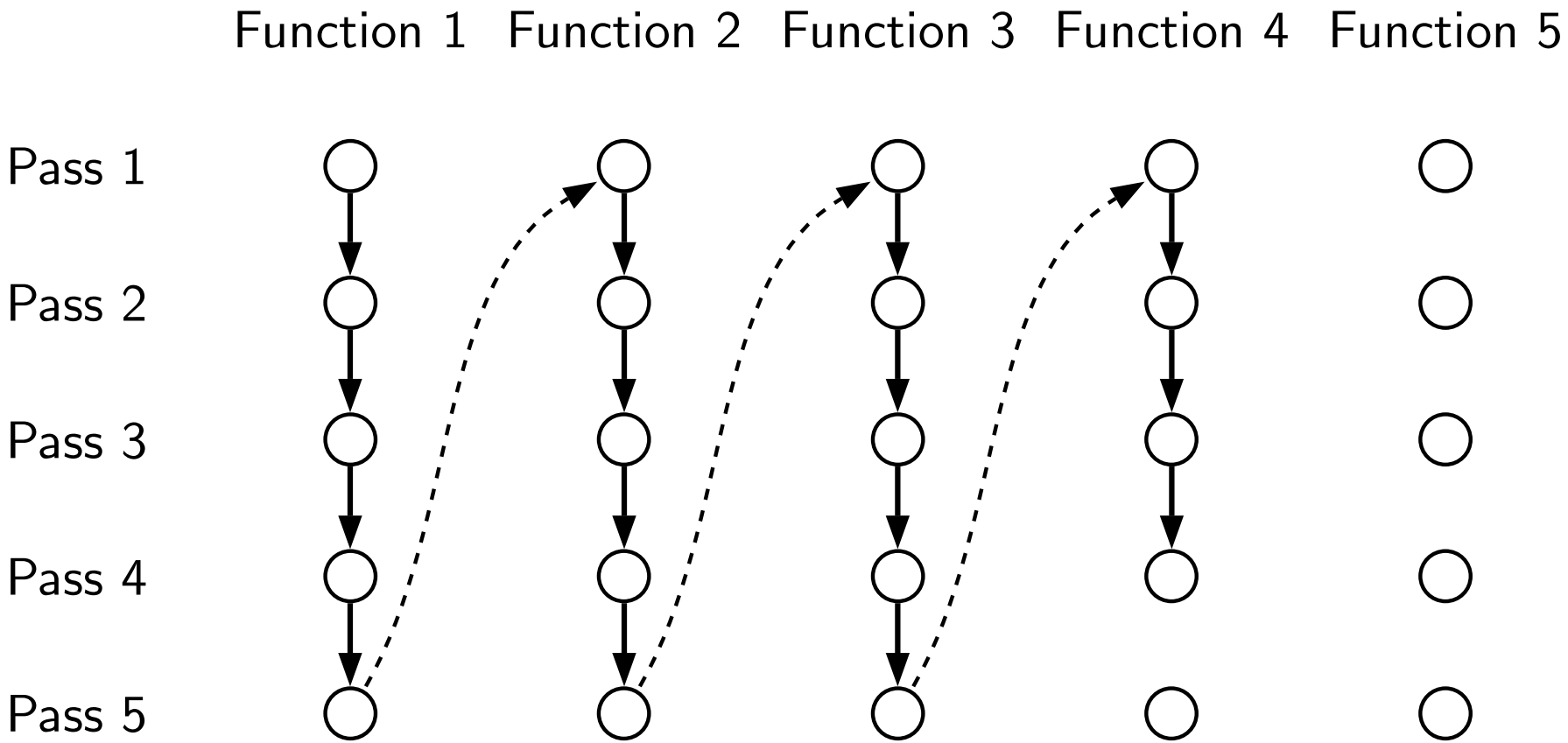
Execution Order in Intraprocedural Passes



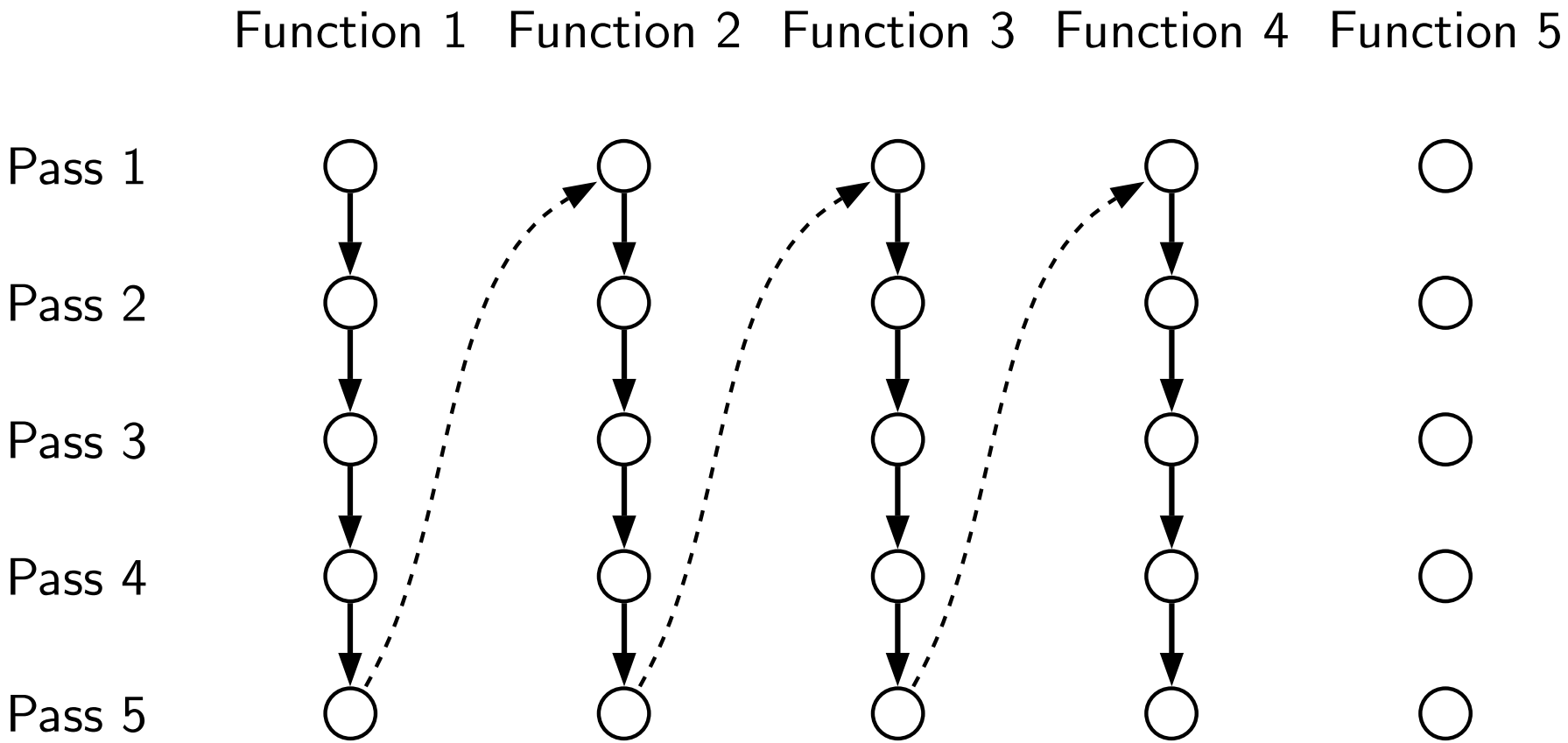
Execution Order in Intraprocedural Passes



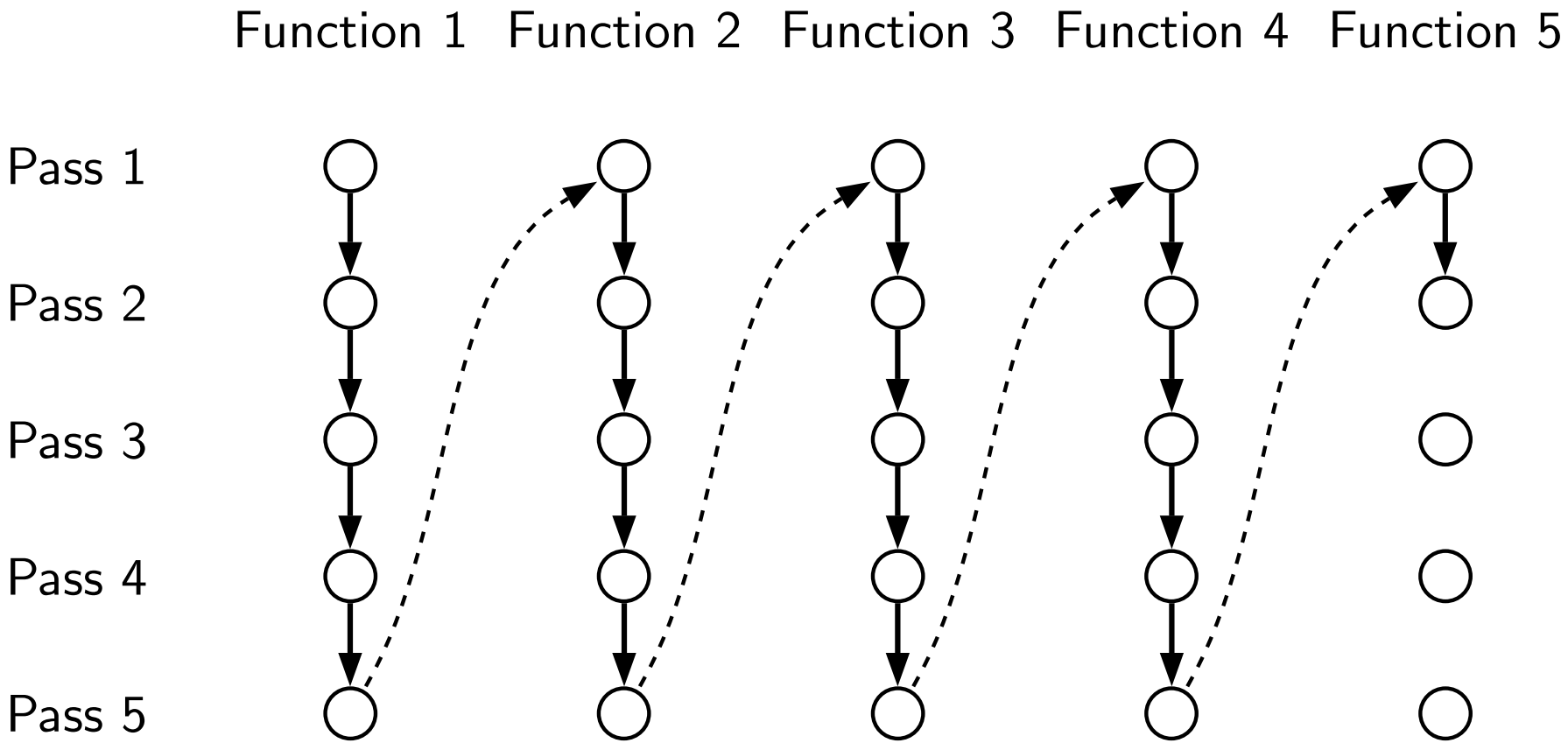
Execution Order in Intraprocedural Passes



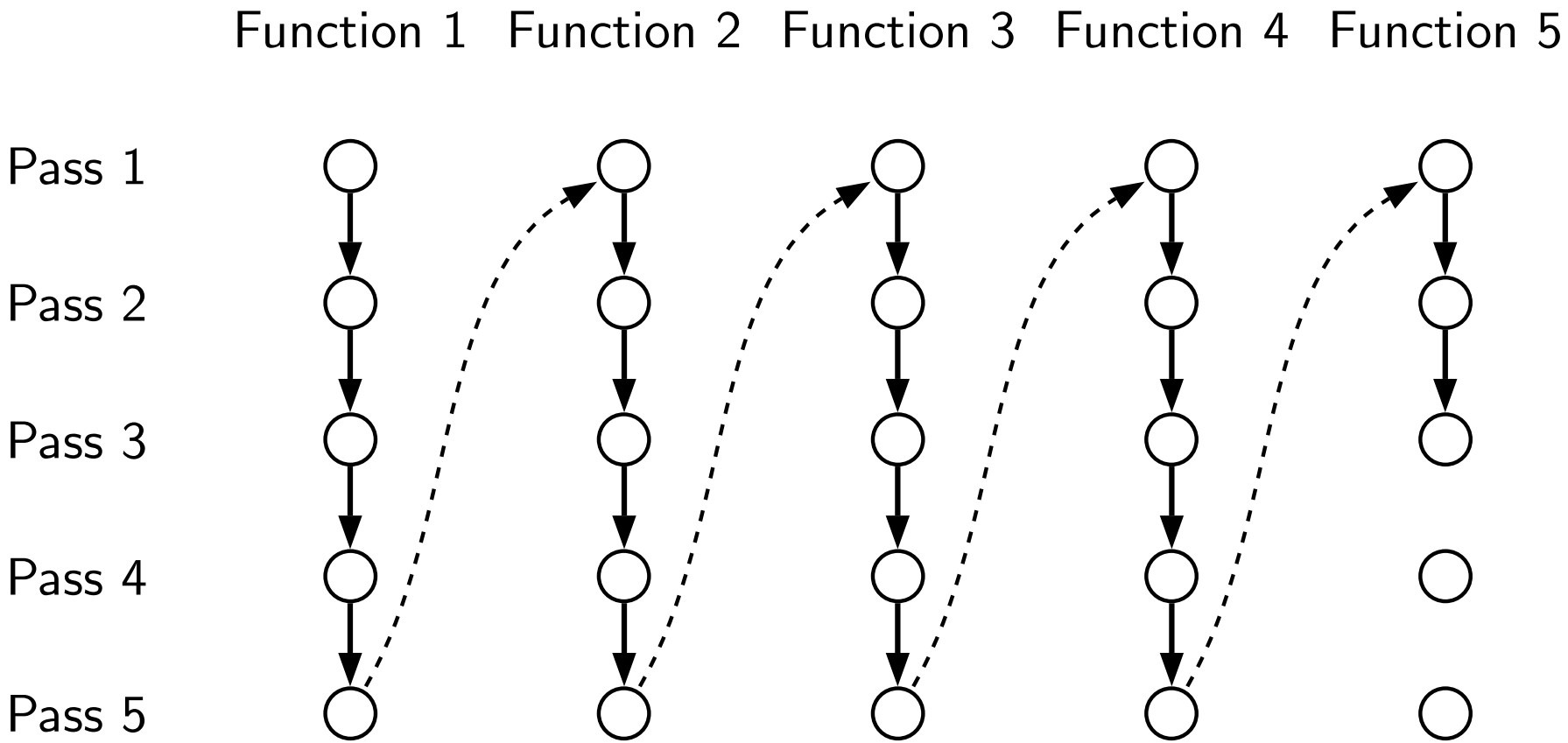
Execution Order in Intraprocedural Passes



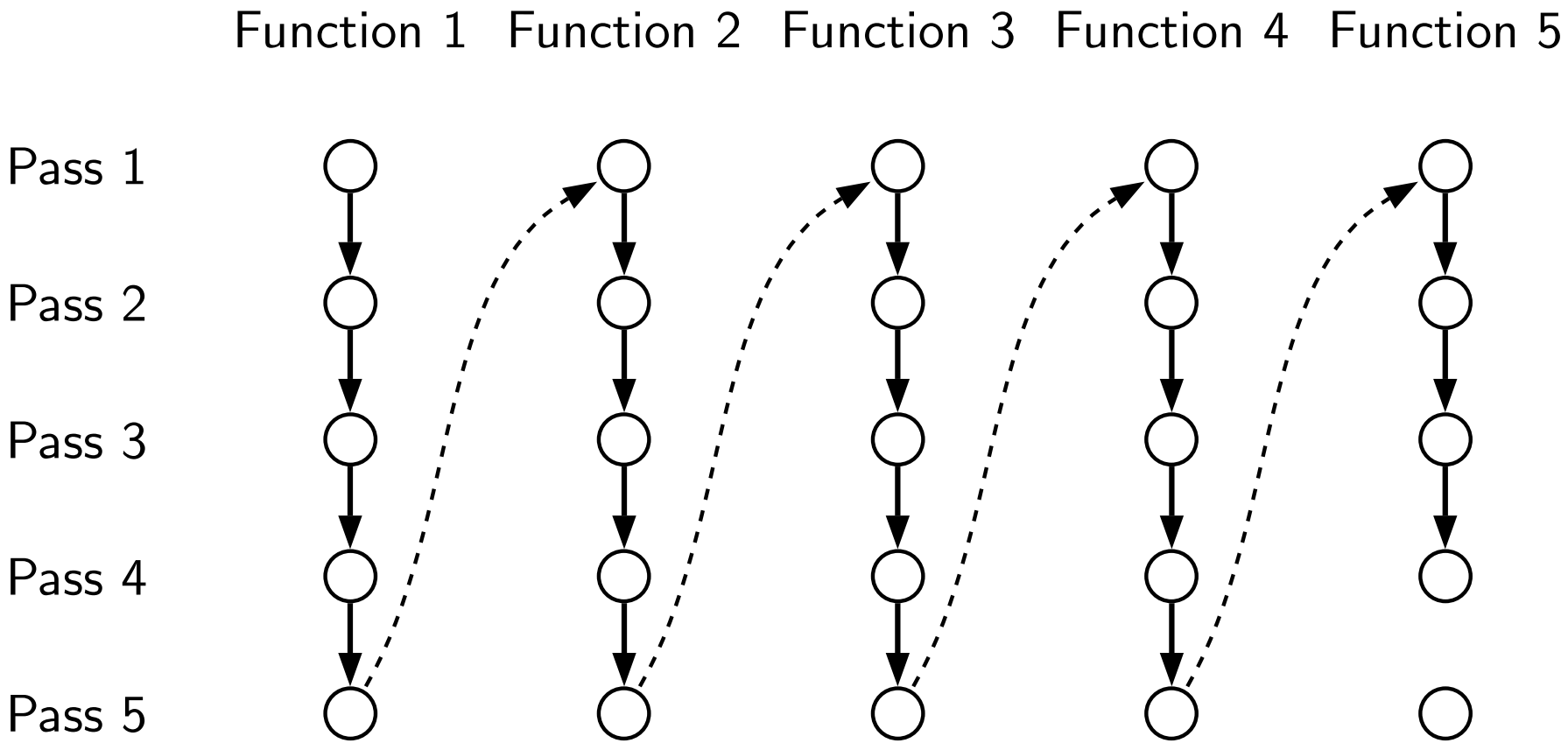
Execution Order in Intraprocedural Passes



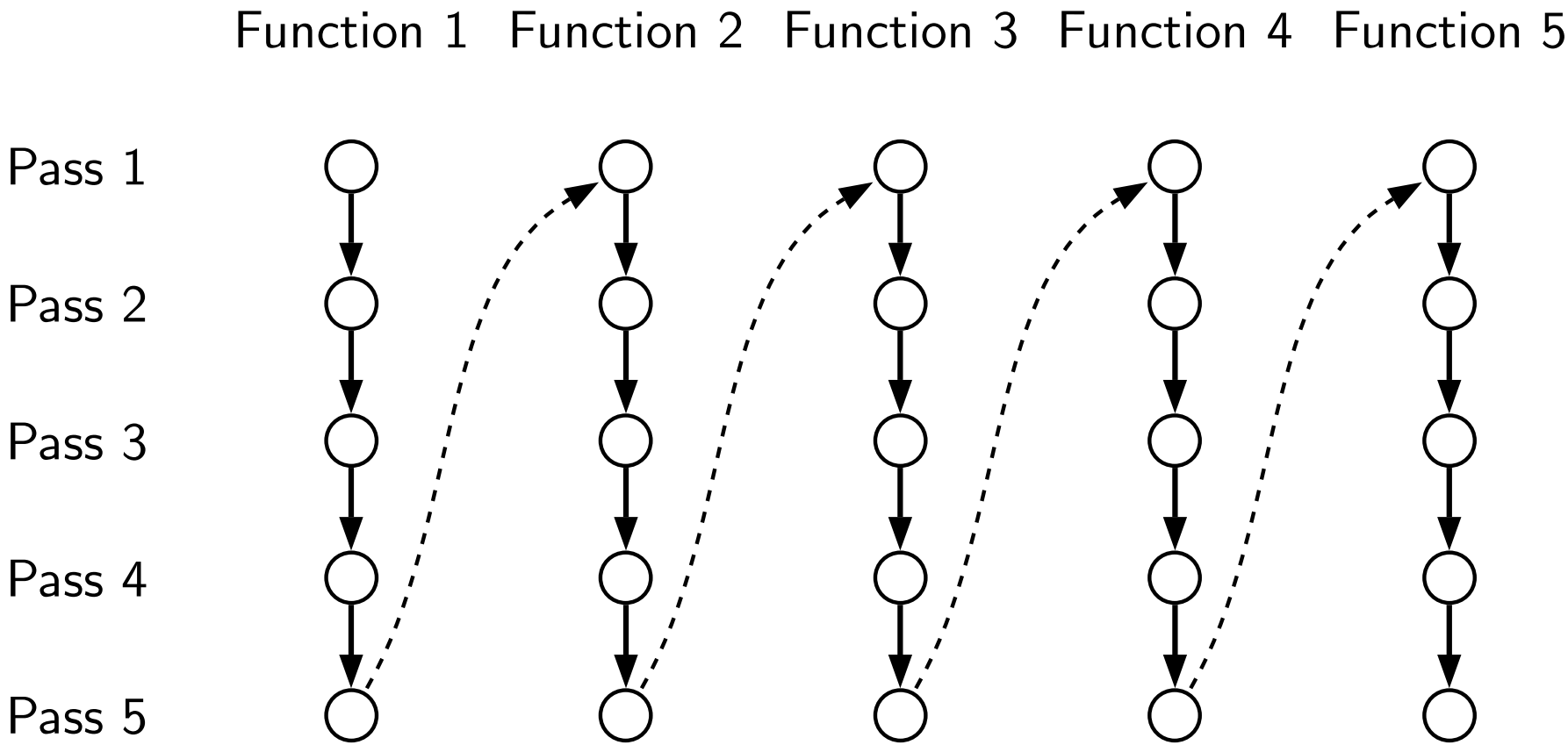
Execution Order in Intraprocedural Passes



Execution Order in Intraprocedural Passes



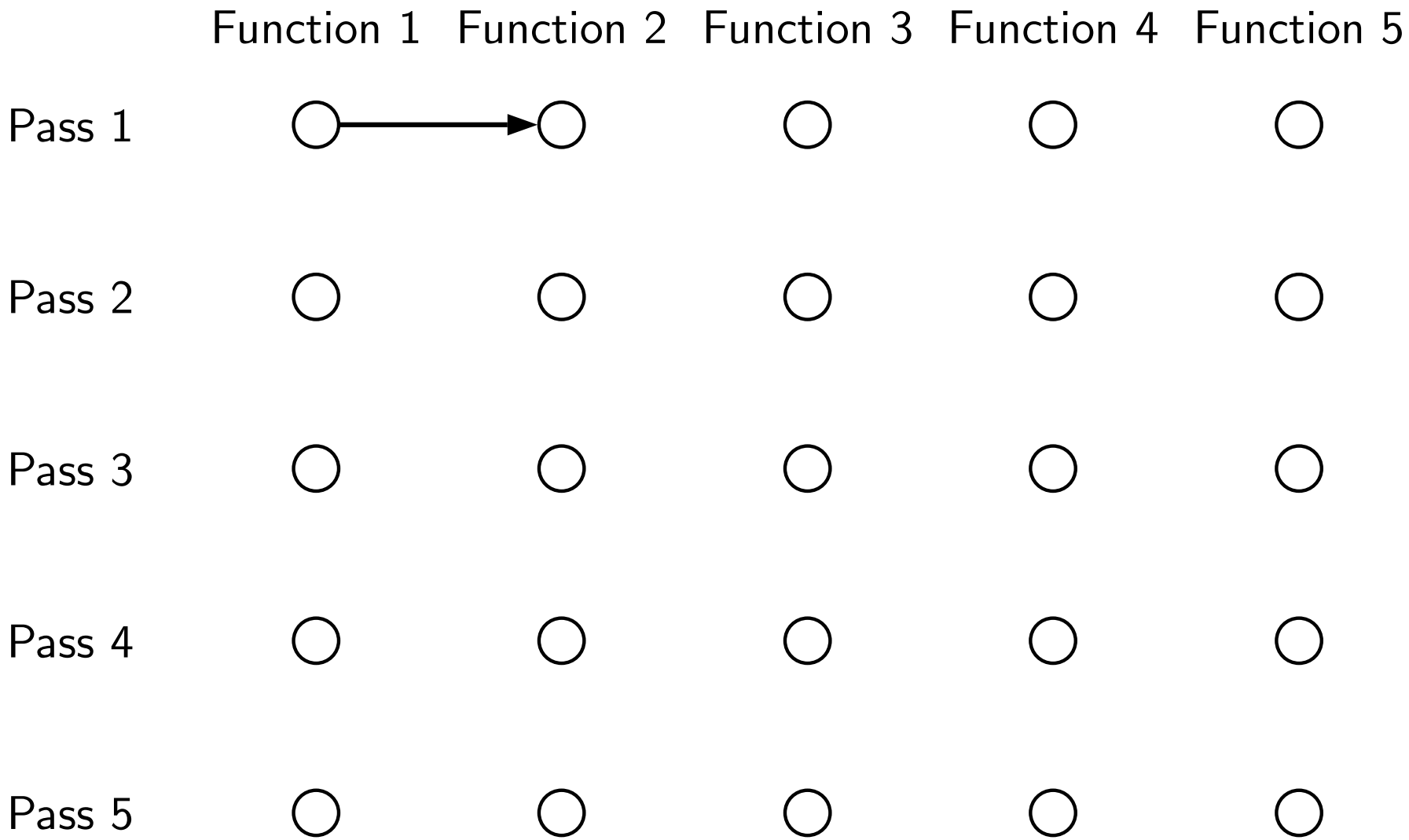
Execution Order in Intraprocedural Passes



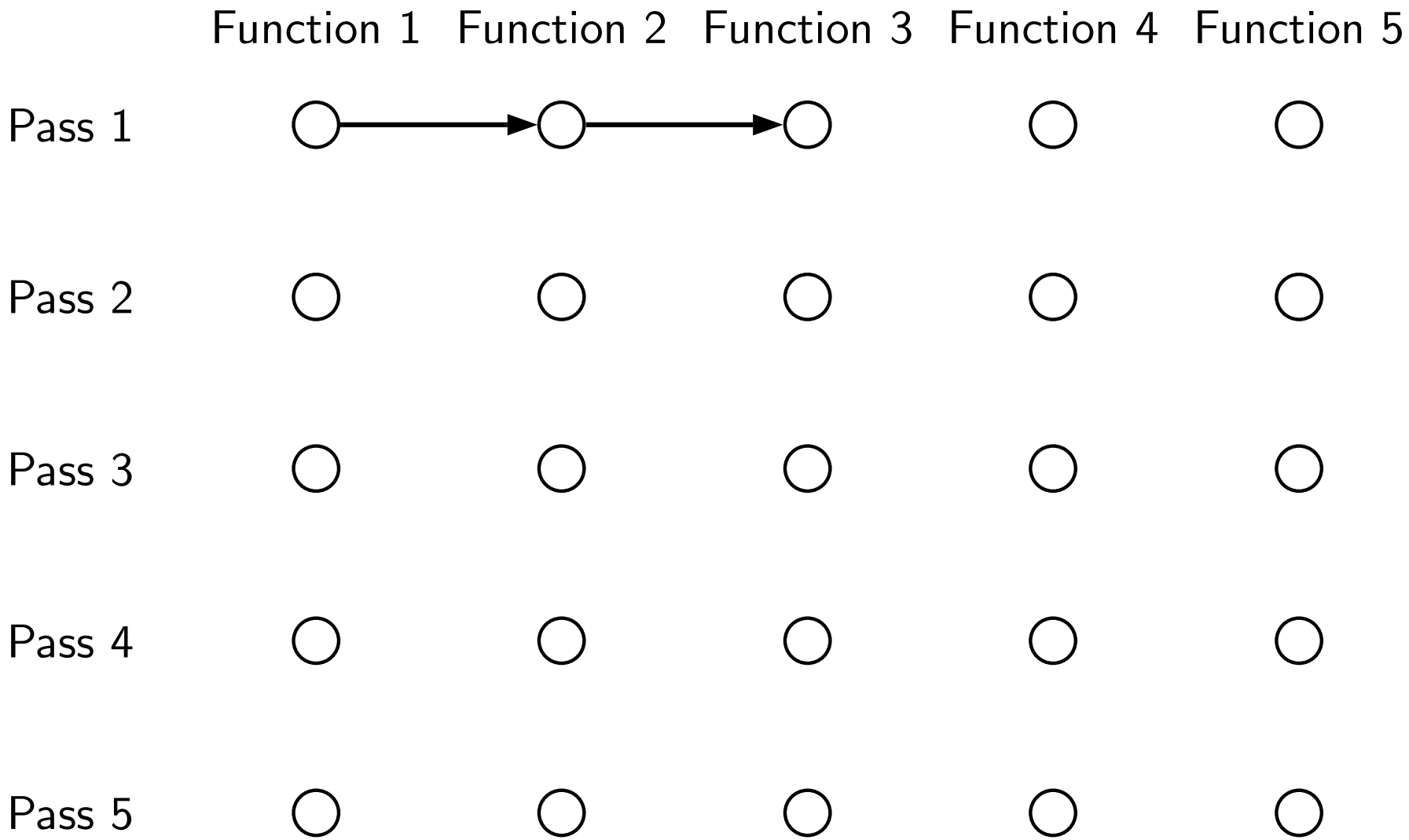
Execution Order in Interprocedural Passes

	Function 1	Function 2	Function 3	Function 4	Function 5
Pass 1	○	○	○	○	○
Pass 2	○	○	○	○	○
Pass 3	○	○	○	○	○
Pass 4	○	○	○	○	○
Pass 5	○	○	○	○	○

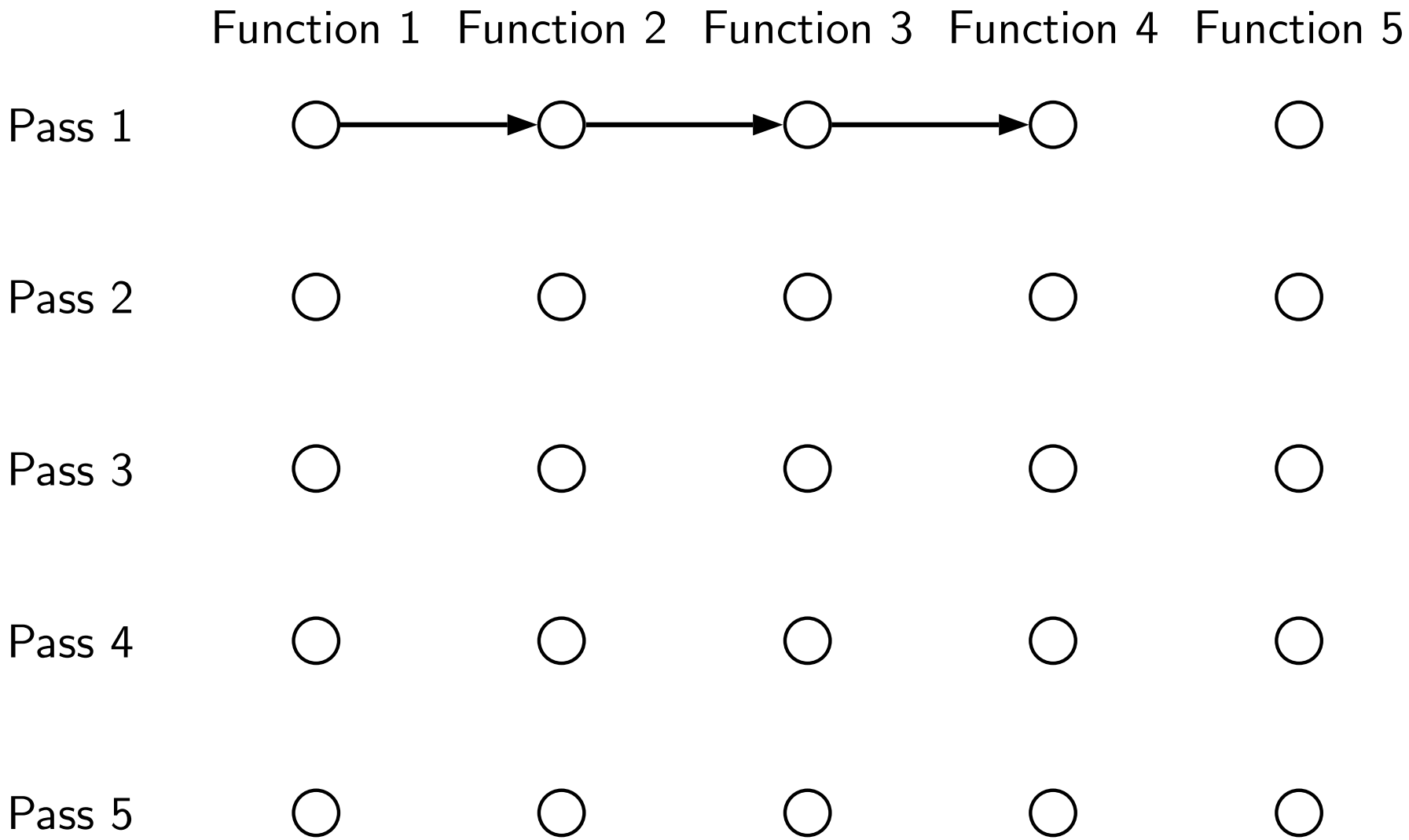
Execution Order in Interprocedural Passes



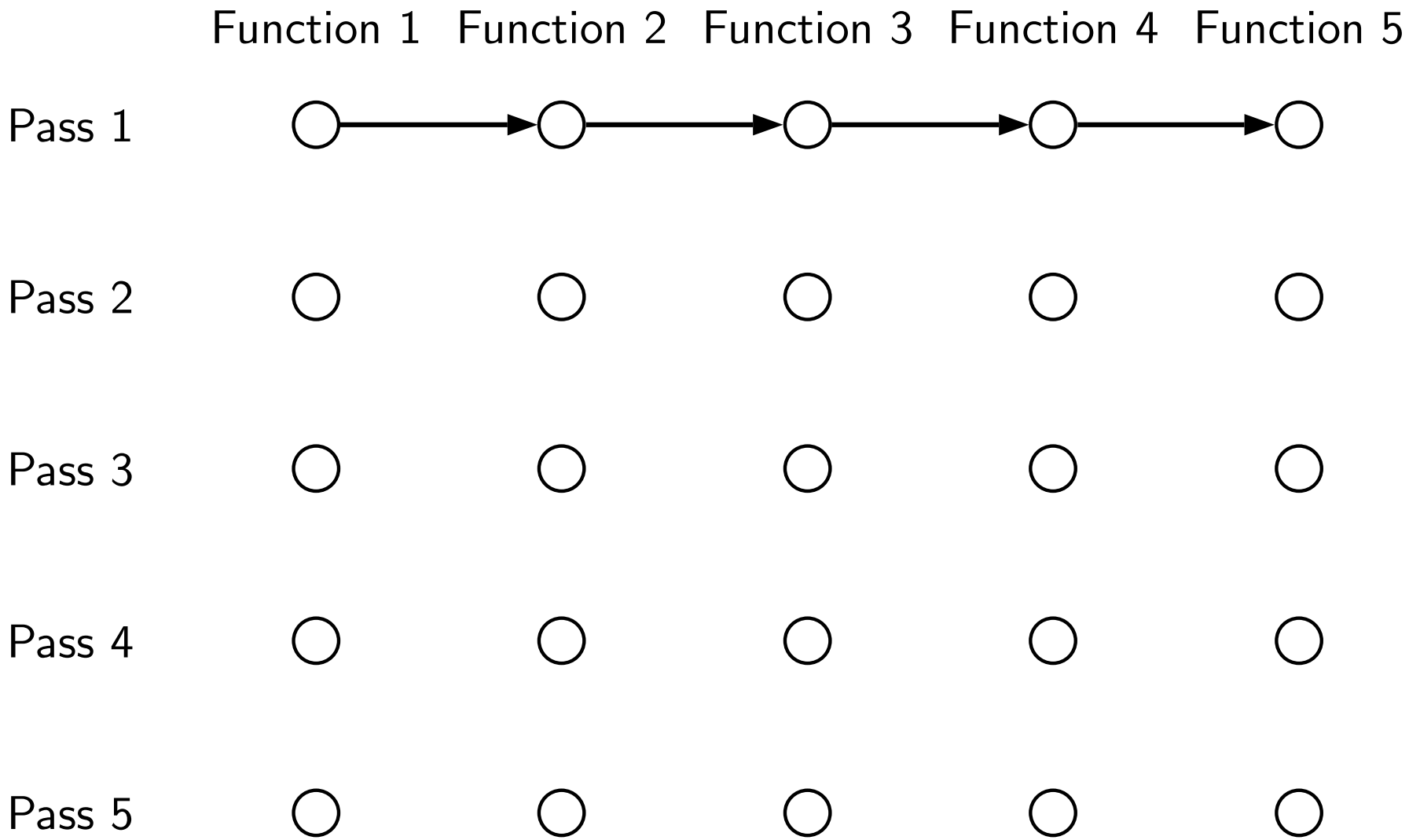
Execution Order in Interprocedural Passes



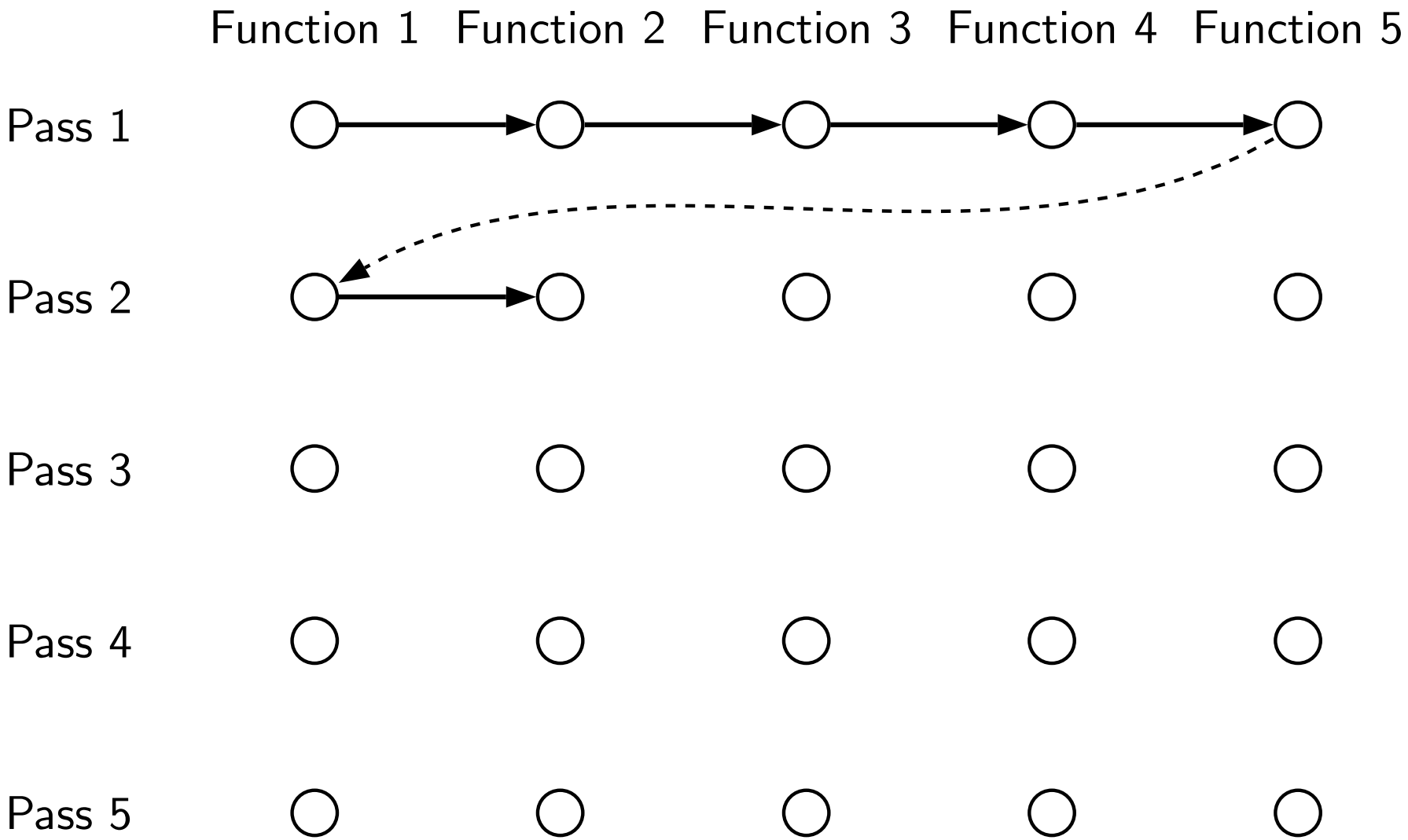
Execution Order in Interprocedural Passes



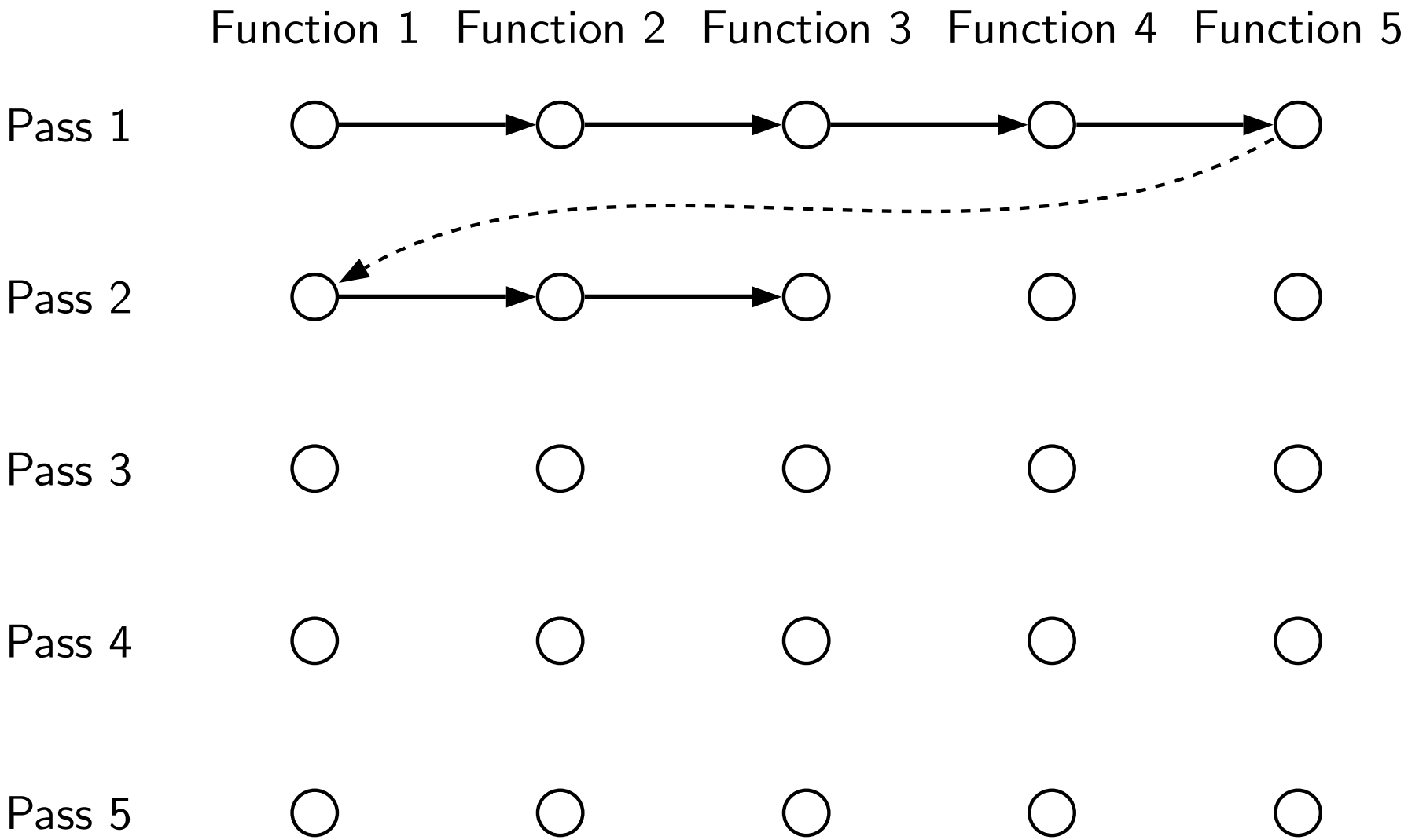
Execution Order in Interprocedural Passes



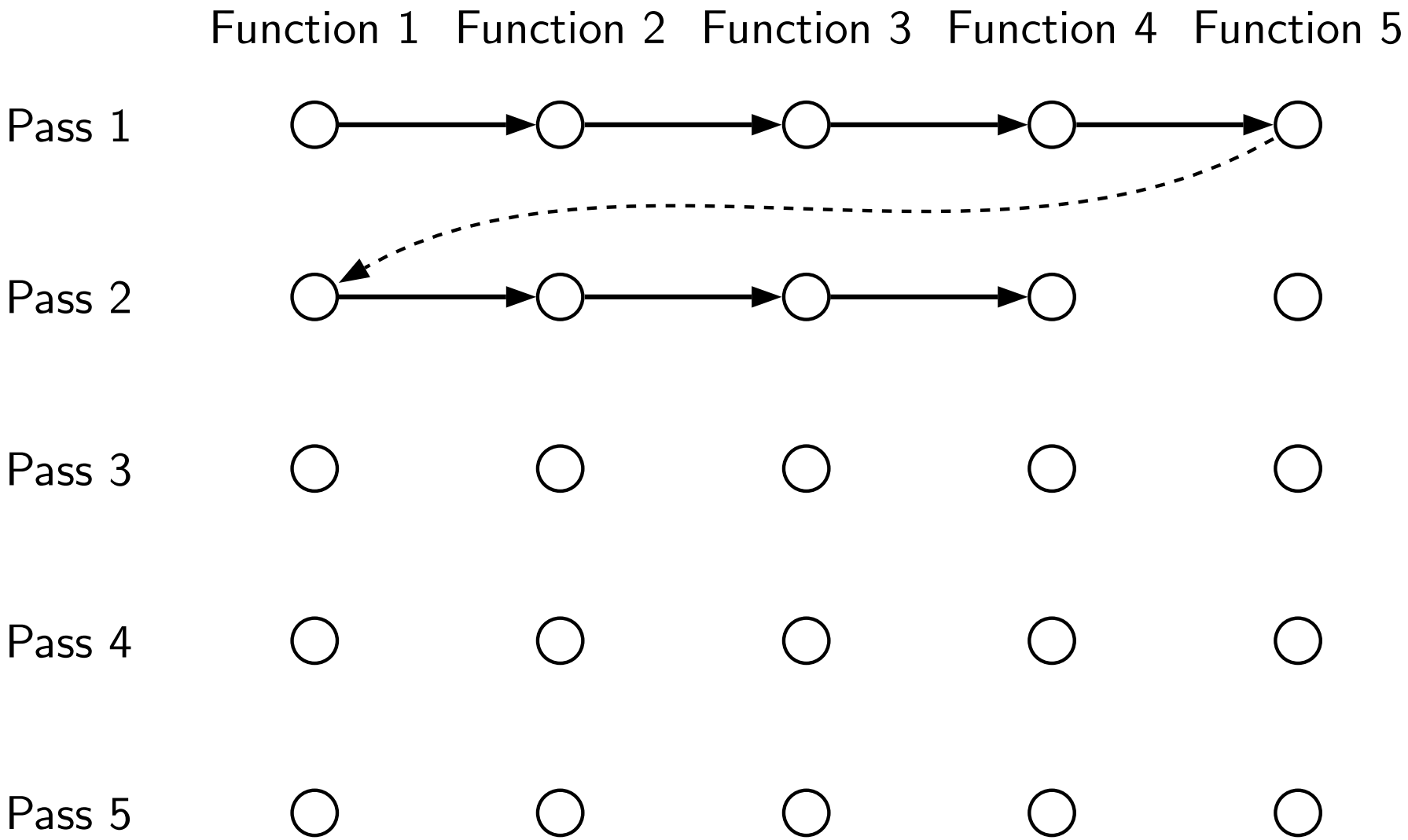
Execution Order in Interprocedural Passes



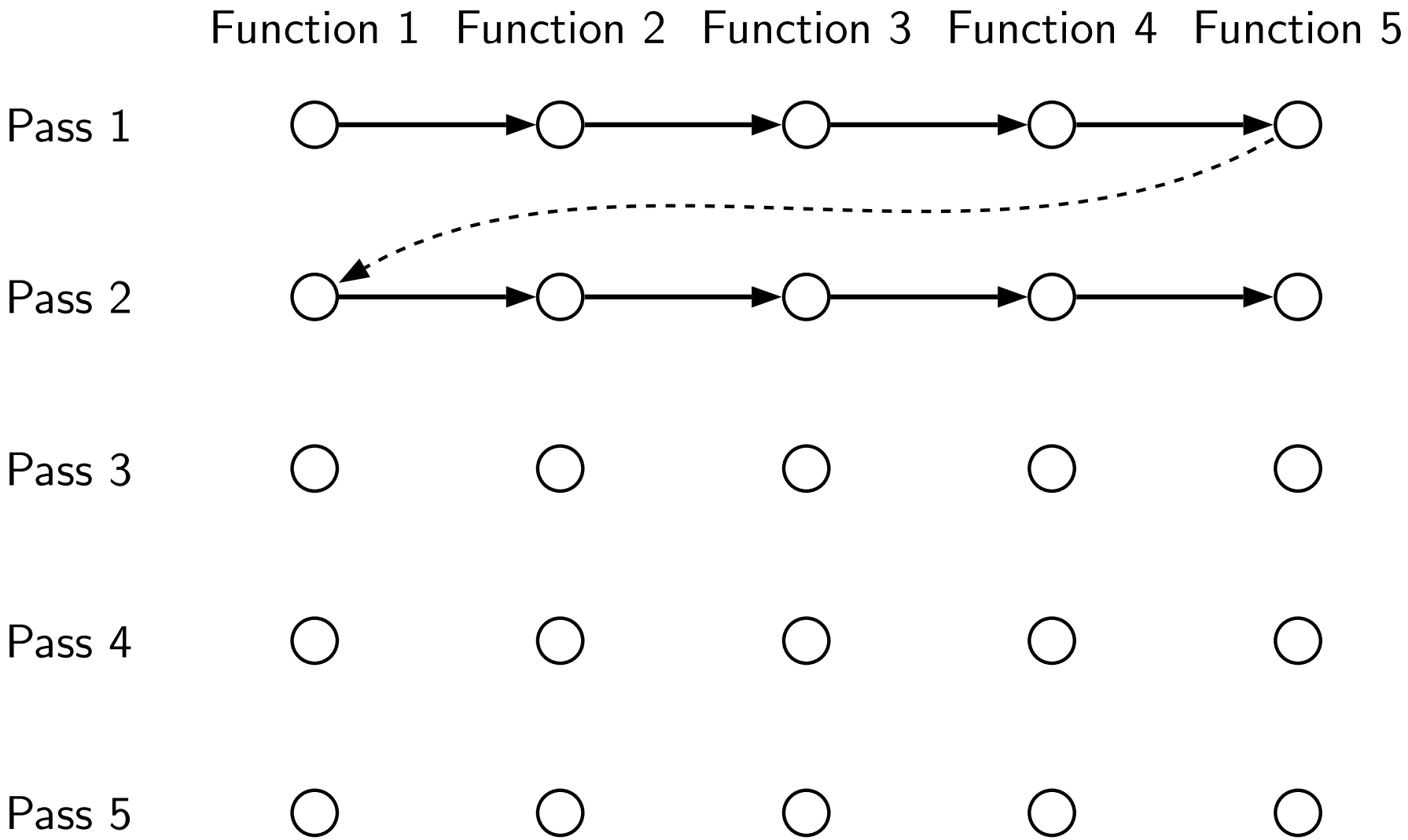
Execution Order in Interprocedural Passes



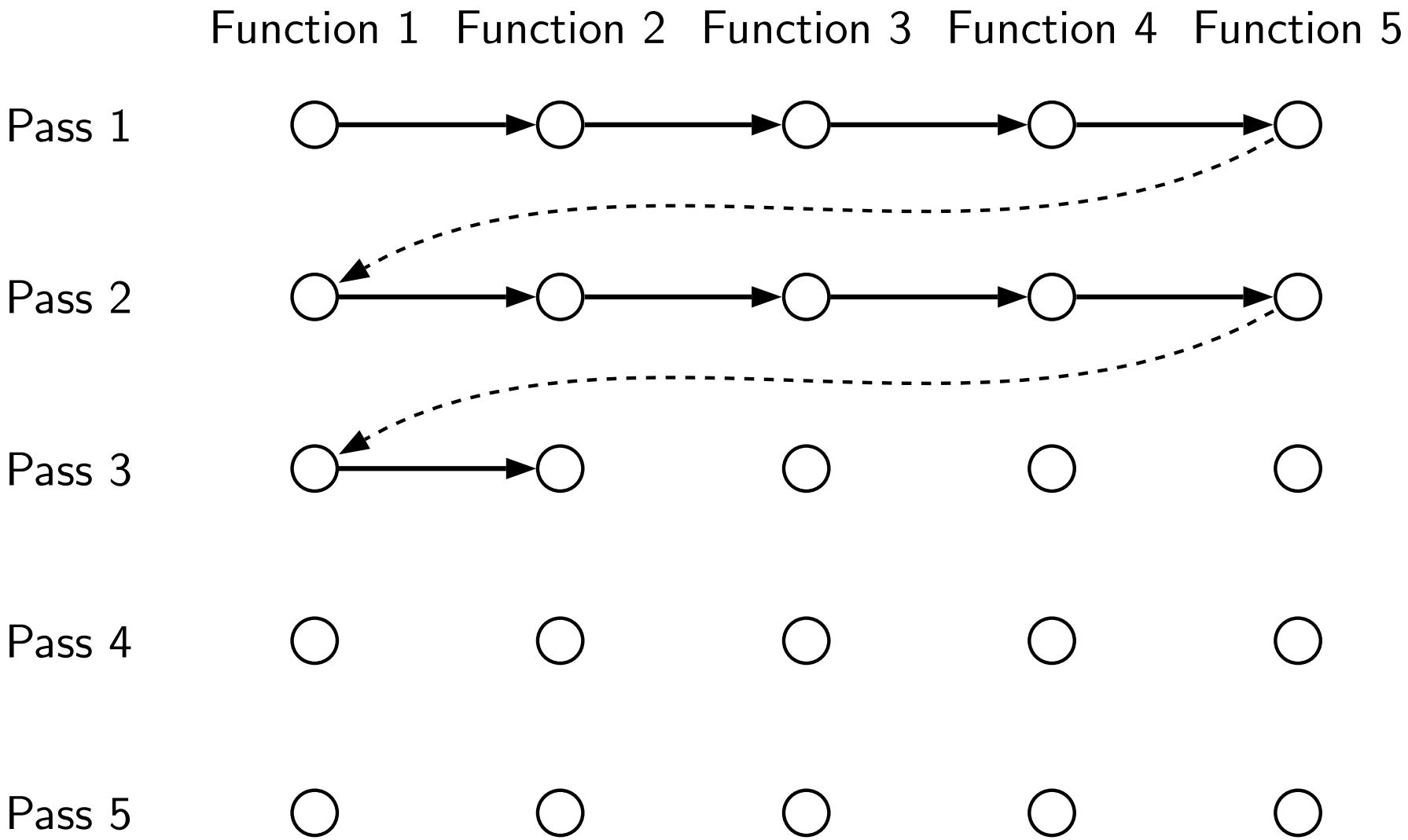
Execution Order in Interprocedural Passes



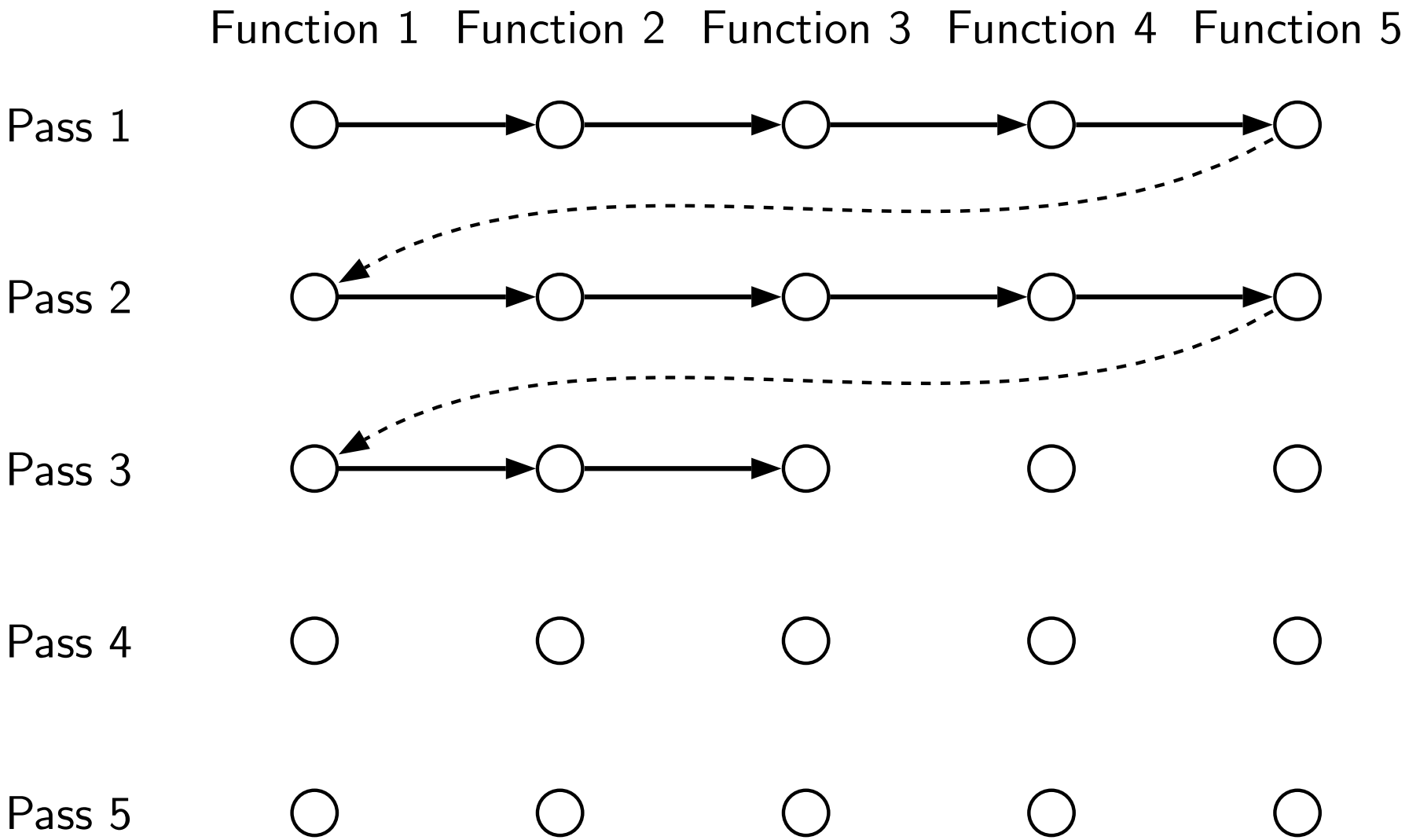
Execution Order in Interprocedural Passes



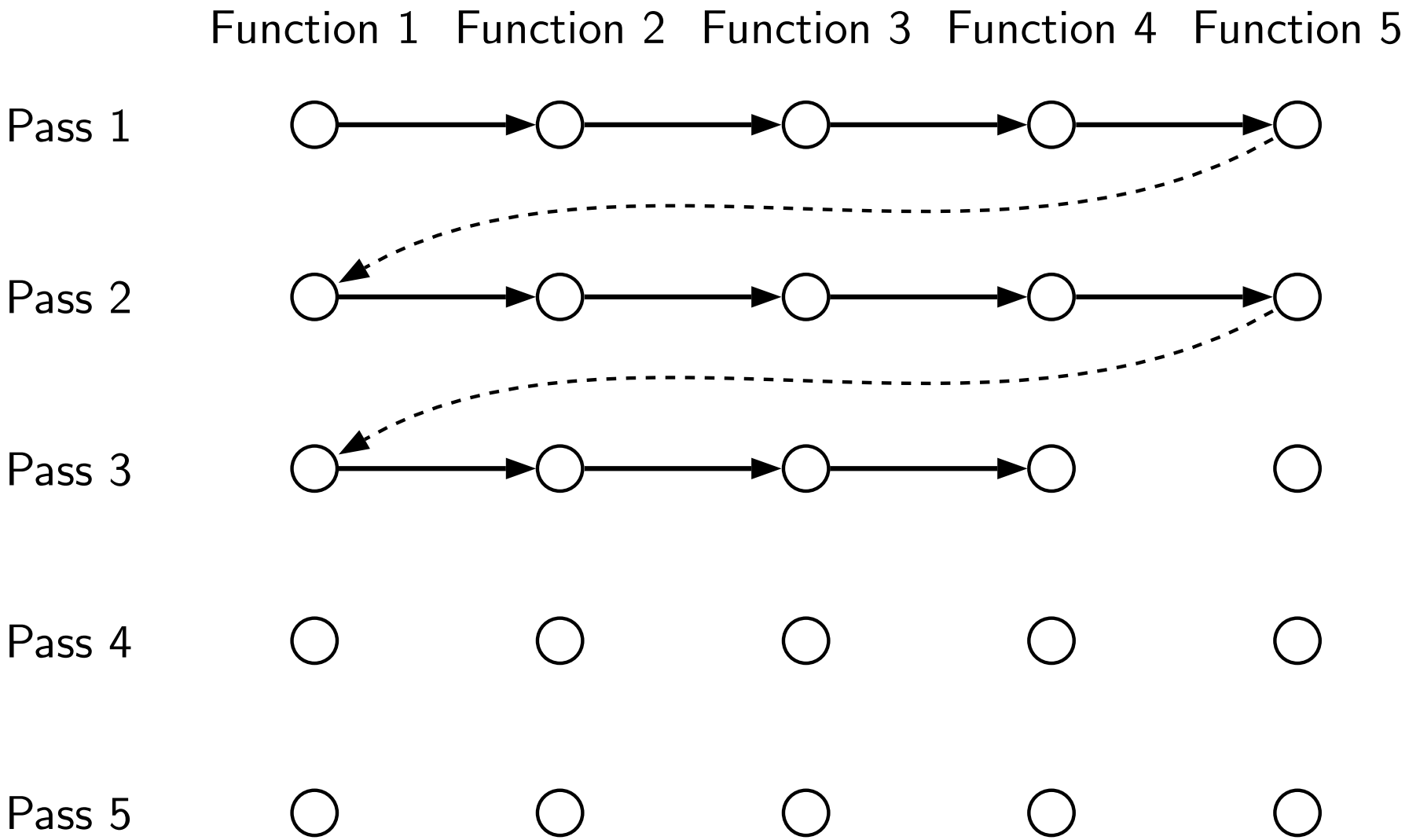
Execution Order in Interprocedural Passes



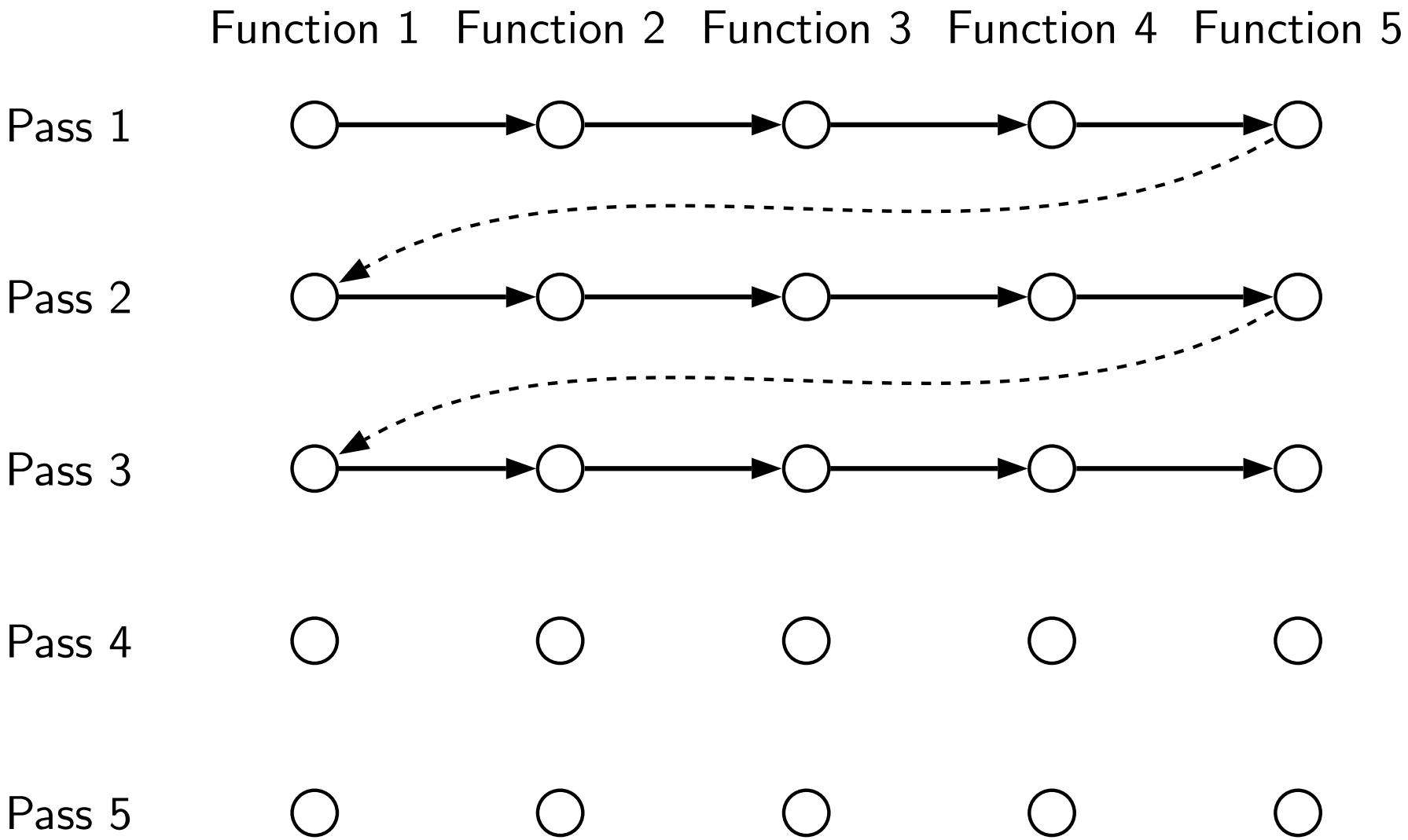
Execution Order in Interprocedural Passes



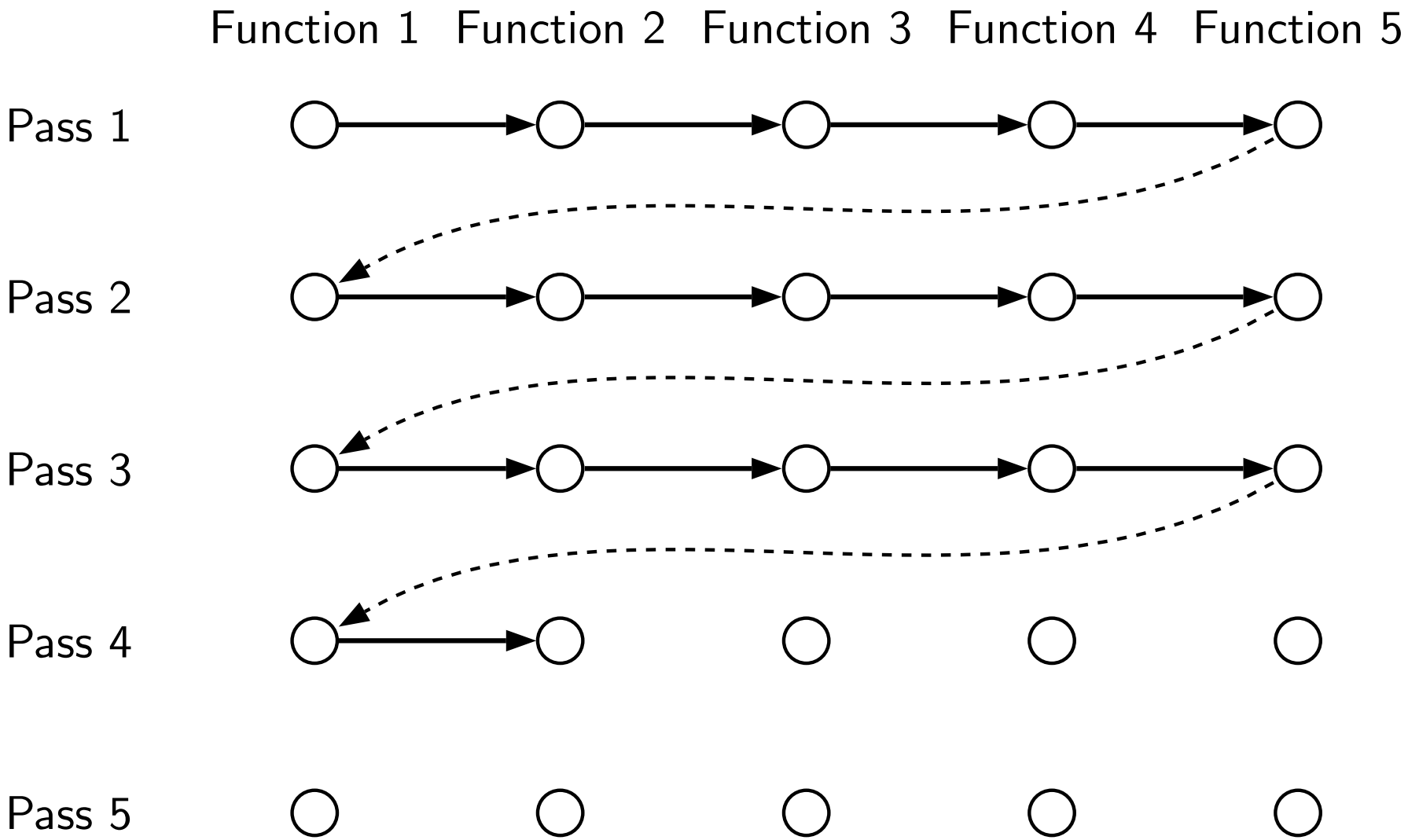
Execution Order in Interprocedural Passes



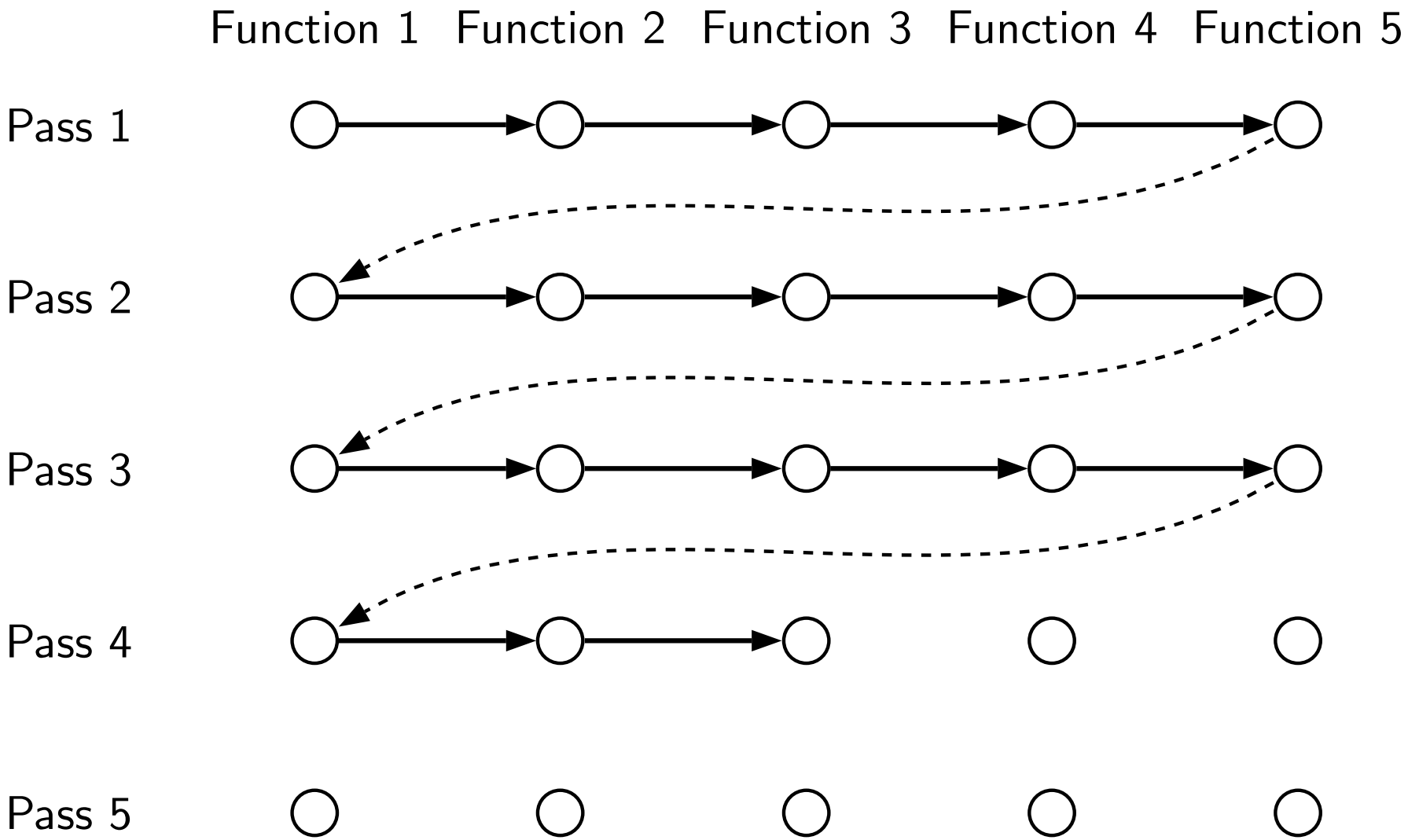
Execution Order in Interprocedural Passes



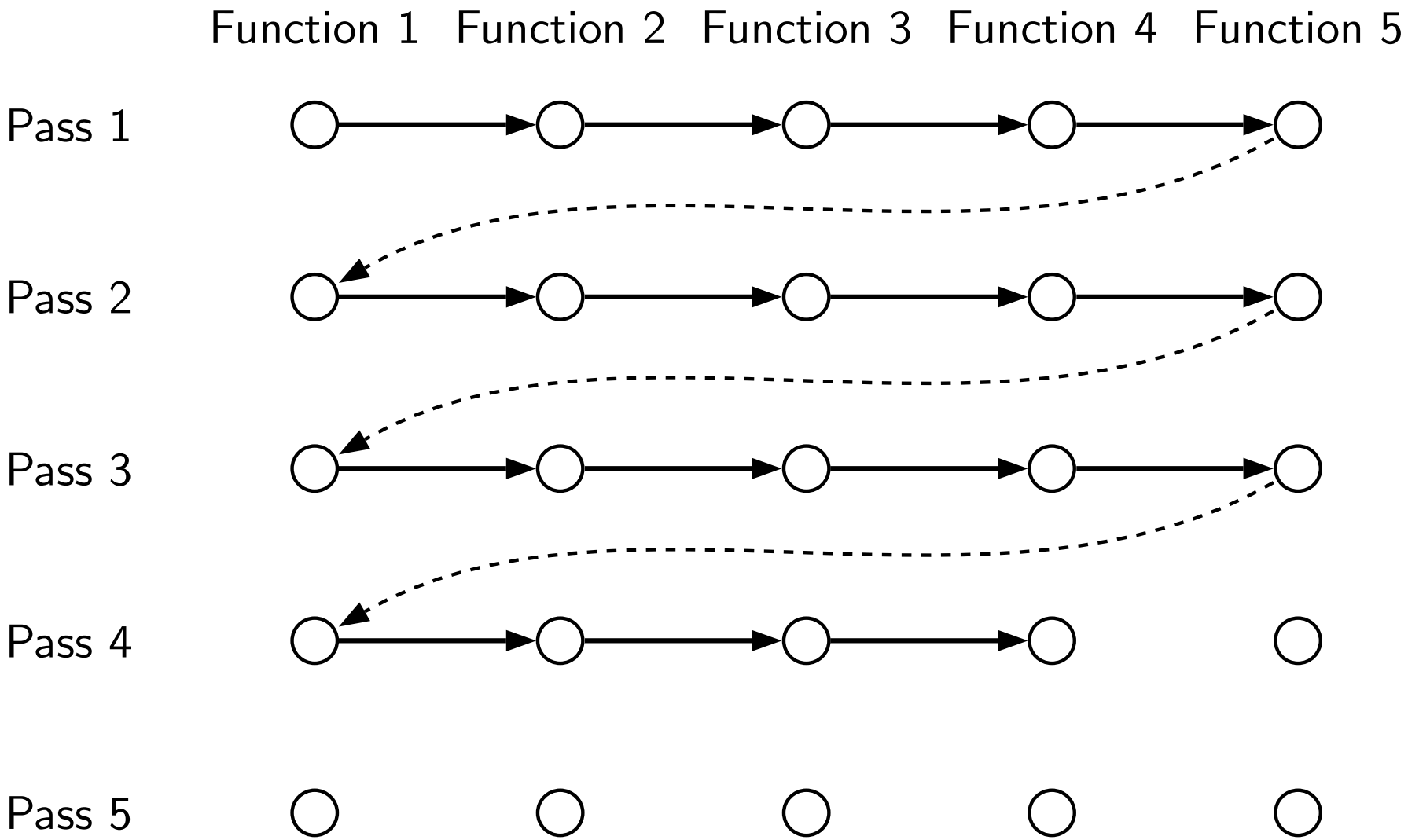
Execution Order in Interprocedural Passes



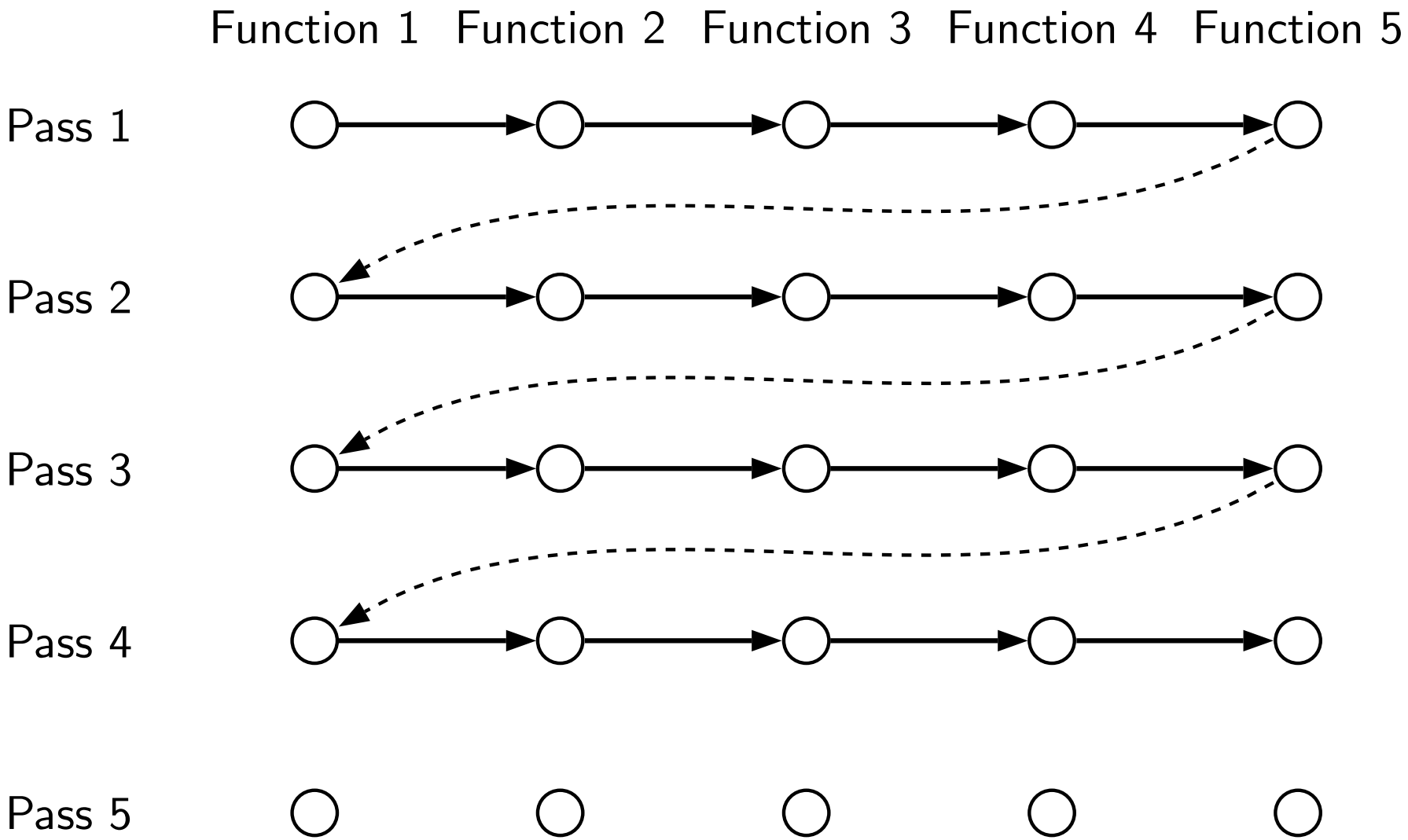
Execution Order in Interprocedural Passes



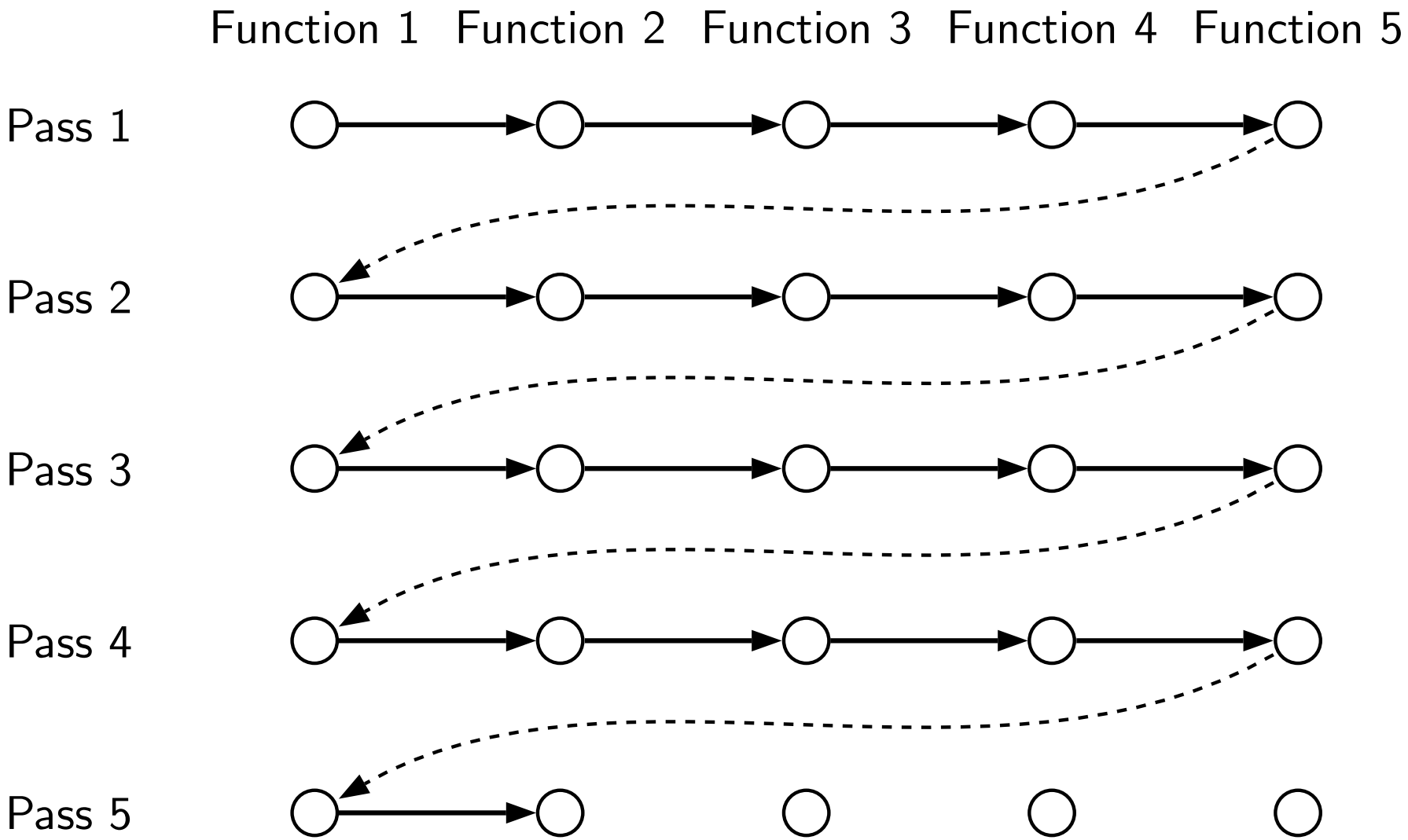
Execution Order in Interprocedural Passes



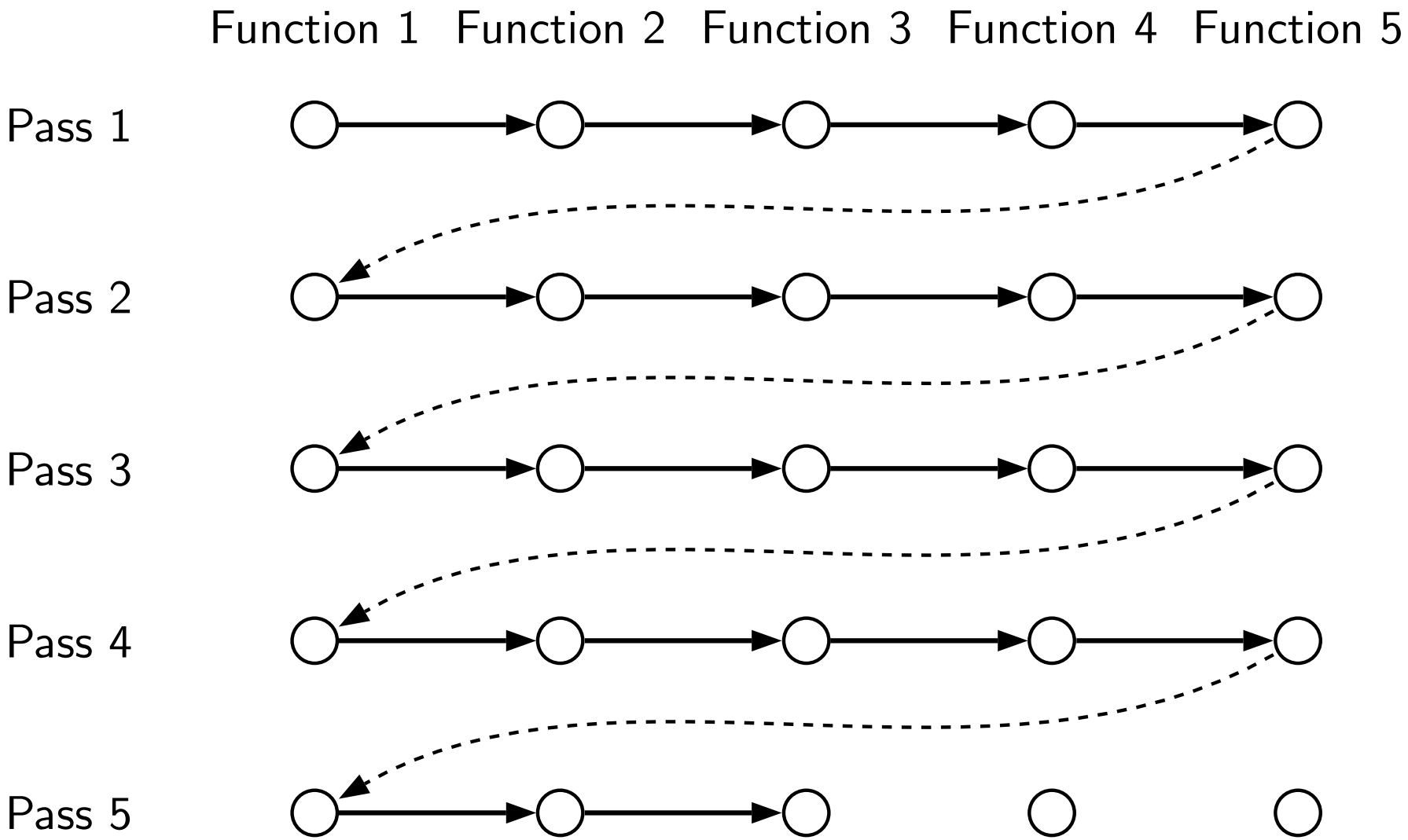
Execution Order in Interprocedural Passes



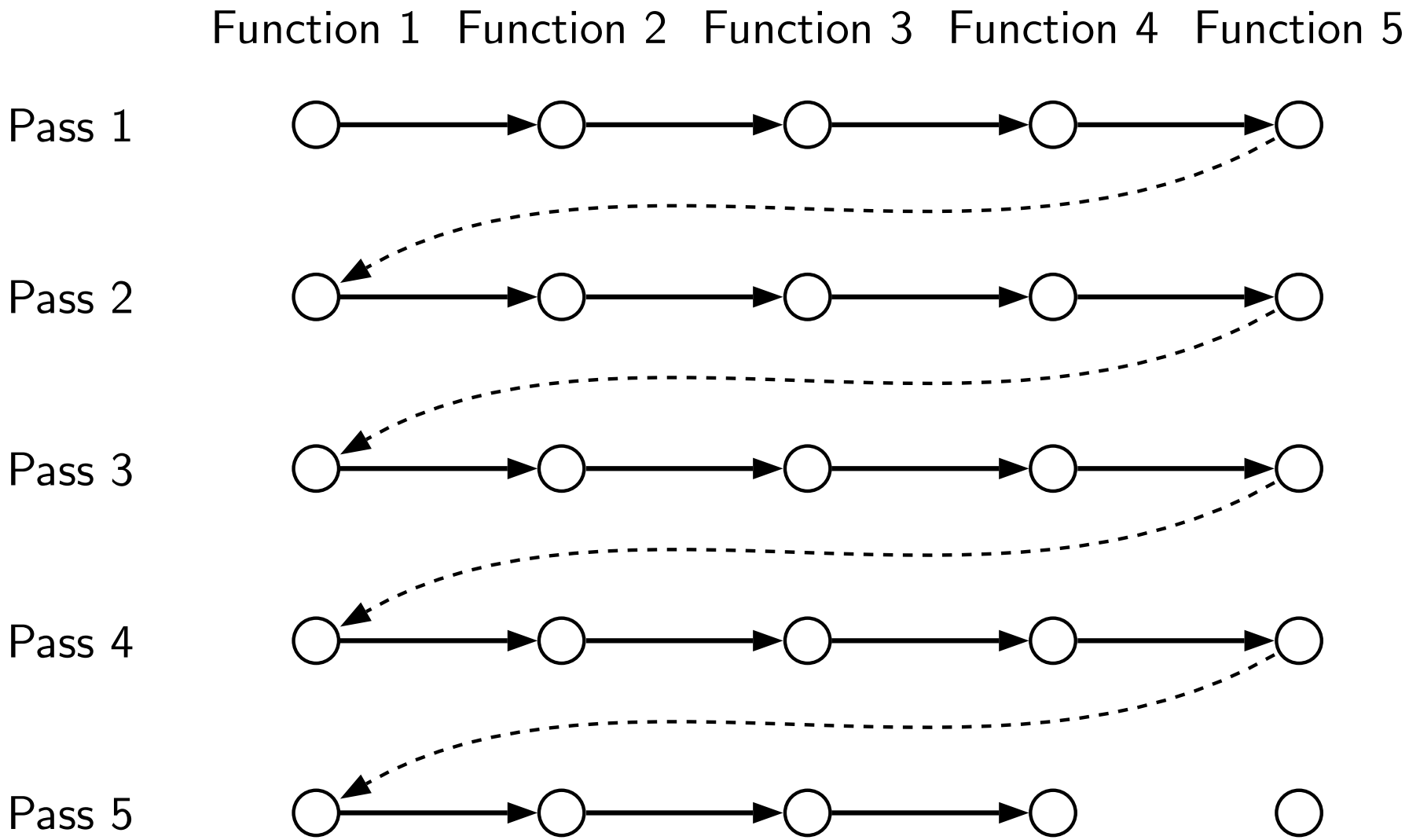
Execution Order in Interprocedural Passes



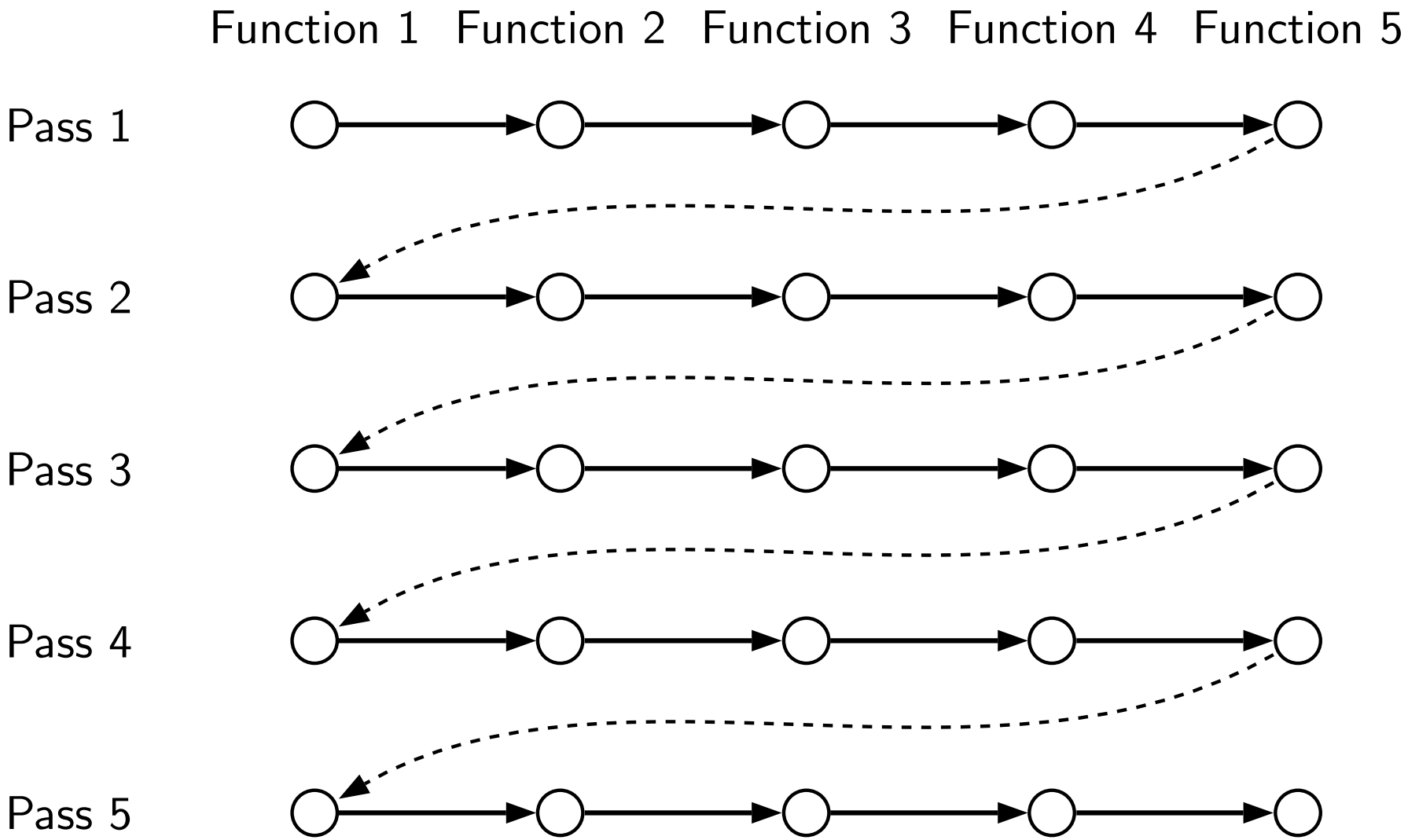
Execution Order in Interprocedural Passes



Execution Order in Interprocedural Passes



Execution Order in Interprocedural Passes



cc1 Control Flow: GIMPLE to RTL Expansion (pass_expand)

```
gimple_expand_cfg
  expand_gimple_basic_block(bb)
     expand_gimple_cond(stmt)
     expand_gimple_stmt(stmt)
         expand_gimple_stmt_1 (stmt)
             expand_expr_real_2
                expand_expr   /* Operands */
                    expand_expr_real
                optab_for_tree_code
                expand_binop /* Now we have rtx for operands */
                   expand_binop_directly
                     /* The plugin for a machine */
                     code=optab_handler(binoptab,mode)->insn_code;
                     GEN_FCN
                     emit_insn
```


What About LTO?

- LTO framework supported in GCC-4.6.0
  Released this week :-)
- Build GCC using `--enable-lto` option during configuration
- Use `-flto` option during compilation
- Generates conventional `.o` files and inserts GIMPLE level information in them
  Complete transltion is performed in this phase
- During linking all object modules are put together and `lto1` is invoked
  It re-executes optimization passes from the function `cgraph_optimize`

Basic Idea: Provide a larger call graph to regular ipa passes

Part 5

Conclusions

Conclusions

- Excellent mechanism of plugging in different
  - translators in the main driver
  - front ends, passes, and back ends in the main compiler
- However, the plugins have been used in an adhoc manner