Introduction to GCC

Uday Khedker
(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

Aug 2015

What is the GNU Compiler Collection (GCC)?

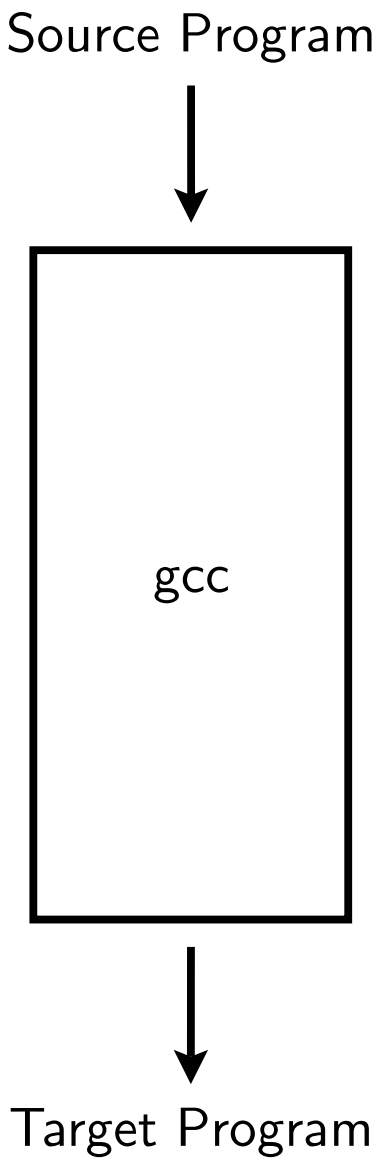
- GCC is a compiler generation framework
  - Supports many input languages
  - Supports a large number of machines
  - New machines can be included by describing them
- GCC is one of the two most important pillars of FOSS

  (the other is the Linux kernel)
- The Linux kernel is compiled using GCC

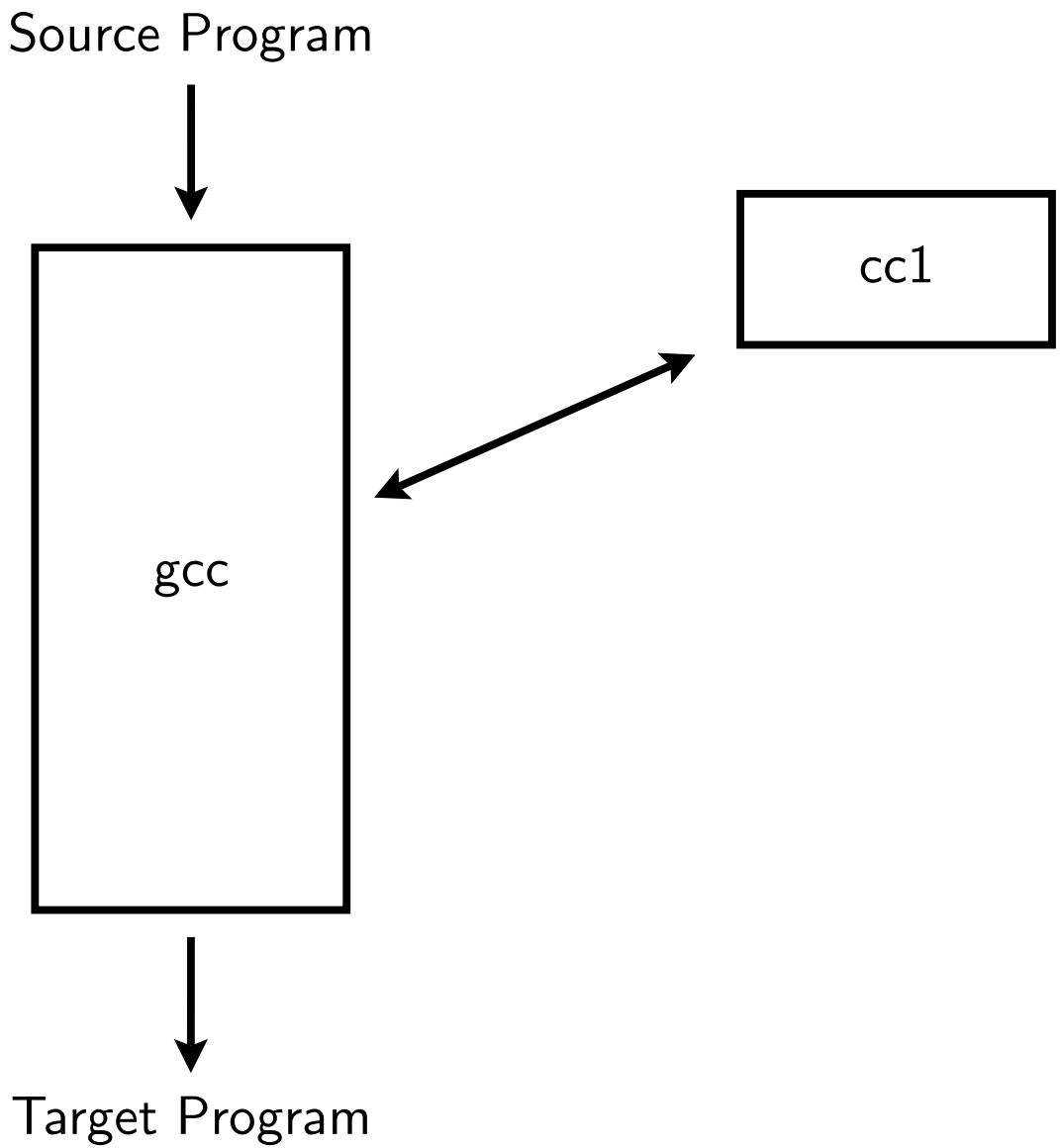
GCC in Numbers

- History: Continuous development and enhancement over last 25 years
- Number of users: Easily several millions
- Number of active developers: Several hundred
- Size: 3 MLoC; 4,000 sub-directories; 80,000 files
- Number of input languages: 7
- Number of processors: Over 60
- Number of optimizations included: close to 200

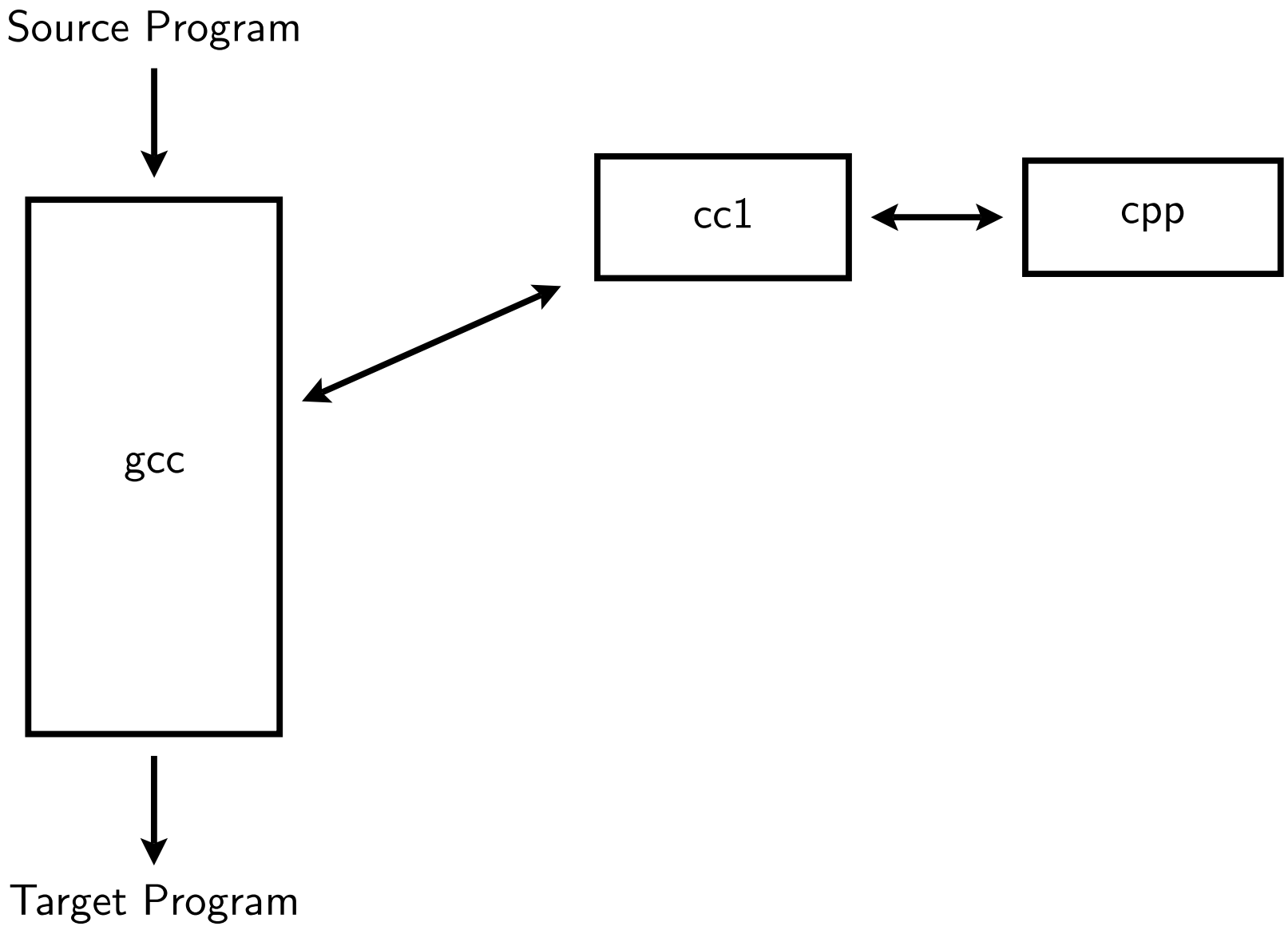
The GNU Tool Chain for C



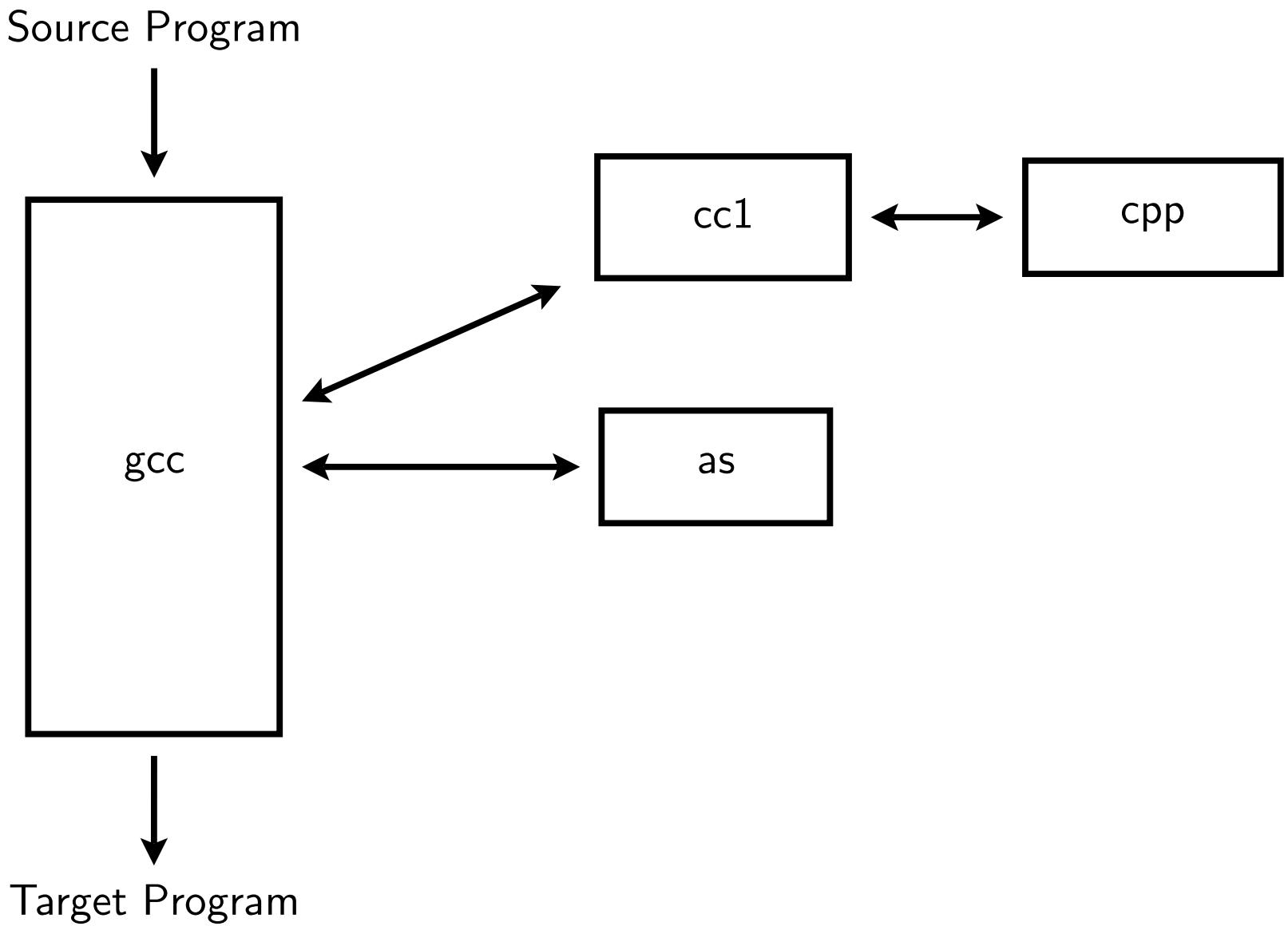
The GNU Tool Chain for C



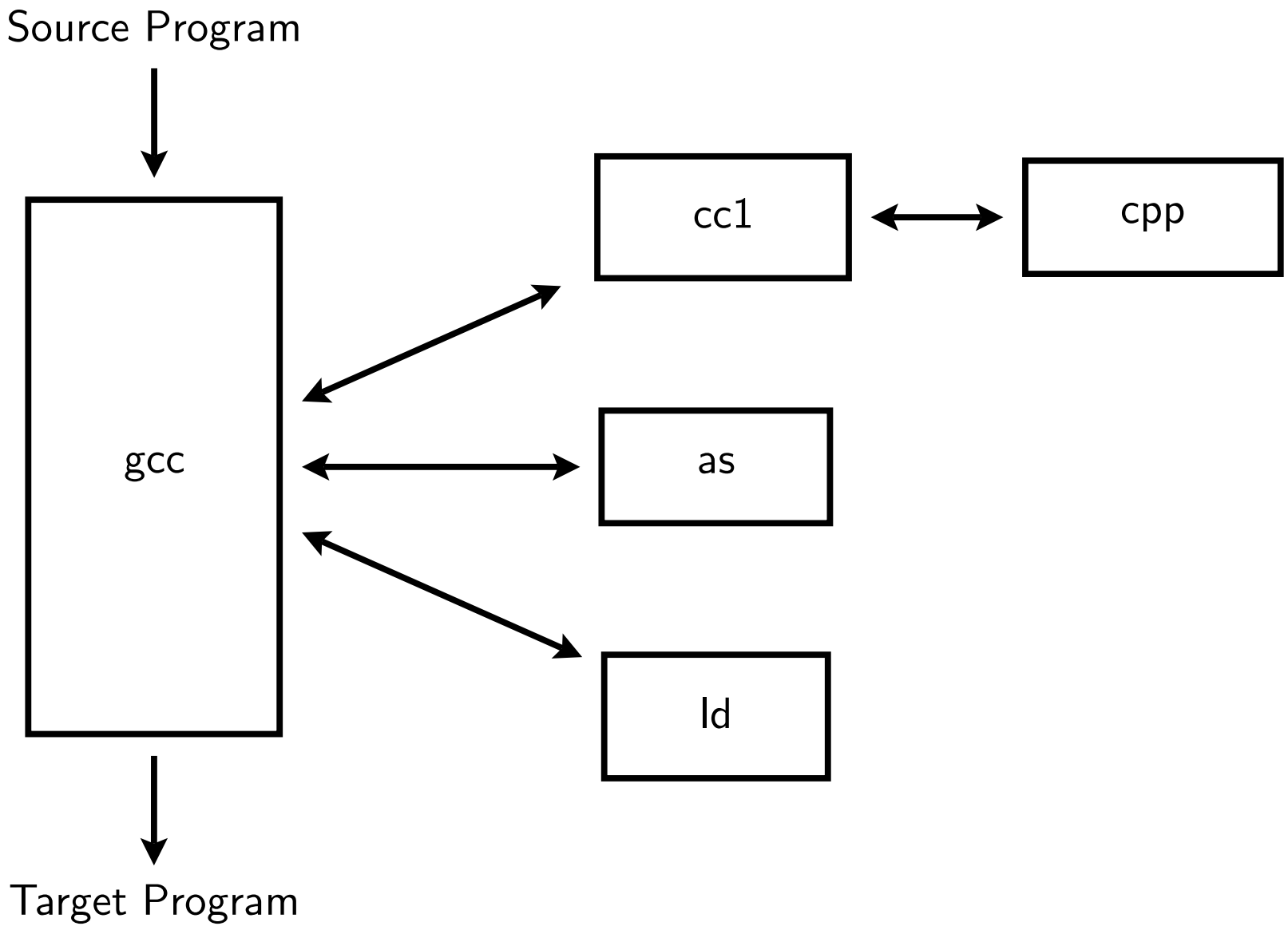
The GNU Tool Chain for C



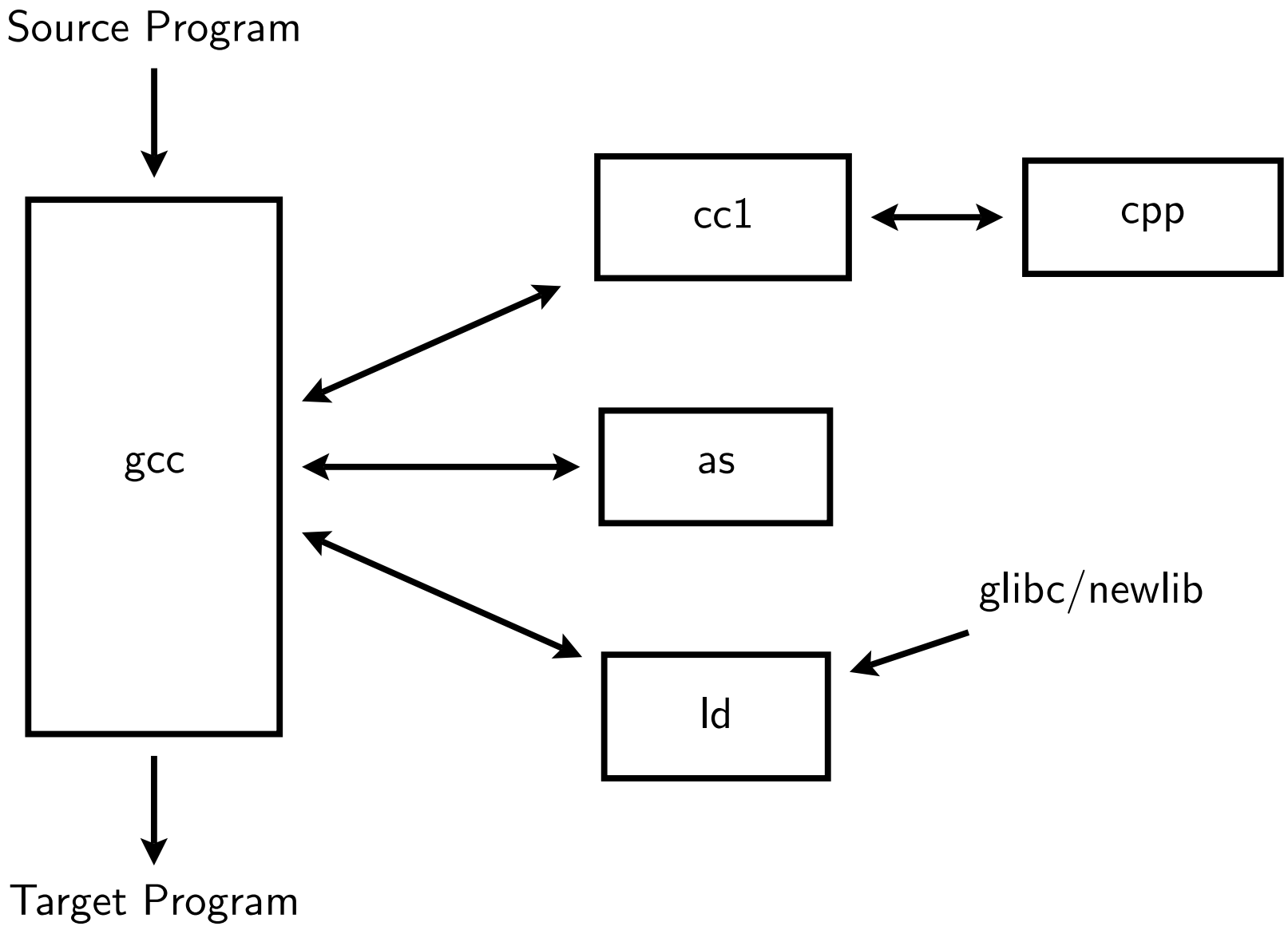
The GNU Tool Chain for C



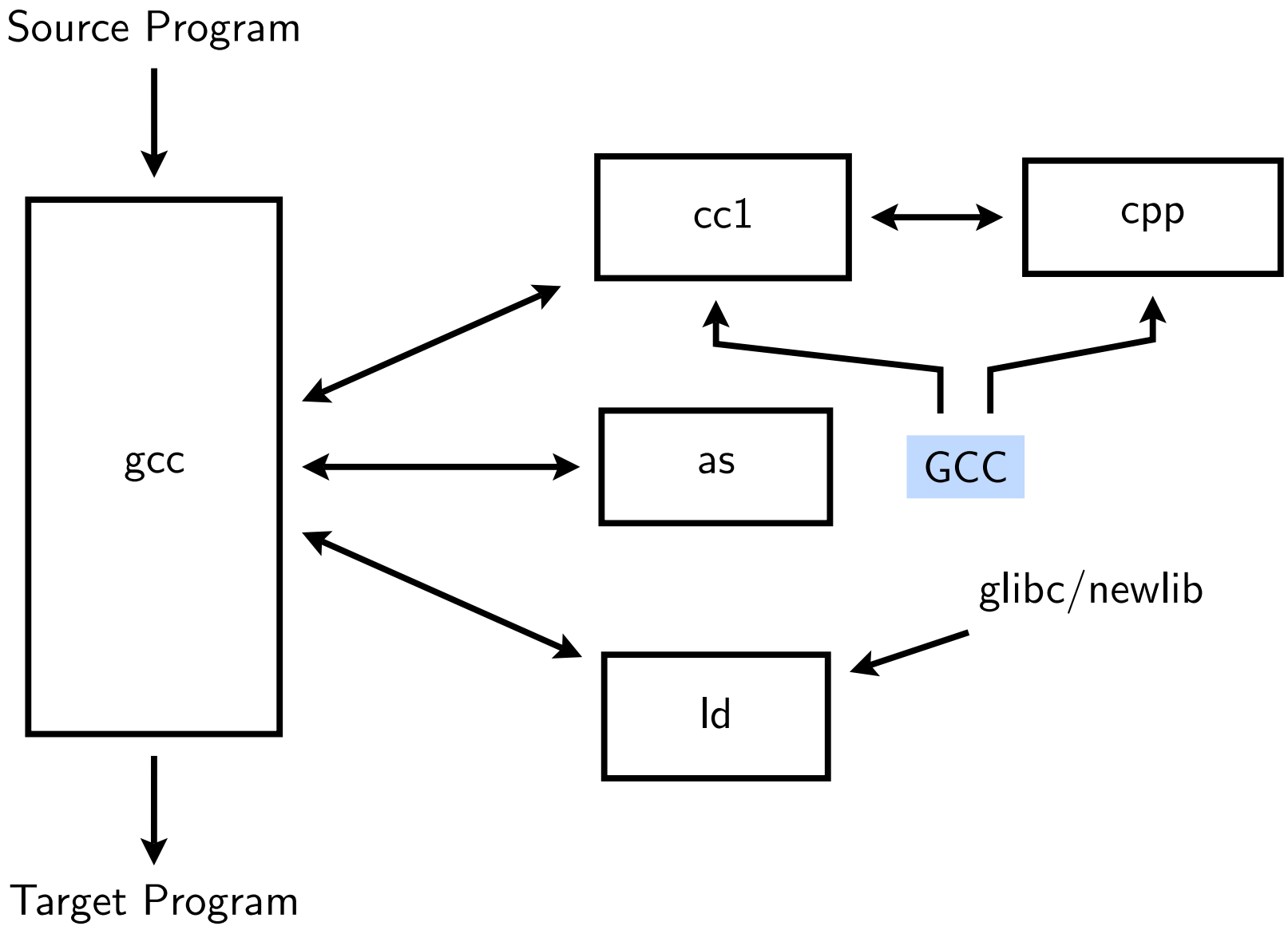
The GNU Tool Chain for C



The GNU Tool Chain for C



The GNU Tool Chain for C



Comprehensiveness of GCC: Wide Applicability

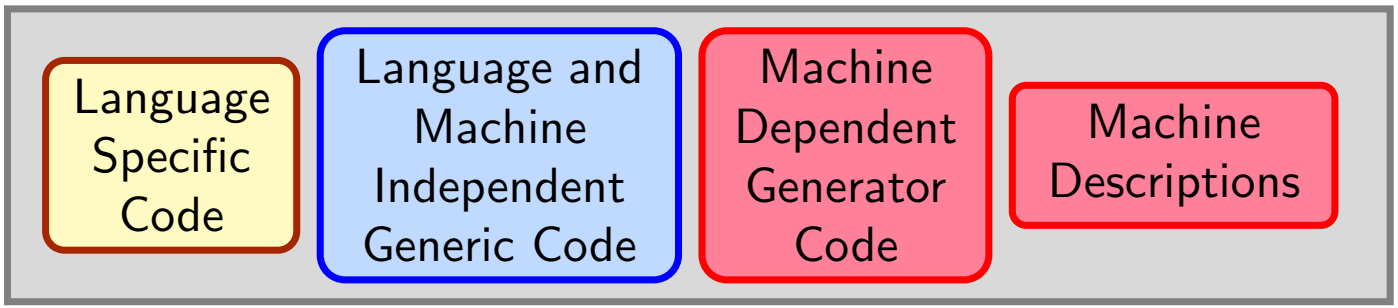
- Input languages supported:

  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada

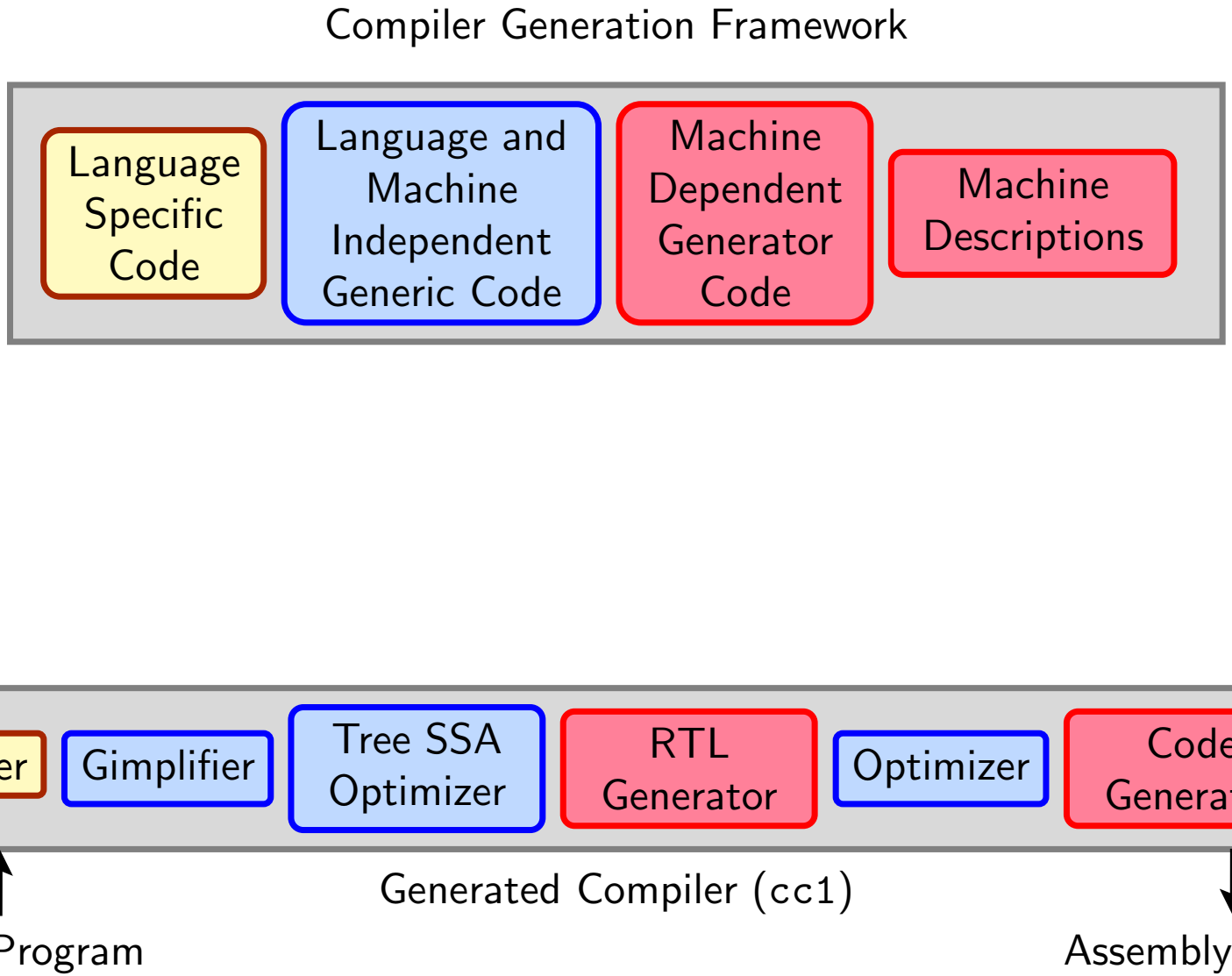
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios, PDP-10, TIGCC (m68k variant), Z8000, PIC24/dsPIC, NEC SX architecture

The Architecture of GCC

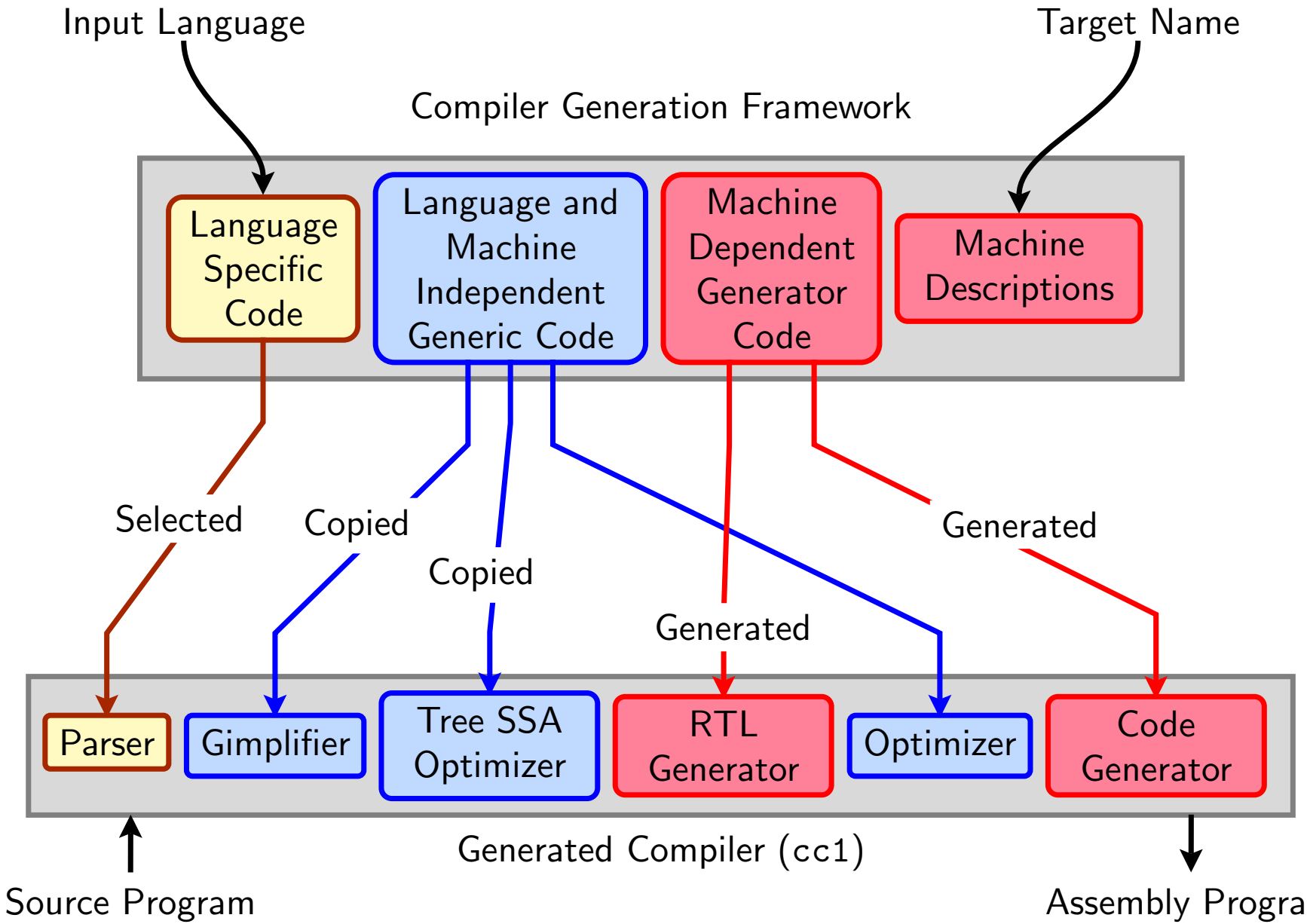
Compiler Generation Framework



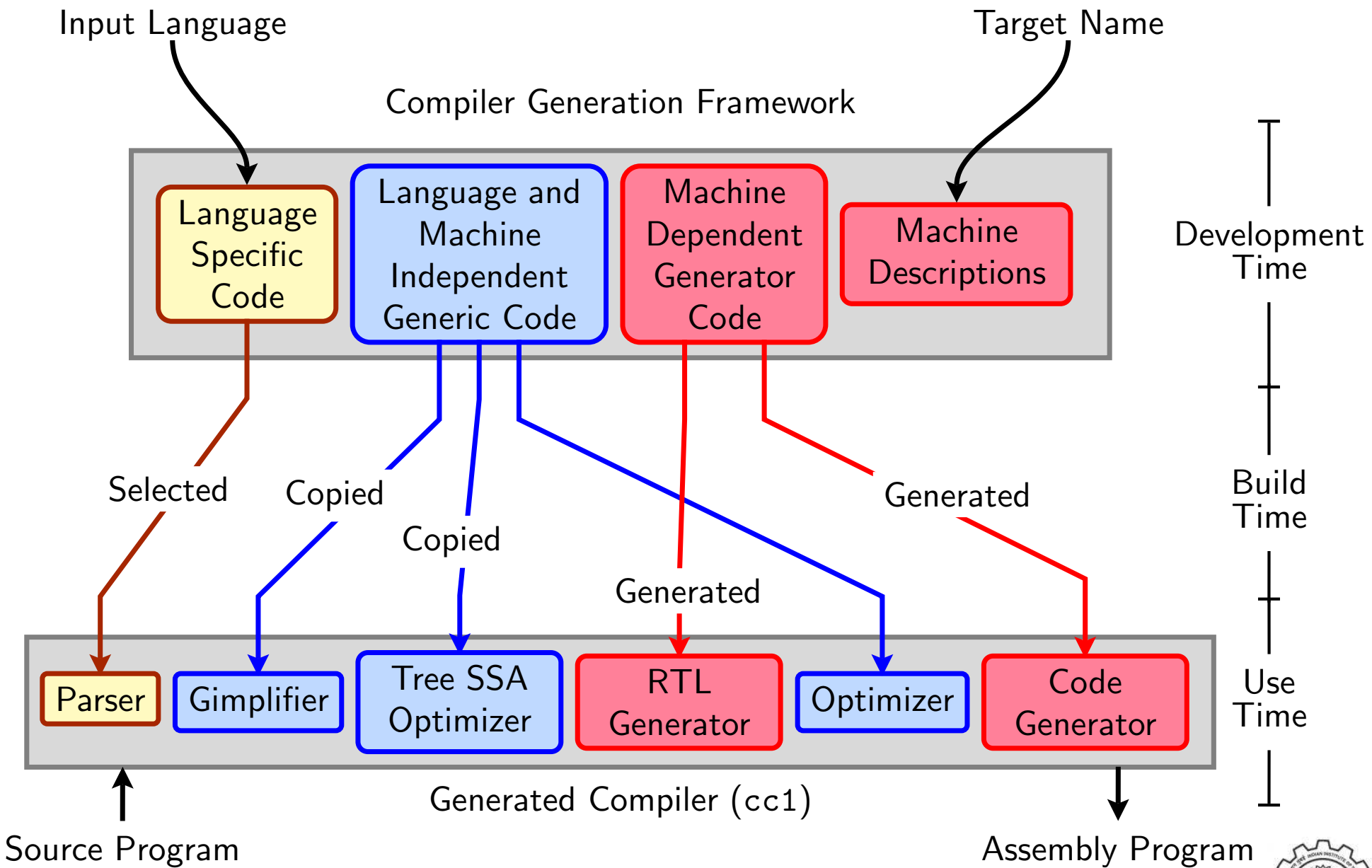
The Architecture of GCC



The Architecture of GCC



The Architecture of GCC



Comprehensiveness of GCC: Size

- Overall size

	Subdirectories	Files
gcc-4.4.2	3794	62301
gcc-4.6.0	4383	71096
gcc-4.7.2	4658	76287

- Core size (src/gcc)

	Subdirectories	Files
gcc-4.4.2	257	30163
gcc-4.6.0	336	36503
gcc-4.7.2	402	40193

- Machine Descriptions (src/gcc/config)

	Subdirectories	.c files	.h files	.md files
gcc-4.4.2	36	241	426	206
gcc-4.6.0	42	275	466	259
gcc-4.7.2	43	103	452	290

ohcount: Line Count of gcc-4.7.2

Language	Files	Code	Comment	Comment %	Blank	Total
c	20857	2289353	472640	17.1%	449939	3211932
cpp	23370	1030227	243717	19.1%	224079	1498023
ada	4913	726638	334360	31.5%	252044	1313042
java	6342	681938	645506	48.6%	169046	1496490
autoconf	94	428267	523	0.1%	66647	495437
html	336	151194	5667	3.6%	33877	190738
fortranfixed	3256	112286	2010	1.8%	15599	129895
make	106	110762	3875	3.4%	13811	128448
xml	76	50179	571	1.1%	6048	56798
assembler	240	49903	10975	18.0%	8584	69462
shell	157	49148	10848	18.1%	6757	66753
objective_c	882	28226	5267	15.7%	8324	41817
fortranfree	872	14474	3445	19.2%	1817	19736
tex	2	11060	5776	34.3%	1433	18269
scheme	6	11023	1010	8.4%	1205	13238
automake	72	10496	1179	10.1%	1582	13257
perl	29	4551	1322	22.5%	854	6727
ocaml	6	2830	576	16.9%	378	3784
xslt	20	2805	436	13.5%	563	3804
awk	16	2103	556	20.9%	352	3011
python	10	1672	400	19.3%	400	2472
css	25	1590	143	8.3%	332	2065
pascal	4	1044	141	11.9%	218	1403
csharp	9	879	506	36.5%	230	1615
dcl	2	402	84	17.3%	13	499
tcl	1	392	113	22.4%	72	577
javascript	3	208	87	29.5%	33	328
haskell	49	153	0	0.0%	17	170
matlab	2	57	0	0.0%	8	65
Total	61760	5773867	1751733	23.3%	1264262	8789862

Language	Files	Code	Comment	Comment %	Blank	Total
c	20857	2289353	472640	17.1%	449939	3211932
cpp	23370	1030227	243717	19.1%	224079	1498023
ada	4913	726638	334360	31.5%	252044	1313042
java	6342	681938	645506	48.6%	169046	1496490
autoconf	94	428267	523	0.1%	66647	495437
html	336	151194	5667	3.6%	33877	190738
fortranfixed	3256	112286	2010	1.8%	15599	129895
make	106	110762	3875	3.4%	13811	128448
xml	76	50179	571	1.1%	6048	56798
assembler	240	49903	10975	18.0%	8584	69462
shell	157	49148	10848	18.1%	6757	66753
objective_c	882	28226	5267	15.7%	8324	41817
fortranfree	872	14474	3445	19.2%	1817	19736
tex	2	11060	5776	34.3%	1433	18269
scheme	6	11023	1010	8.4%	1205	13238
automake	72	10496	1179	10.1%	1582	13257
perl	29	4551	1322	22.5%	854	6727
ocaml	6	2830	576	16.9%	378	3784
xslt	20	2805	436	13.5%	563	3804
awk	16	2103	556	20.9%	352	3011
python	10	1672	400	19.3%	400	2472
css	25	1590	143	8.3%	332	2065
pascal	4	1044	141	11.9%	218	1403
csharp	9	879	506	36.5%	230	1615
dcl	2	402	84	17.3%	13	499
tcl	1	392	113	22.4%	72	577
javascript	3	208	87	29.5%	33	328
haskell	49	153	0	0.0%	17	170
matlab	2	57	0	0.0%	8	65
Total	61760	5773867	1751733	23.3%	1264262	8789862

ohcount: Line Count of gcc-4.7.2/gcc

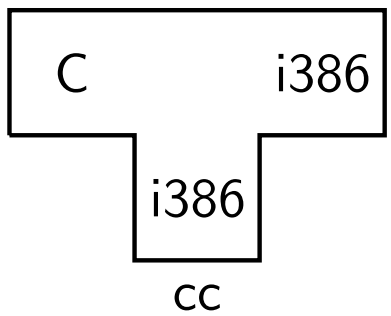
Language	Files	Code	Comment	Comment %	Blank	Total
c	17849	1601863	335879	17.3%	344693	2282435
ada	4903	724957	333800	31.5%	251445	1310202
cpp	9563	275971	63875	18.8%	71647	411493
fortranfixed	3158	105987	1961	1.8%	15175	123123
autoconf	3	30014	12	0.0%	4139	34165
objective_c	877	28017	5109	15.4%	8249	41375
fortranfree	834	13516	3234	19.3%	1716	18466
scheme	6	11023	1010	8.4%	1205	13238
make	6	6248	1113	15.1%	916	8277
tex	1	5441	2835	34.3%	702	8978
ocaml	6	2830	576	16.9%	378	3784
shell	22	2265	735	24.5%	391	3391
awk	11	1646	390	19.2%	271	2307
perl	3	913	226	19.8%	163	1302
assembler	7	343	136	28.4%	27	506
haskell	49	153	0	0.0%	17	170
matlab	2	57	0	0.0%	8	65
Total	37300	2811244	750891	21.1%	701142	4263277

Language	Files	Code	Comment	Comment %	Blank	Total
c	17849	1601863	335879	17.3%	344693	2282435
ada	4903	724957	333800	31.5%	251445	1310202
cpp	9563	275971	63875	18.8%	71647	411493
fortranfixed	3158	105987	1961	1.8%	15175	123123
autoconf	3	30014	12	0.0%	4139	34165
objective_c	877	28017	5109	15.4%	8249	41375
fortranfree	834	13516	3234	19.3%	1716	18466
scheme	6	11023	1010	8.4%	1205	13238
make	6	6248	1113	15.1%	916	8277
tex	1	5441	2835	34.3%	702	8978
ocaml	6	2830	576	16.9%	378	3784
shell	22	2265	735	24.5%	391	3391
awk	11	1646	390	19.2%	271	2307
perl	3	913	226	19.8%	163	1302
assembler	7	343	136	28.4%	27	506
haskell	49	153	0	0.0%	17	170
matlab	2	57	0	0.0%	8	65
Total	37300	2811244	750891	21.1%	701142	4263277

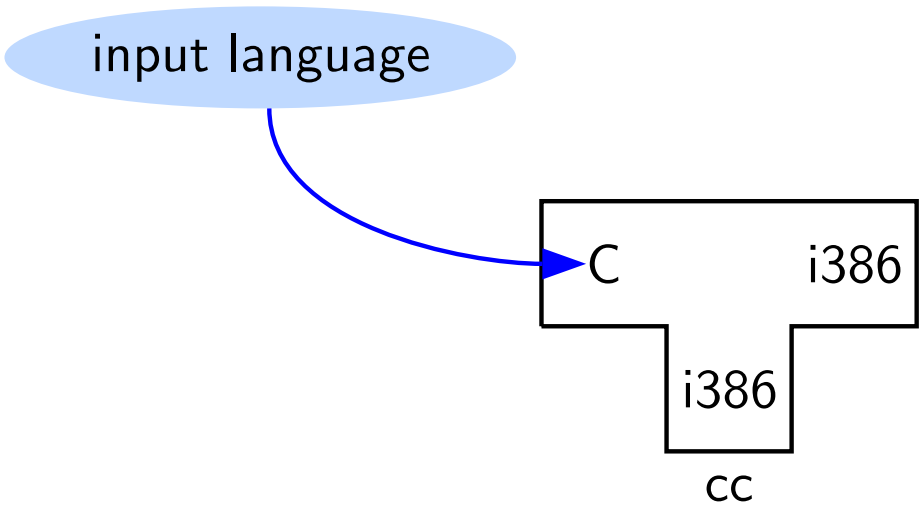
Part 2

Configuring and Building GCC

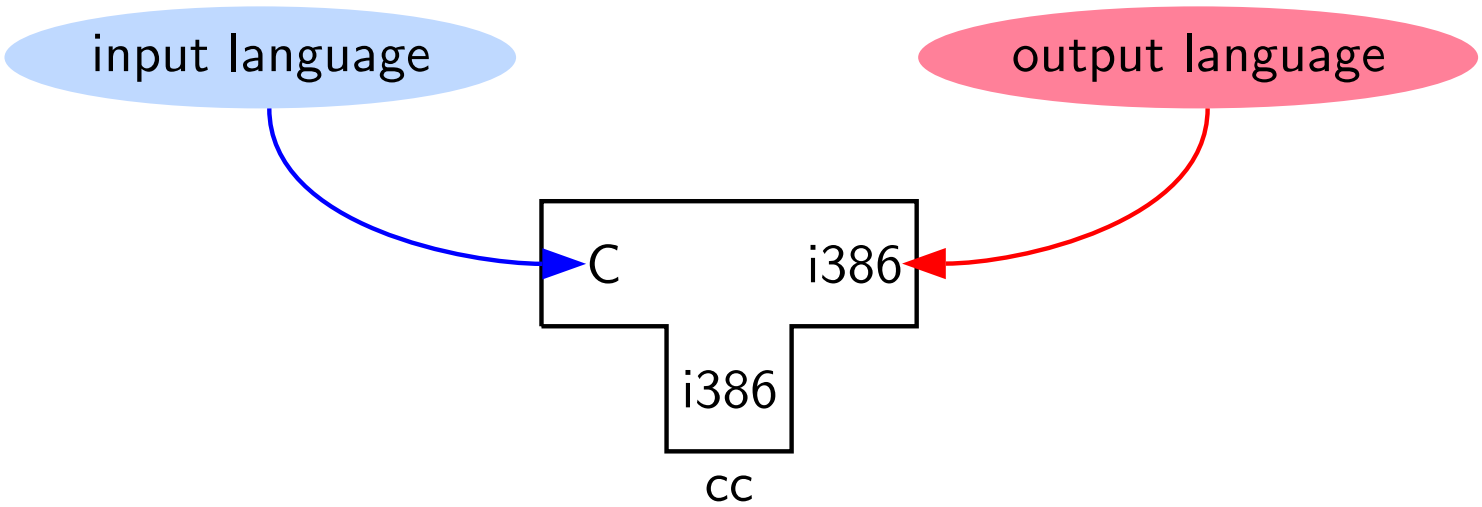
T Notation for a Compiler



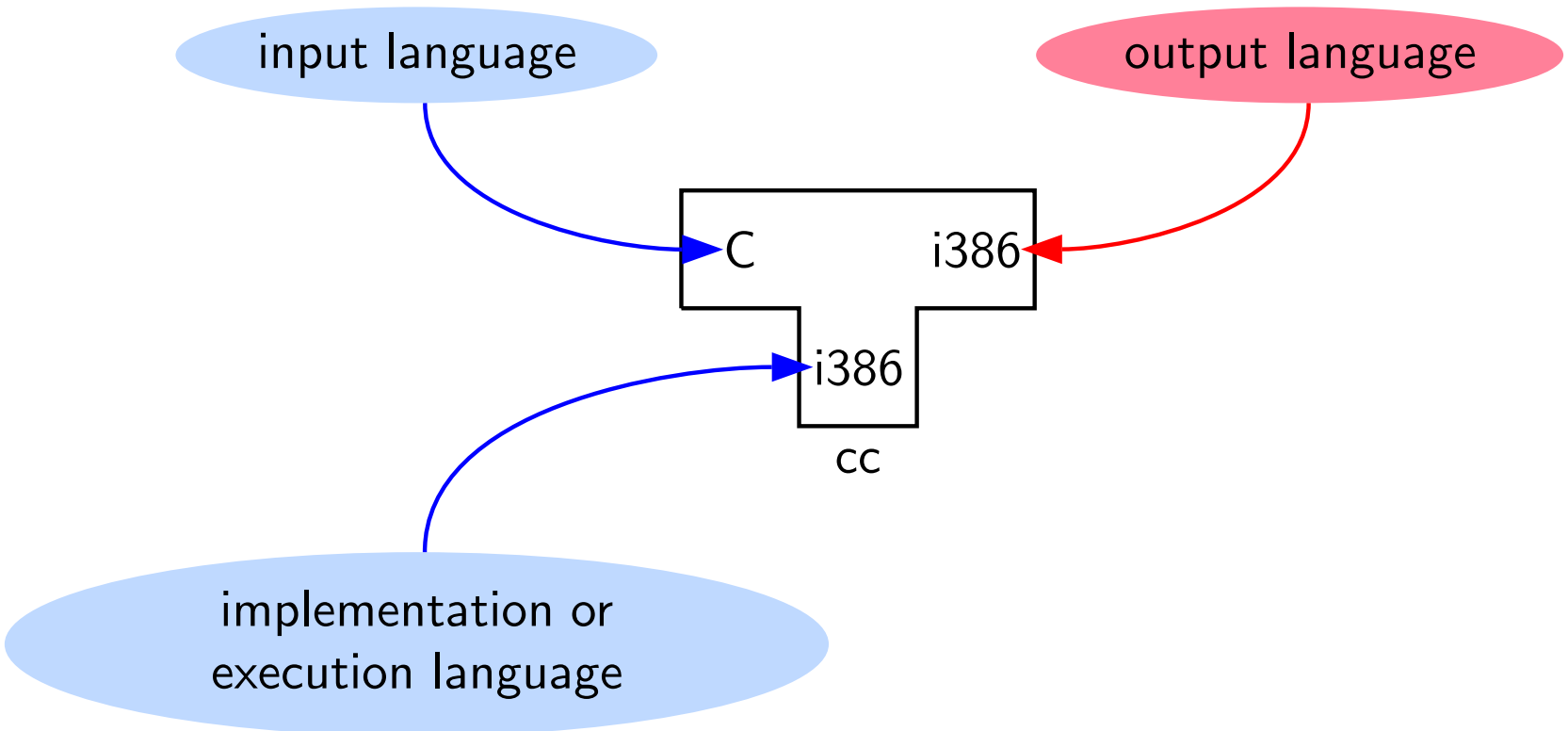
T Notation for a Compiler



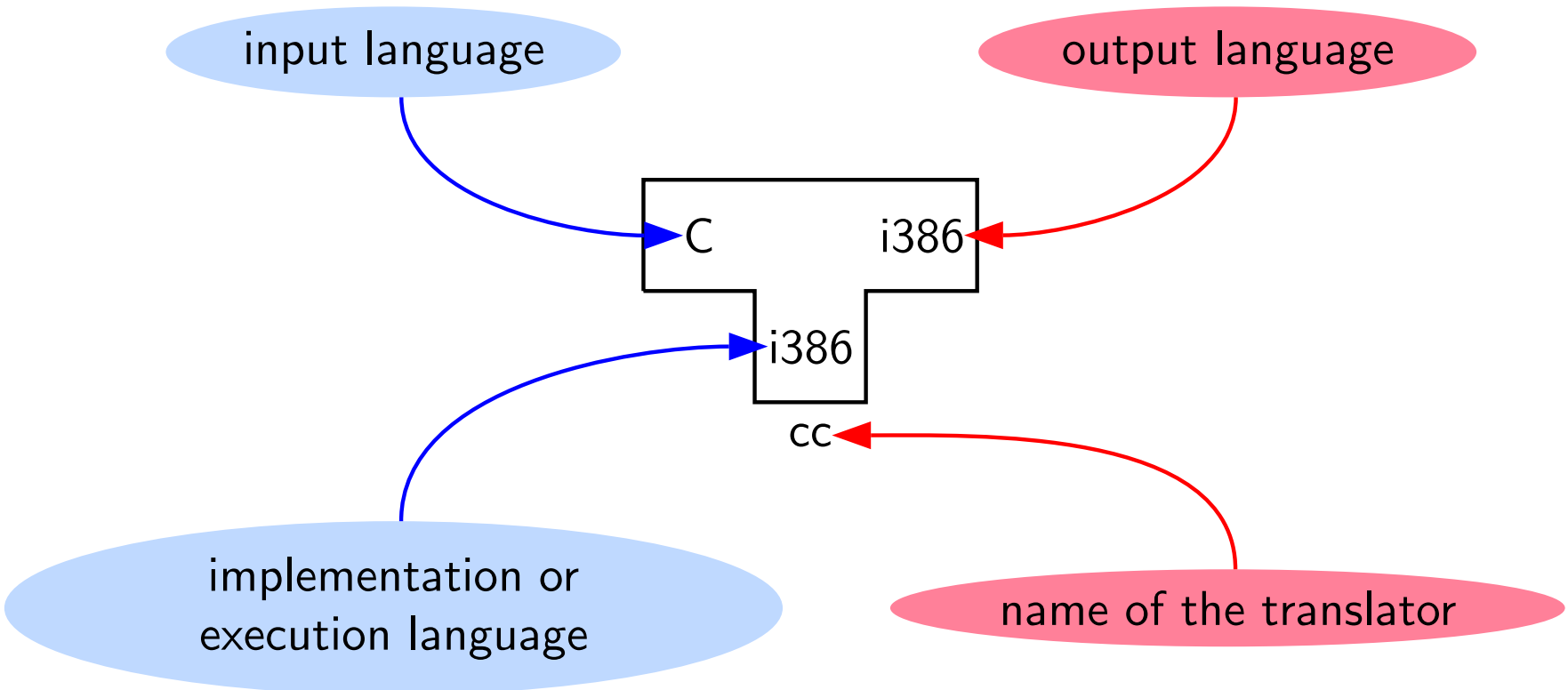
T Notation for a Compiler



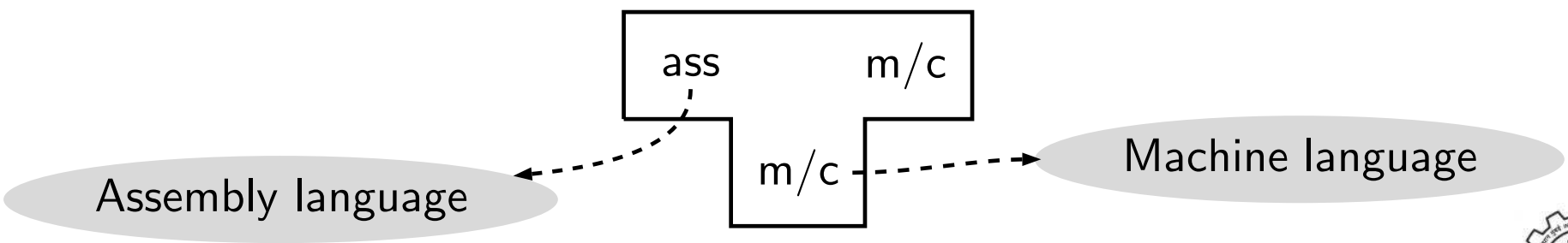
T Notation for a Compiler



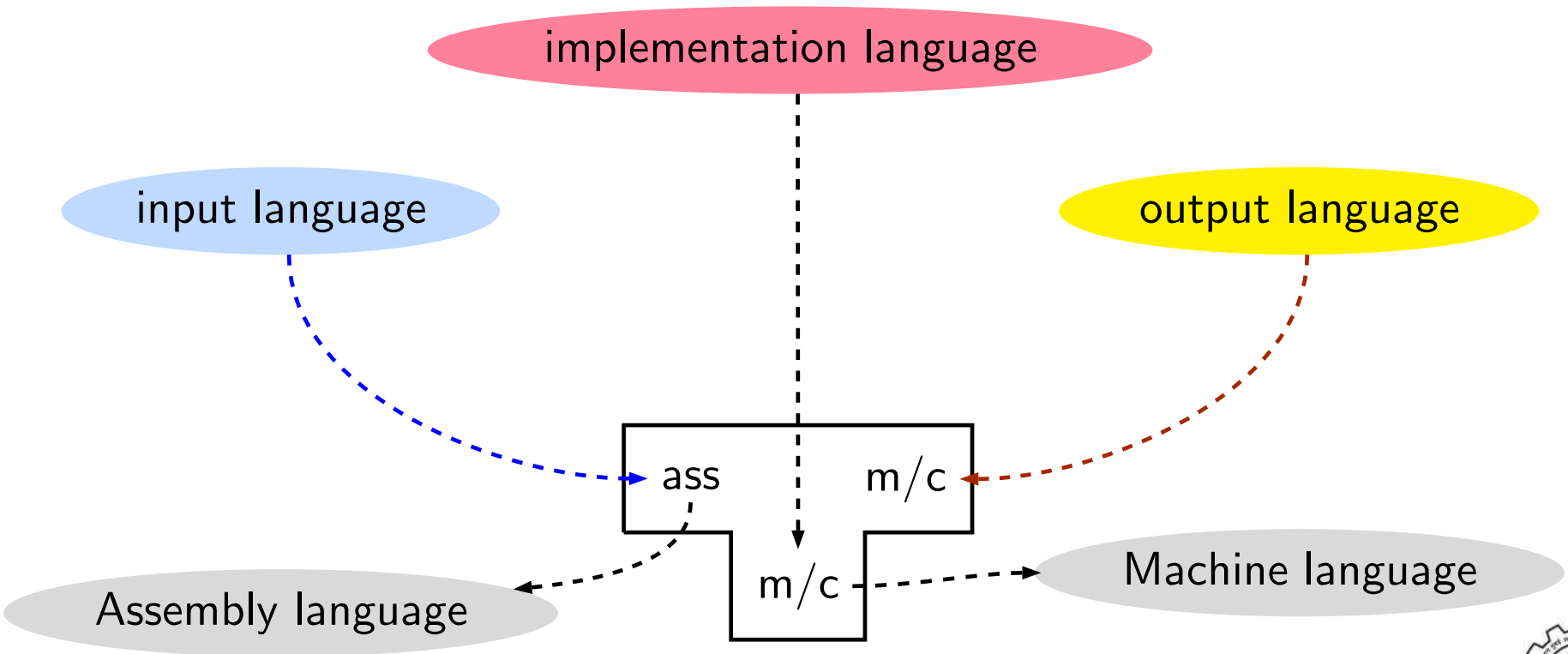
T Notation for a Compiler



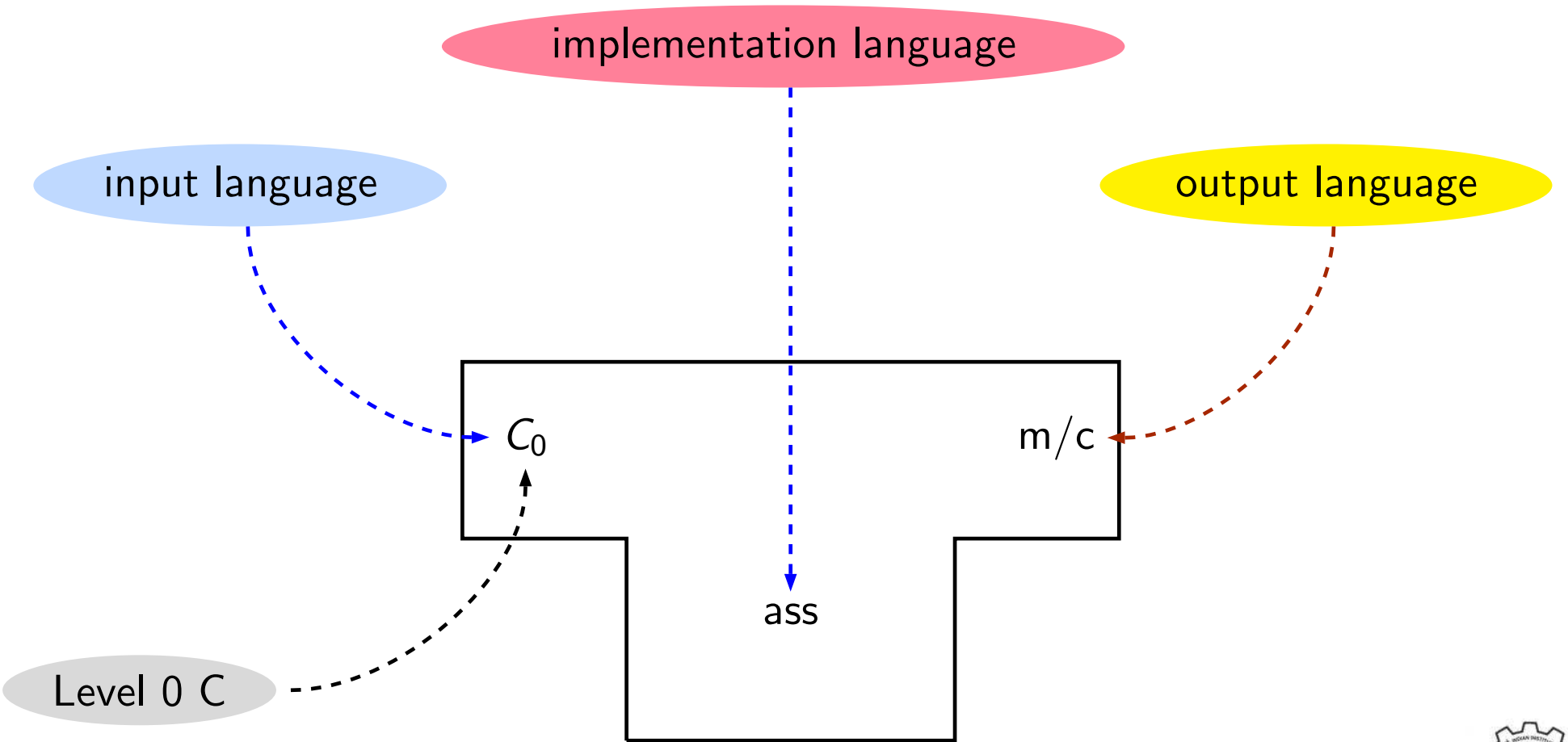
Bootstrapping: The Conventional View



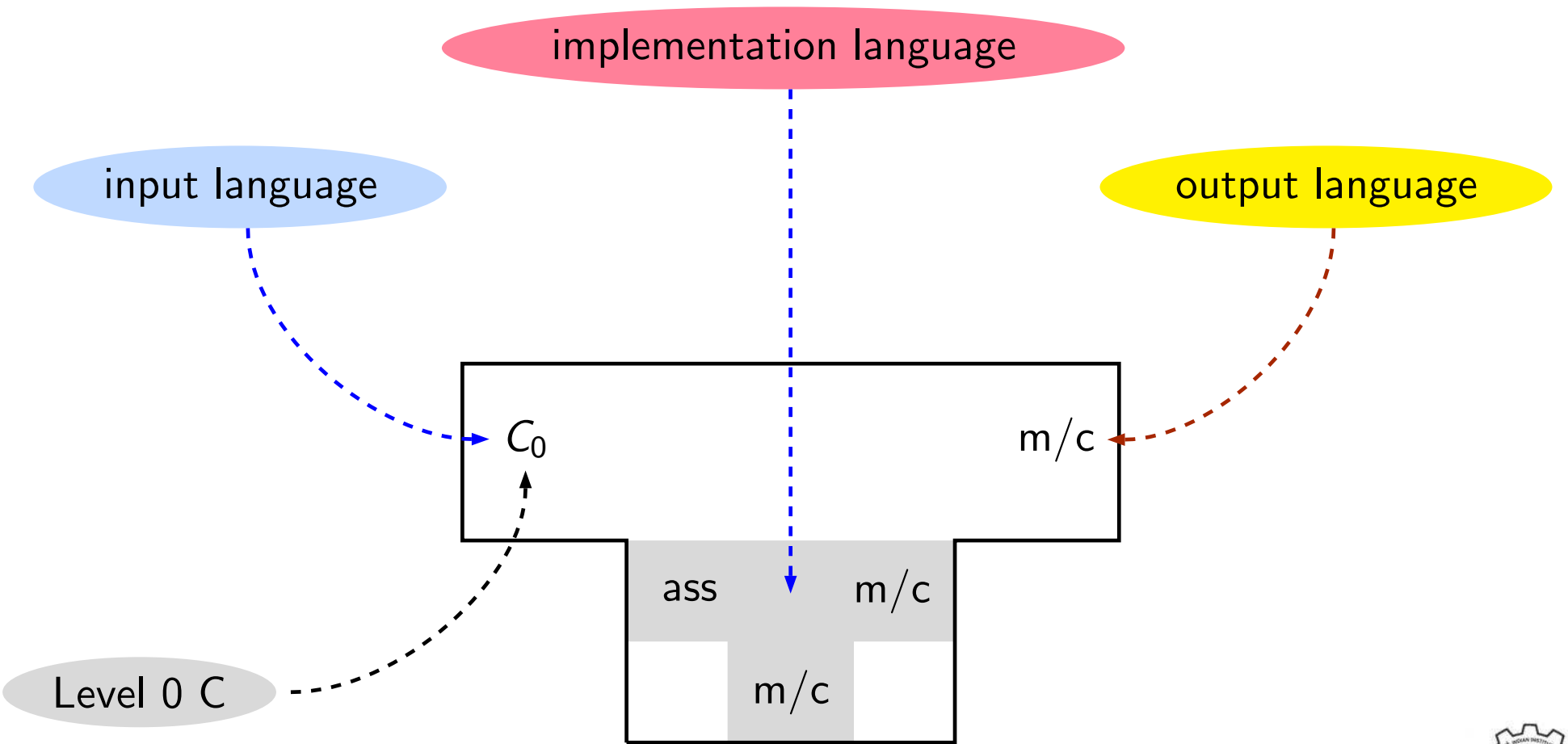
Bootstrapping: The Conventional View



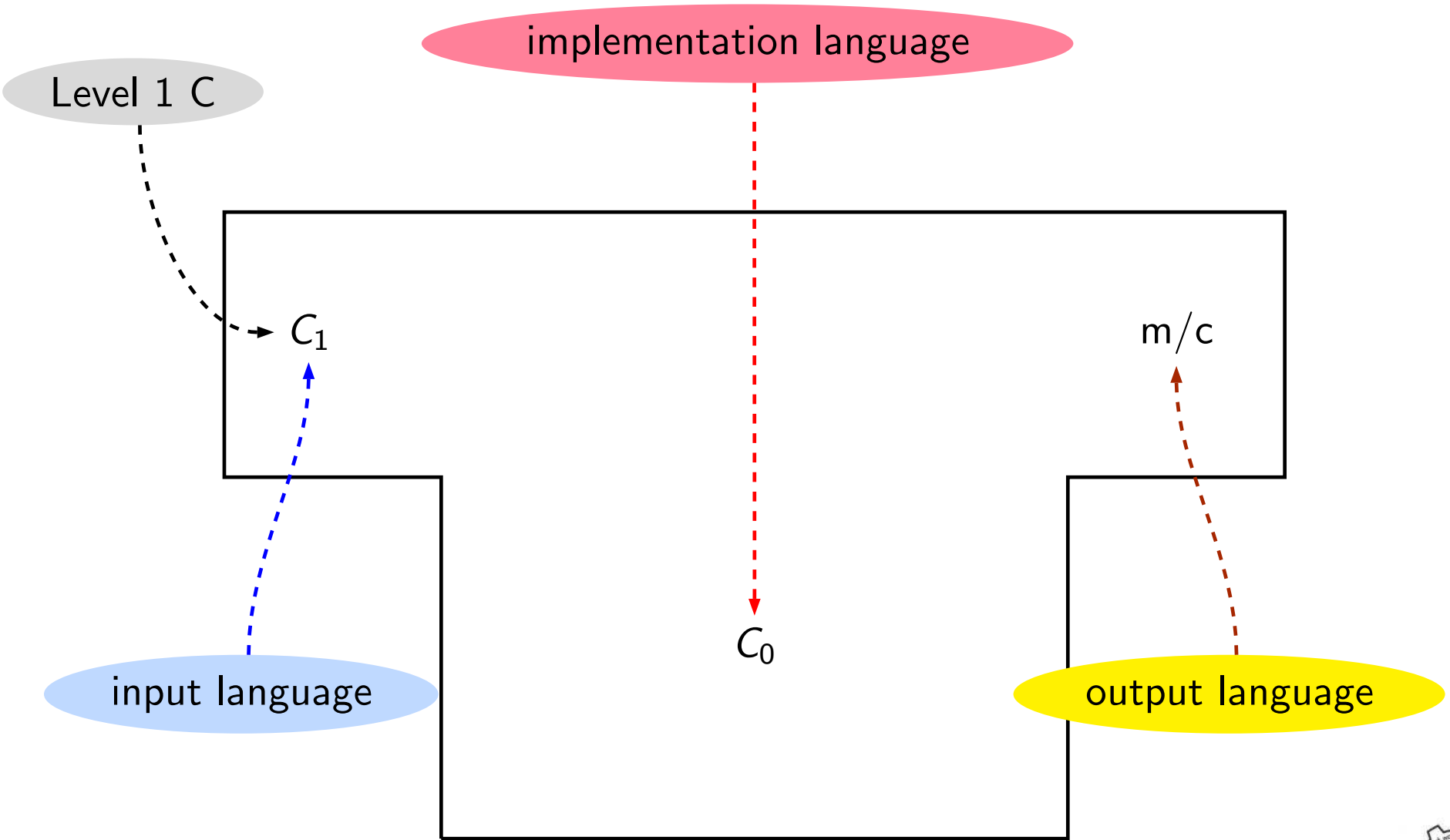
Bootstrapping: The Conventional View



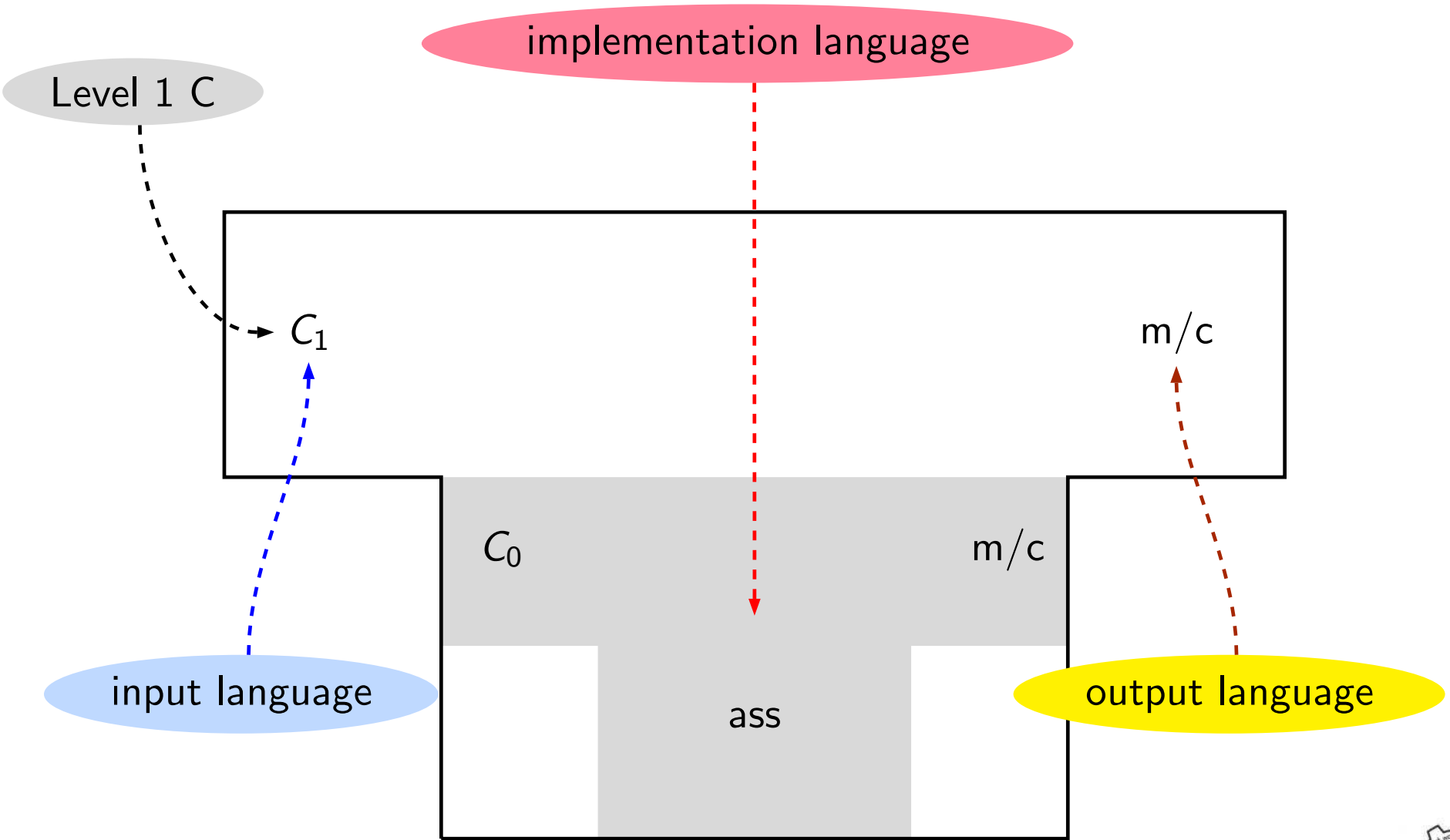
Bootstrapping: The Conventional View



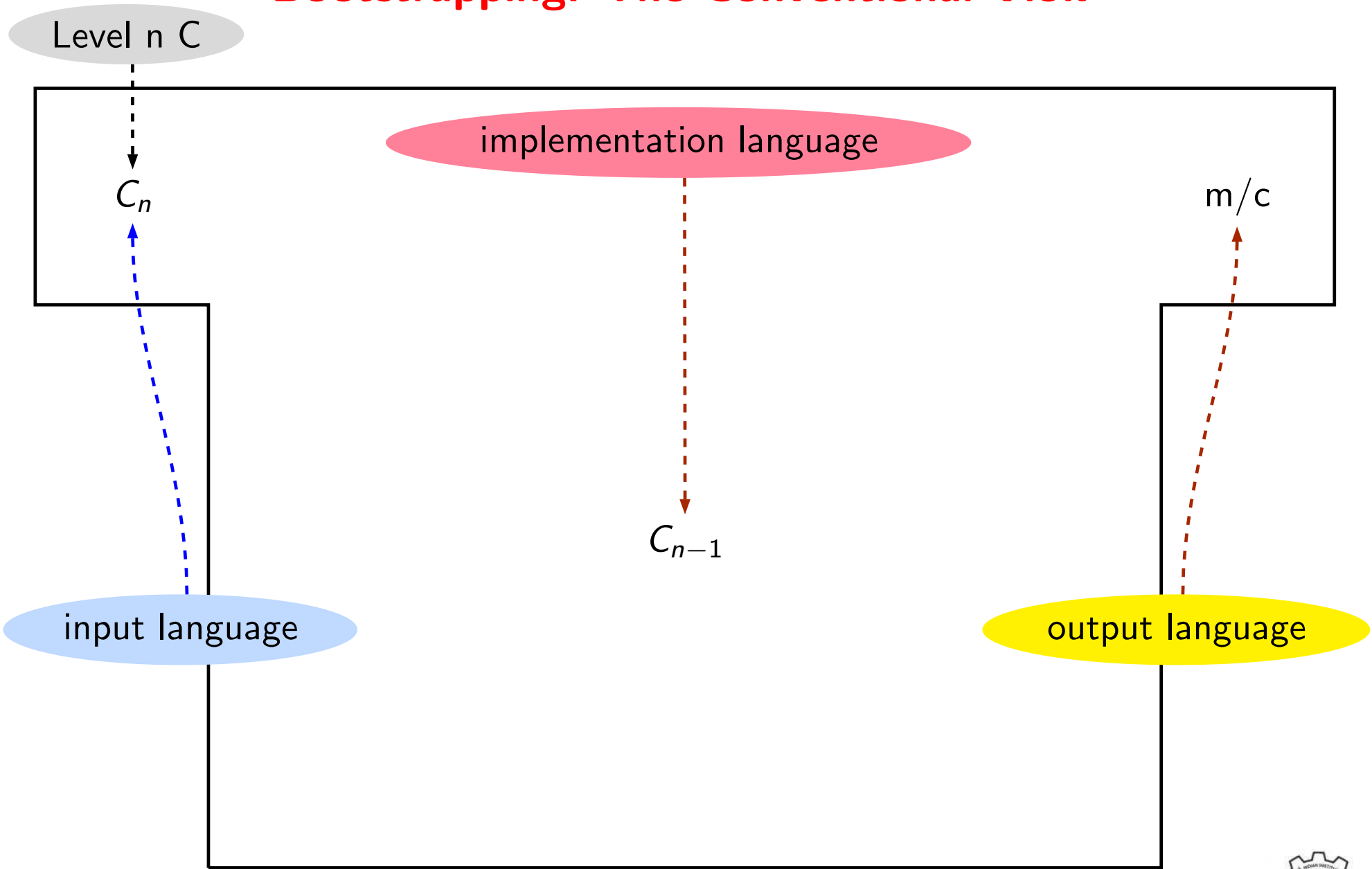
Bootstrapping: The Conventional View



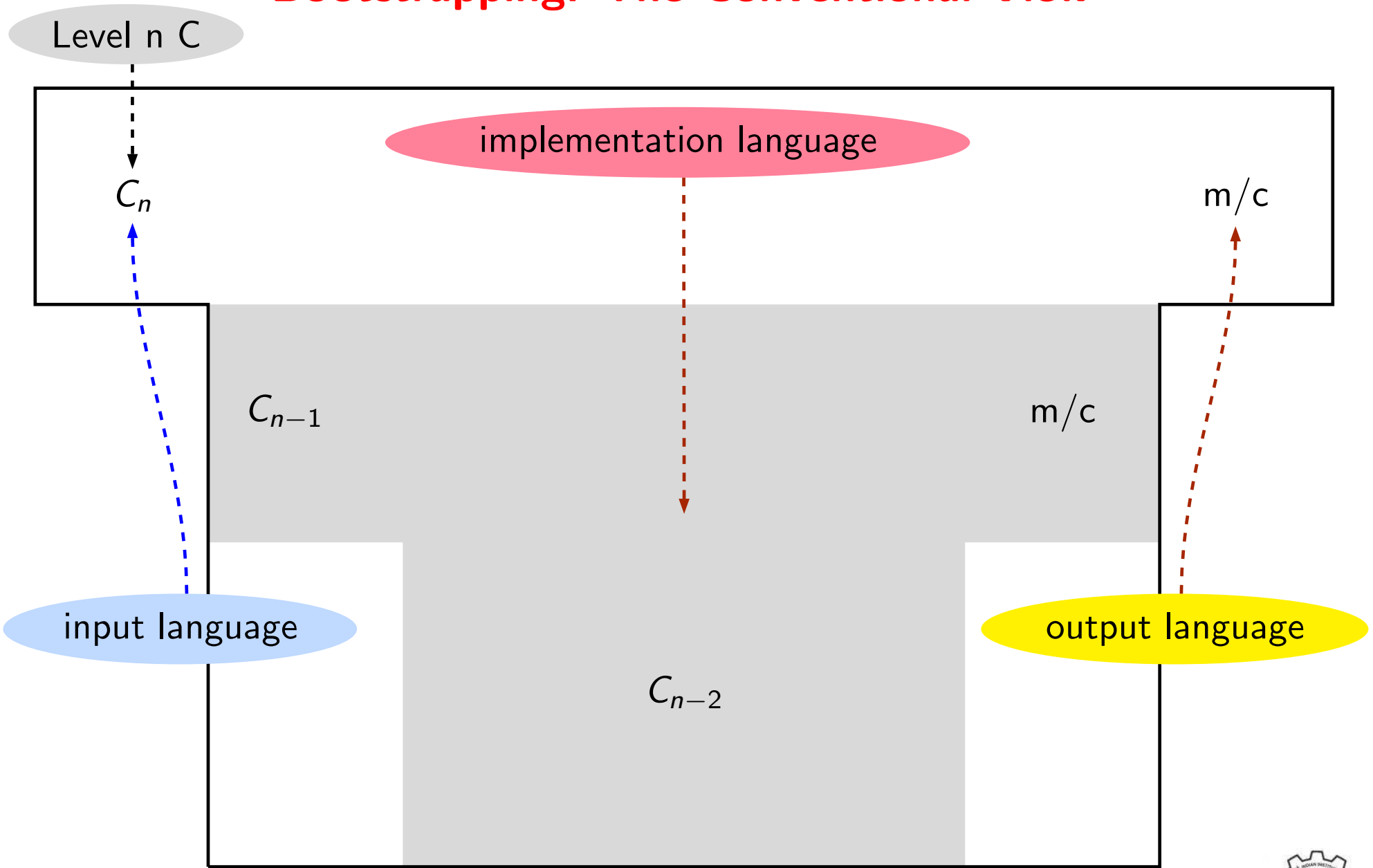
Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Bootstrapping: GCC View

- Language need not change, but the compiler may change

  Compiler is improved, bugs are fixed and newer versions are released

- To build a new version of a compiler given a *built* old version:
  - Stage 1: Build the new compiler using the old compiler
  - Stage 2: Build another new compiler using compiler from stage 1
  - Stage 3: Build another new compiler using compiler from stage 2

    Stage 2 and stage 3 builds must result in identical compilers

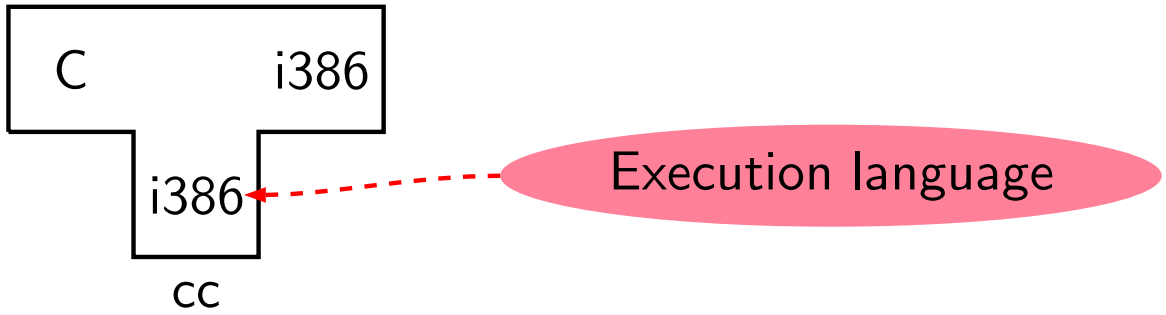
$\Rightarrow$ Building cross compilers stops after Stage 1!

A Native Build on i386

GCC
Source

Requirement: BS = HS = TS = i386

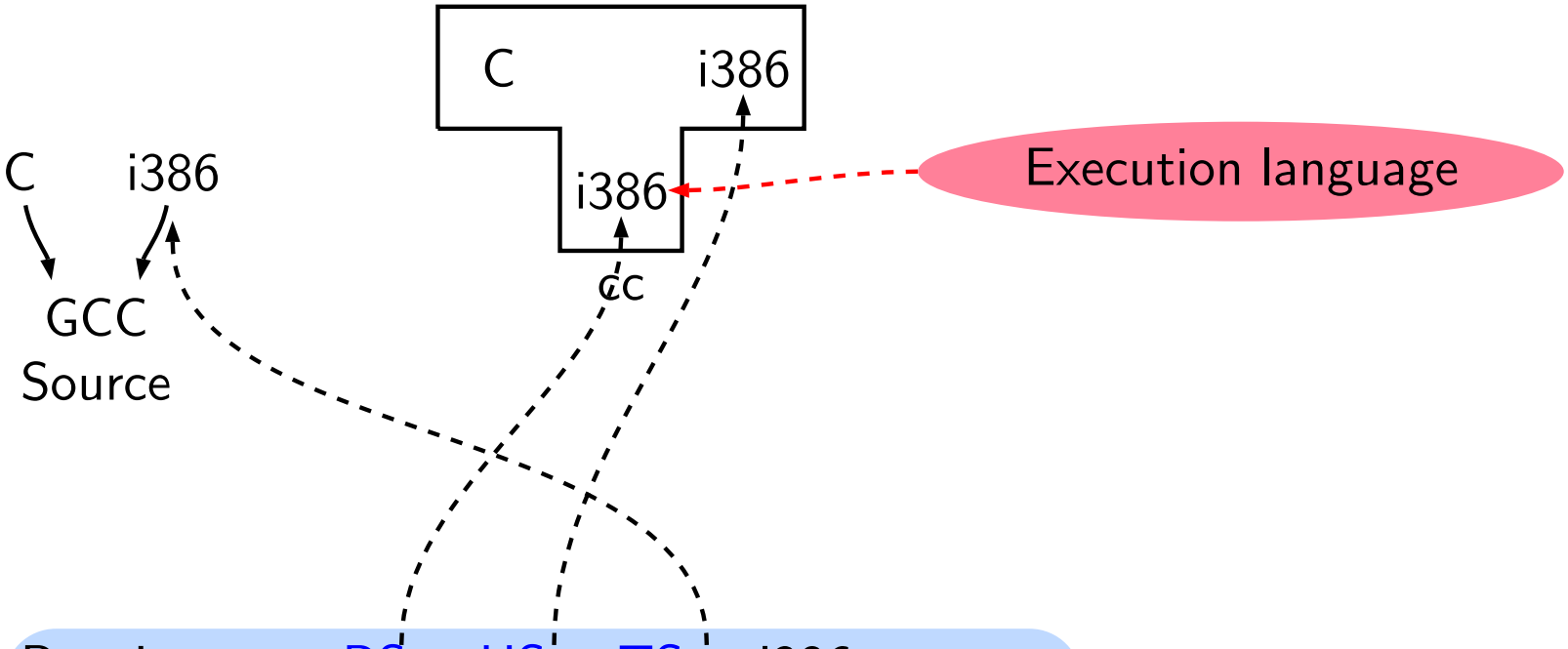
A Native Build on i386



GCC Source

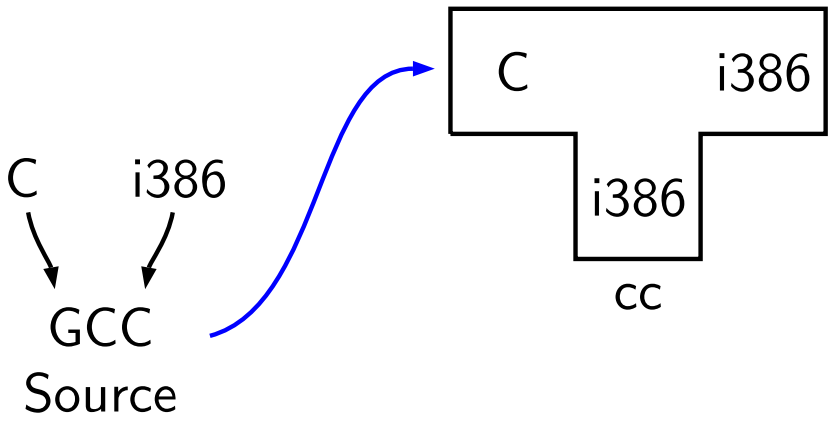
Requirement: BS = HS = TS = i386

A Native Build on i386



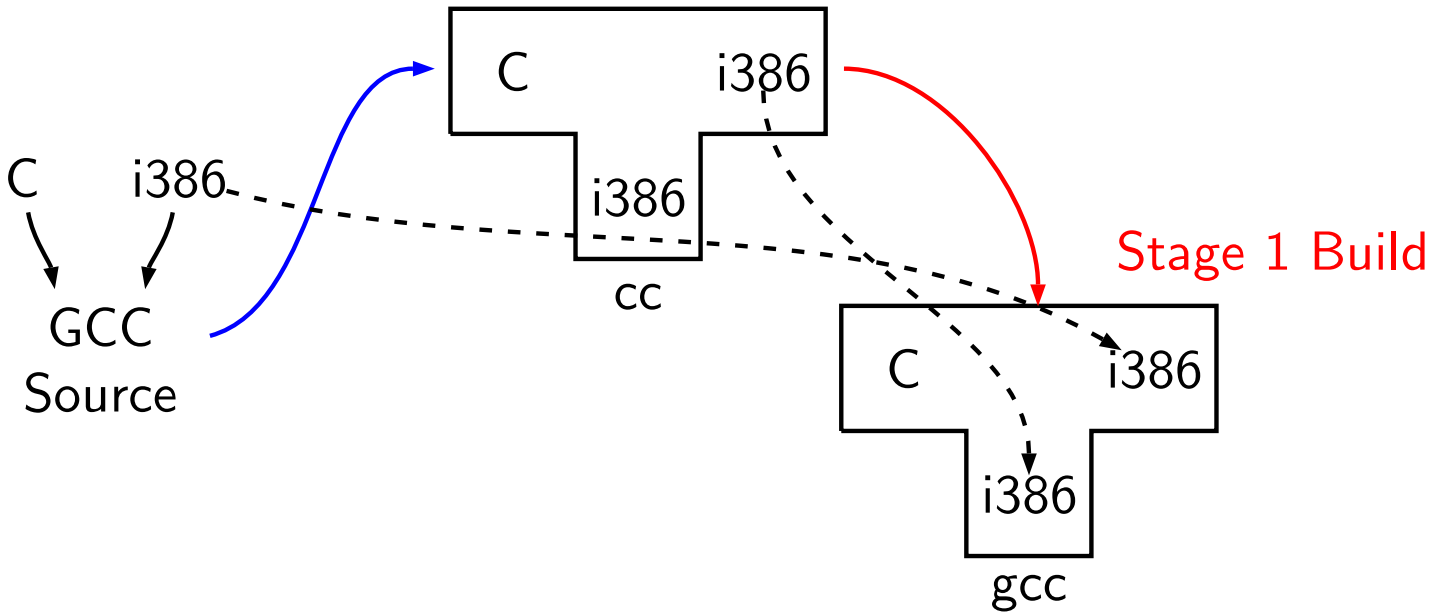
Requirement: BS = HS = TS = i386

A Native Build on i386



Requirement: BS = HS = TS = i386

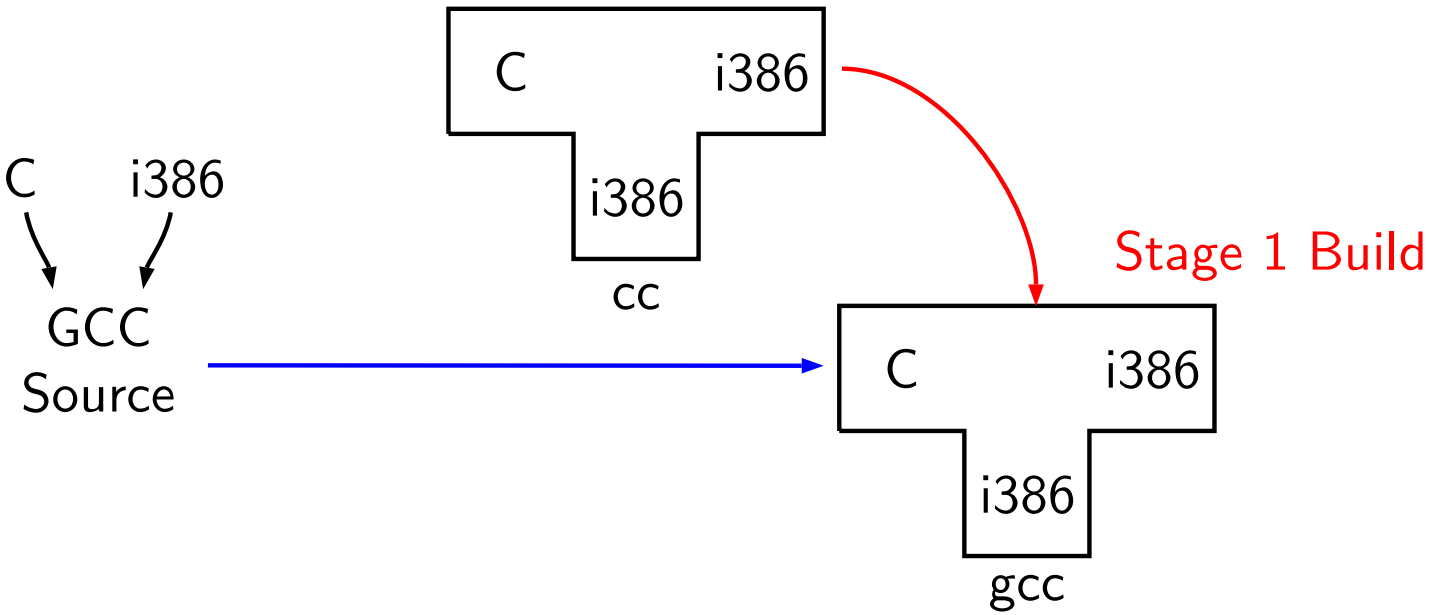
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

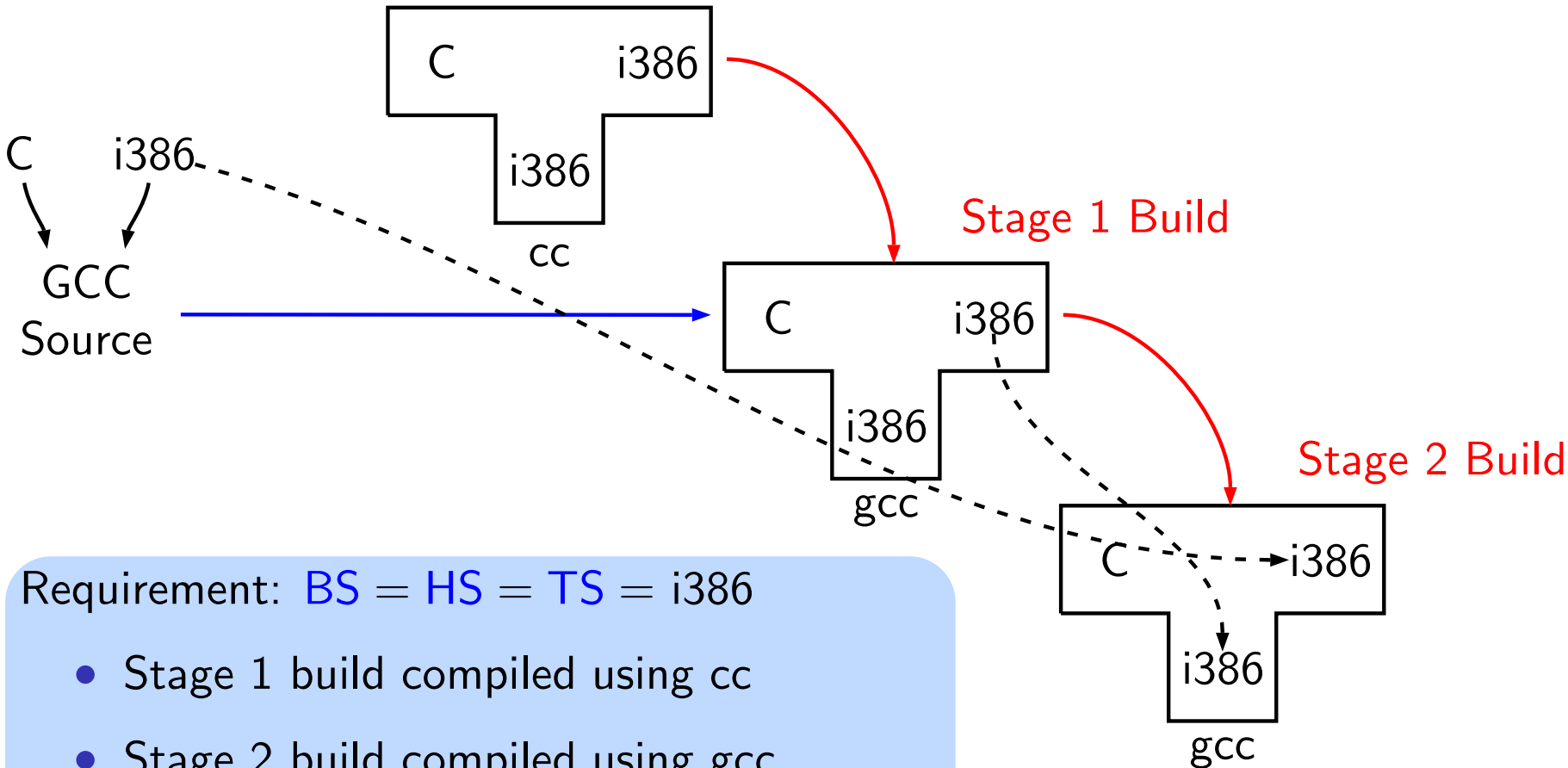
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

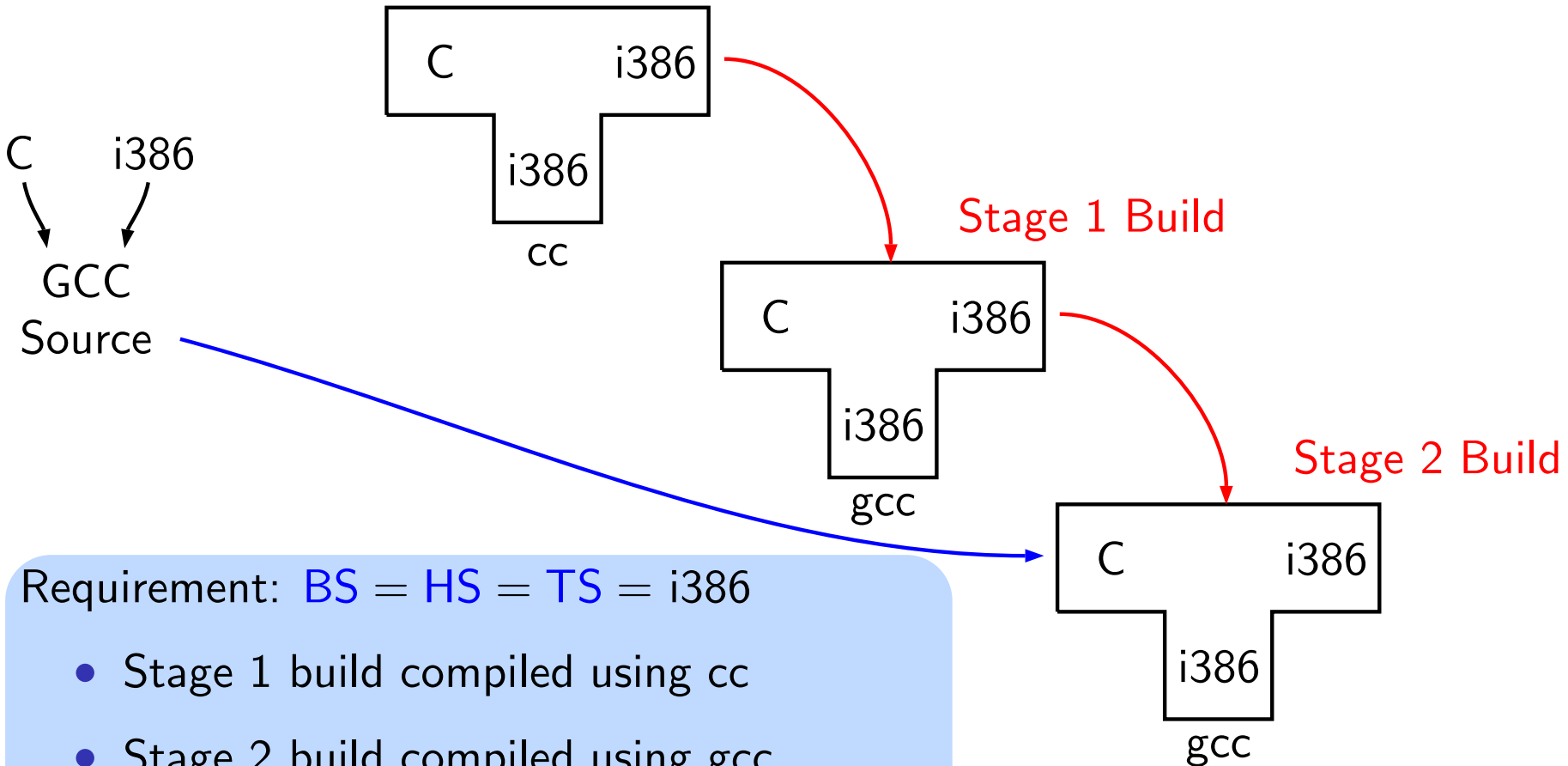
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc

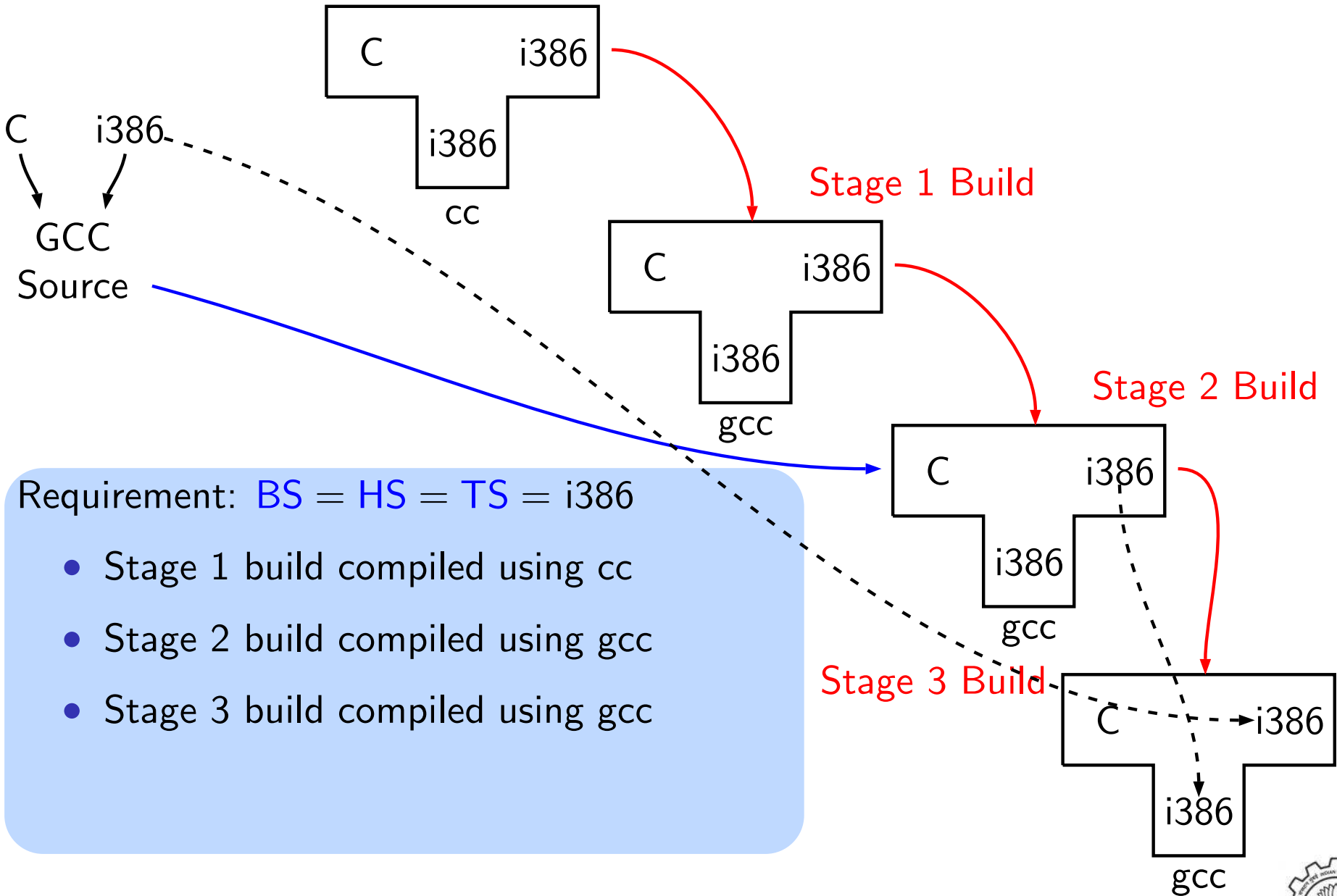
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc

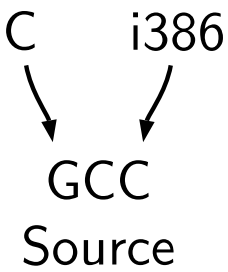
A Native Build on i386

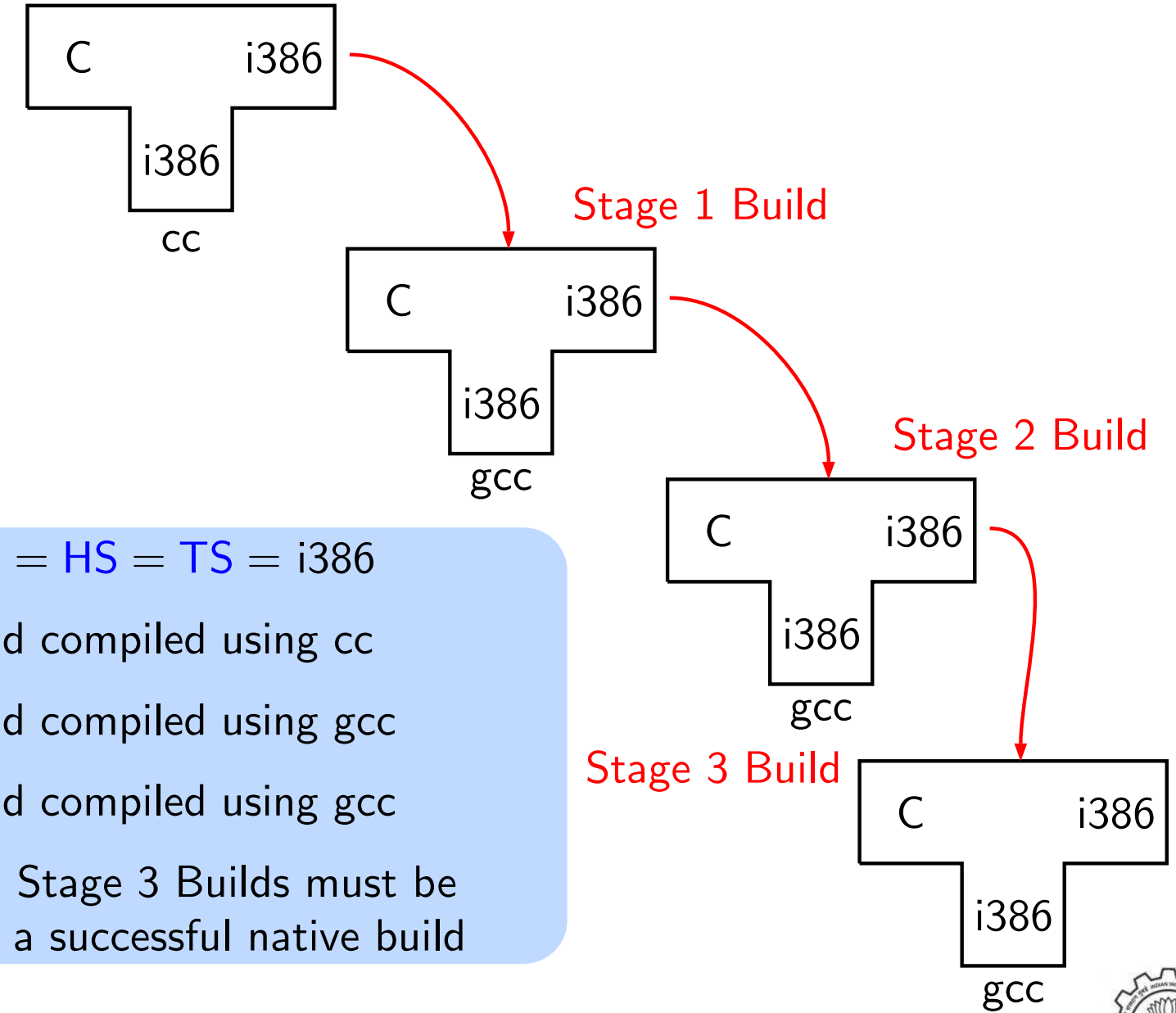


Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc

A Native Build on i386





Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc
- Stage 2 and Stage 3 Builds must be identical for a successful native build

Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`

Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE_D)/configure <options>`

  `configure` output: customized `Makefile`

Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE_D)/configure <options>`

  `configure` output: customized `Makefile`
- `make 2> make.err > make.log`

Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE_D)/configure <options>`

  `configure` output: customized `Makefile`
- `make 2> make.err > make.log`
- `make install 2> install.err > install.log`

Build for a Given Target

This is what actually happens!

- Generation
  - Generator sources (`$(SOURCE_D)/gcc/gen*.c`) are read and generator executables are created in `$(BUILD)/gcc/build`
  - MD files are read by the generator executables and back end source code is generated in `$(BUILD)/gcc`
- Compilation

  Other source files are read from `$(SOURCE_D)` and executables created in corresponding subdirectories of `$(BUILD)`
- Installation

  Created executables and libraries are copied in `$(INSTALL)`

Build for a Given Target

This is what actually happens!

- Generation
  - Generator sources (`$(SOURCE_D)/gcc/gen*.c`) are read and generator executables are created in `$(BUILD)/gcc/build`
  - MD files are read by the generator executables and back end source code is generated in `$(BUILD)/gcc`
- Compilation

  Other source files are read from `$(SOURCE_D)` and executables created in corresponding subdirectories of `$(BUILD)`
- Installation

  Created executables and libraries are copied in `$(INSTALL)`

```
genattr
gencheck
genconditions
genconstants
genflags
genopinit
genpreds
genattrtab
gencheckcsum
gencondmd
genemit
gengenrtl
genmddeps
genoutput
genrecog
genautomata
gencodes
genconfig
genextract
gengtype
genmodes
genpeep
```

Generated Compiler Executable for All Languages

• Main driver	`$BUILD/gcc/xgcc`
• C compiler	`$BUILD/gcc/cc1`
• C++ compiler	`$BUILD/gcc/cc1plus`
• Fortran compiler	`$BUILD/gcc/f951`
• Ada compiler	`$BUILD/gcc/gnat1`
• Java compiler	`$BUILD/gcc/jcl`
• Java compiler for generating main class	`$BUILD/gcc/jvgenmain`
• LTO driver	`$BUILD/gcc/lto1`
• Objective C	`$BUILD/gcc/cc1obj`
• Objective C++	`$BUILD/gcc/cc1objplus`

Part 3

Gray Box Probing

What is Gray Box Probing of GCC?

- Black Box probing:

  Examining only the input and output relationship of a system

- White Box probing:

  Examining internals of a system for a given set of inputs

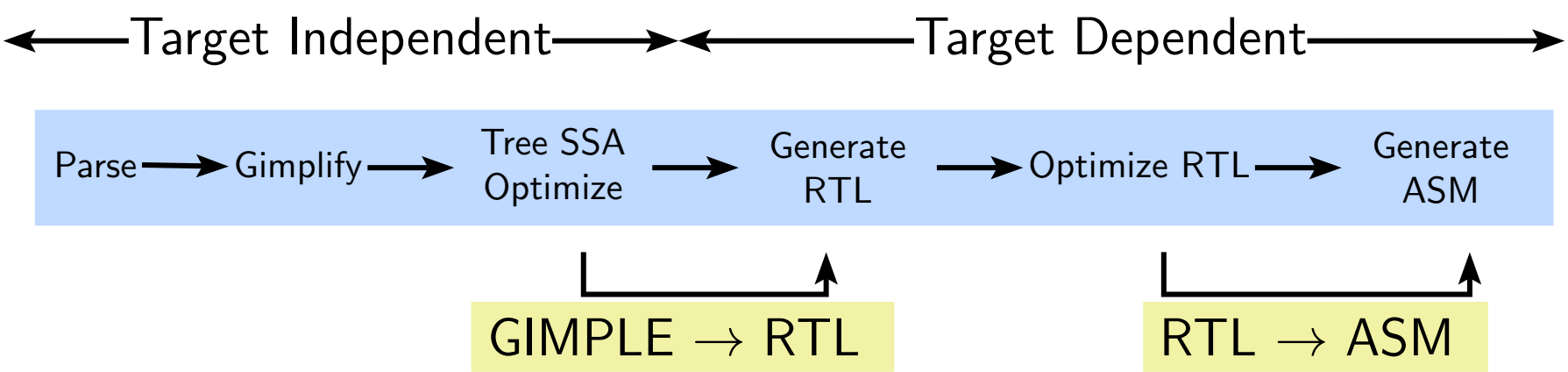
- Gray Box probing:

  Examining input and output of various components/modules

  - Overview of translation sequence in GCC
  - Overview of intermediate representations
  - Intermediate representations of programs across important phases

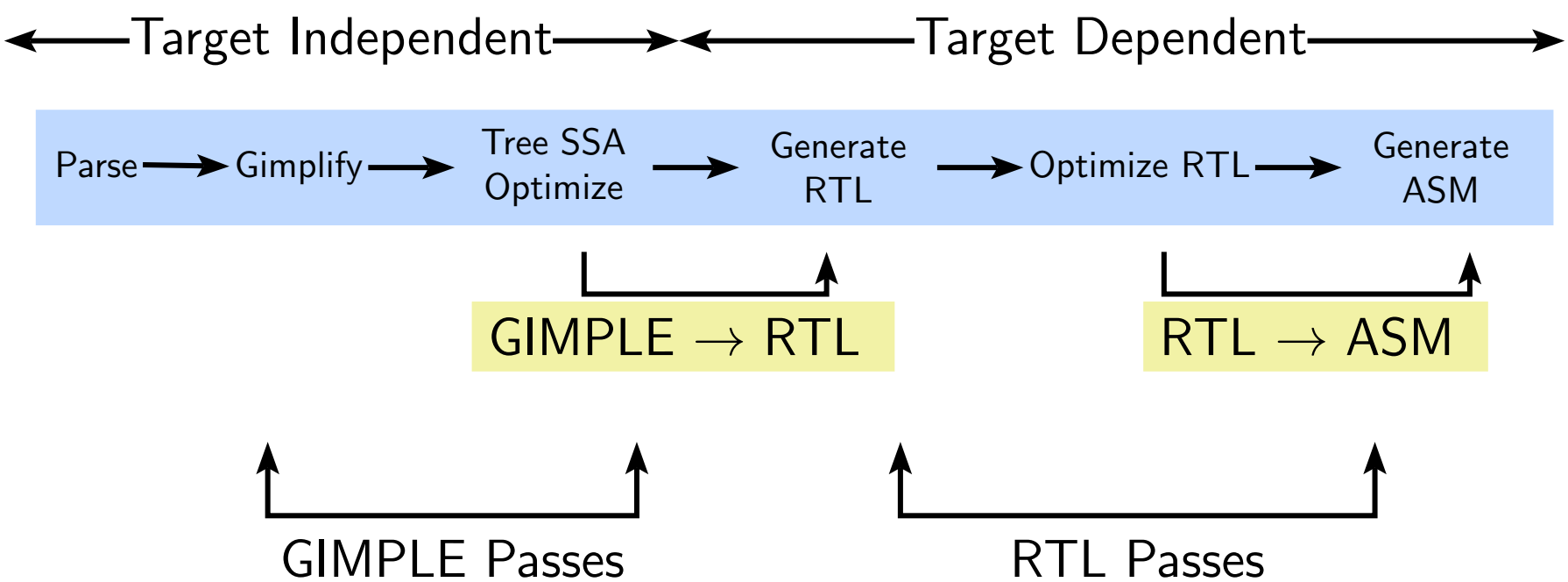
Basic Transformations in GCC

Tranformation from a language to a *different* language



Basic Transformations in GCC

Tranformation from a language to a *different* language



Transformation Passes in GCC 4.7.2

- A total of 215 unique pass names initialized in `${SOURCE}/gcc/passes.c`
  Total number of passes is 252.
  - Some passes are called multiple times in different contexts
    Conditional constant propagation is called thrice.
  - Some passes are enabled for specific architectures
  - Some passes have many variations

Pass Name	Optimization	Times
pass_cd_dce	Dead code elimination	2
pass_call_cdce	Dead call elimination	1
pass_dce	Dead code elimination	2
pass_dce_loop	Dead code elimination	3
pass_ud_rtl_dce	RTL dead code elimination	1
pass_fast_rtl_dce	RTL dead code elimination	1

- The pass sequence can be divided broadly in two parts
  - Passes on GIMPLE
  - Passes on RTL
- Some passes are organizational passes to group related passes

Passes On GIMPLE in GCC 4.7.2

Pass Group	Examples	Number of passes
Lowering	GIMPLE IR, CFG Construction	12
Simple Interprocedural Passes (Non-LTO)	Conditional Constant Propagation, Inlining, SSA Construction	40
Regular Interprocedural Passes (LTO)	Constant Propagation, Inlining	7
LTO generation passes		02
Late interprocedural passes (LTO)	Pointer Analysis	01
Other Intraprocedural Optimizations	Constant Propagation, Dead Code Elimination, PRE Value Range Propagation, Rename SSA	72
Loop Optimizations	Vectorization, Parallelization, Copy Propagation, Dead Code Elimination	28
Generating RTL		01
Total number of passes on GIMPLE		163

Passes On RTL in GCC 4.7.2

Pass Group	Examples	Number of passes
Intraprocedural Optimizations	CSE, Jump Optimization, Dead Code Elimination, Jump Optimization	27
Loop Optimizations	Loop Invariant Movement, Peeling, Unswitching	07
Machine Dependent Optimizations	Register Allocation, Instruction Scheduling, Peephole Optimizations	52
Assembly Emission and Finishing		03
Total number of passes on RTL		89

Finding Out List of Optimizations

Along with the associated flags

- A complete list of optimizations with a brief description

  ```
  gcc -c --help=optimizers
  ```

- Optimizations enabled at level 2 (other levels are 0, 1, 3, and s)

  ```
  gcc -c -O2 --help=optimizers -Q
  ```

Producing the Output of GCC Passes

- Use the option `-fdump-<ir>-<passname>`

  `<ir>` could be

  - `tree`: Intraprocedural passes on GIMPLE
  - `ipa`: Interprocedural passes on GIMPLE
  - `rtl`: Intraprocedural passes on RTL

- Use `all` in place of `<pass>` to see all dumps

  Example: `gcc -fdump-tree-all -fdump-rtl-all test.c`

- Dumping more details:

  Suffix `raw` for tree passes and `details` or `slim` for RTL passes

  Individual passes may have more verbosity options (e.g. `-fsched-verbose=5`)

- Use `-S` to stop the compilation with assembly generation

- Use `--verbose-asm` to see more detailed assembly dump

Total Number of Dumps

Dump Options: `-fdump-tree-all -fdump-ipa-all -fdump-rtl-all`

Optimization Level	Number of Dumps	Goals
Default	47	Fast compilation
O1	137	
O2	164	
O3	173	
Os	160	Optimize for space

Selected Dumps for Our Example Program

GIMPLE dumps (t)

001t.tu
003t.original
004t.gimple
006t.vcg
009t.omplower
010t.lower
013t.eh
014t.cfg
018t.ssa
020t.inline_param1
021t.einline
039t.release_ssa
040t.inline_param2
077t.cplxlower
137t.tailc
149t.optimized
232t.statistics

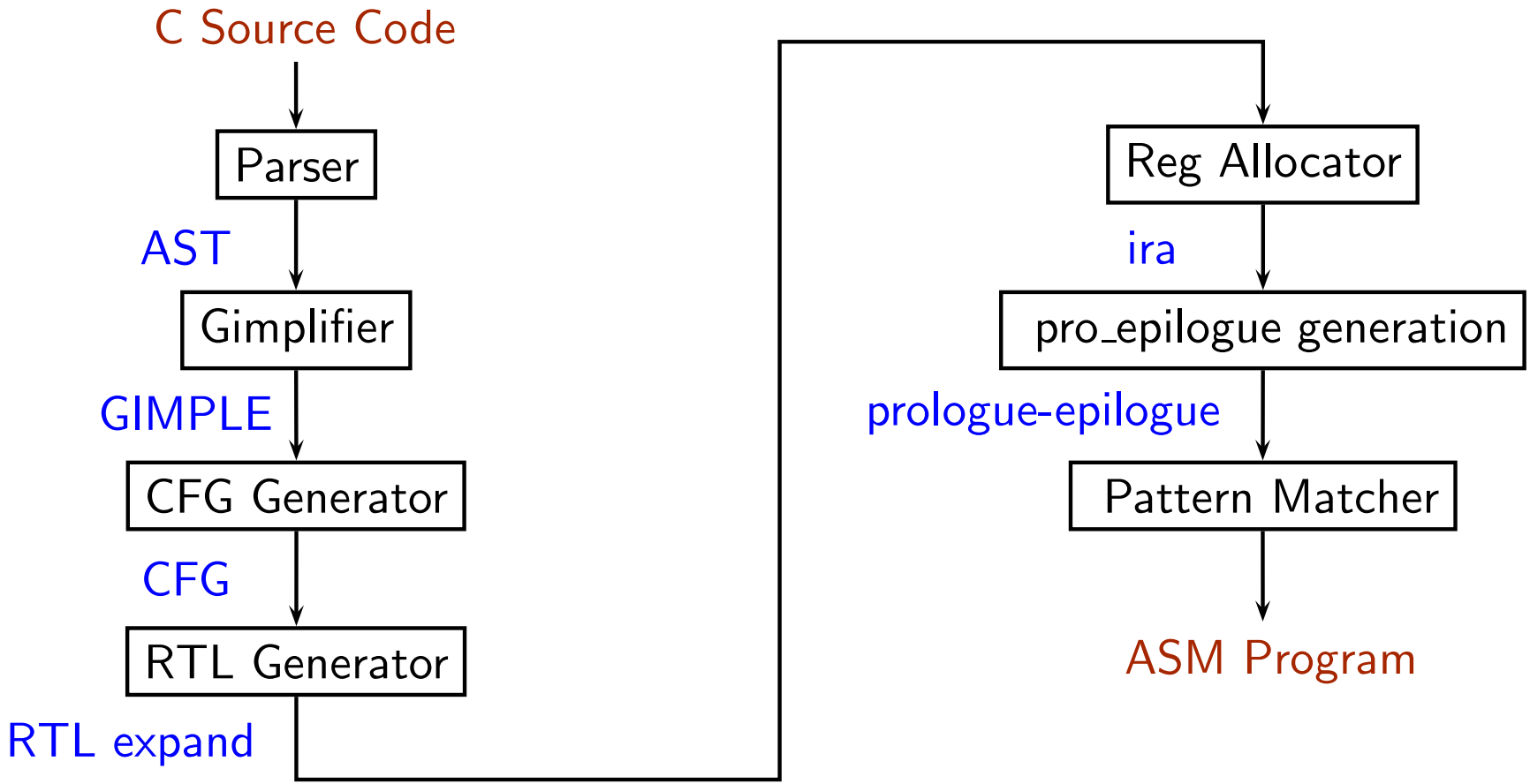
ipa dumps (i)

000i.cgraph
015i.visibility
016i.early_local_cleanups
047i.whole-program
048i.inline
054i.lto_gimple_out
055i.lto_decls_out

rtl dumps (r)

150r.expand
151r.sibling
153r.initvals
154r.unshare
155r.vregs
156r.into_cfglayout
157r.jump
158r.subreg1
159r.dfinit
163r.pre
169r.reginfo
189r.outof_cfglayout
190r.split1
193r.mode_sw
194r.asmcons
197r.ira
201r.split2
205r.pro_and_epilogue
218r.stack
219r.alignments
222r.mach
223r.barriers
227r.shorten
228r.nothrow
230r.final
231r.dfinish
assembly

Passes for First Level Graybox Probing of GCC



Lowering of abstraction!

Gimplifier

- About GIMPLE
  - Three-address representation derived from GENERIC
    Computation represented as a sequence of basic operations
    Temporaries introduced to hold intermediate values
  - Control construct are explicated into conditional jumps
- Examining GIMPLE Dumps
  - Examining translation of data accesses
  - Examining translation of control flow
  - Examining translation of function calls

GIMPLE: Composite Expressions Involving Scalar Variables

test.c

```
int a;

int main()
{
  int x = 10;
  int y = 5;

  x = a + x * y;
  y = y - a * x;

}
```

test.c.004t.gimple

```
  x = 10;
  y = 5;
  D.1954 = x * y;
  a.0 = a;
  x = D.1954 + a.0;
  a.1 = a;
  D.1957 = a.1 * x;
  y = y - D.1957;
```

Global variables are treated as "memory locations" and local variables are treated as "registers"

GIMPLE: Composite Expressions Involving Scalar Variables

test.c

```
int a;

int main()
{
  int x = 10;
  int y = 5;

  x = a + x * y;
  y = y - a * x;

}
```

test.c.004t.gimple

```
  x = 10;
  y = 5;
  D.1954 = x * y;
  a.0 = a;
  x = D.1954 + a.0;
  a.1 = a;
  D.1957 = a.1 * x;
  y = y - D.1957;
```

Global variables are treated as "memory locations" and local variables are treated as "registers"

GIMPLE: Composite Expressions Involving Scalar Variables

test.c

```
int a;

int main()
{
  int x = 10;
  int y = 5;

  x = a + x * y;
  y = y - a * x;

}
```

test.c.004t.gimple

```
  x = 10;
  y = 5;
  D.1954 = x * y;
  a.0 = a;
  x = D.1954 + a.0;
  a.1 = a;
  D.1957 = a.1 * x;
  y = y - D.1957;
```

Global variables are treated as "memory locations" and local variables are treated as "registers"

GIMPLE: Composite Expressions Involving Scalar Variables

test.c

```
int a;

int main()
{
  int x = 10;
  int y = 5;

  x = a + x * y;
  y = y - a * x;

}
```

test.c.004t.gimple

```
  x = 10;
  y = 5;
  D.1954 = x * y;
  a.0 = a;
  x = D.1954 + a.0;
  a.1 = a;
  D.1957 = a.1 * x;
  y = y - D.1957;
```

Global variables are treated as "memory locations" and local variables are treated as "registers"

GIMPLE: 1-D Array Accesses

test.c

```
int main()
{
  int a[3], x;
  a[1] = a[2] = 10;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

test.c.004t.gimple

```
try {
    a[2] = 10;
    D.1952 = a[2];
    a[1] = D.1952;
    D.1953 = a[1];
    D.1954 = a[2];
    x = D.1953 + D.1954;
    D.1955 = x + 1;
    D.1956 = a[1];
    D.1957 = D.1955 * D.1956;
    a[0] = D.1957;
}
finally {
    a = {CLOBBER};
}
```

GIMPLE: 1-D Array Accesses

test.c

```
int main()
{
  int a[3], x;
  a[1] = a[2] = 10;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

test.c.004t.gimple

```
try {
    a[2] = 10;
    D.1952 = a[2];
    a[1] = D.1952;
    D.1953 = a[1];
    D.1954 = a[2];
    x = D.1953 + D.1954;
    D.1955 = x + 1;
    D.1956 = a[1];
    D.1957 = D.1955 * D.1956;
    a[0] = D.1957;
}
finally {
    a = {CLOBBER};
}
```

GIMPLE: 1-D Array Accesses

test.c

```
int main()
{
  int a[3], x;
  a[1] = a[2] = 10;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

test.c.004t.gimple

```
try {
    a[2] = 10;
    D.1952 = a[2];
    a[1] = D.1952;
    D.1953 = a[1];
    D.1954 = a[2];
    x = D.1953 + D.1954;
    D.1955 = x + 1;
    D.1956 = a[1];
    D.1957 = D.1955 * D.1956;
    a[0] = D.1957;
}
finally {
    a = {CLOBBER};
}
```

GIMPLE: 1-D Array Accesses

test.c

```
int main()
{
  int a[3], x;
  a[1] = a[2] = 10;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

test.c.004t.gimple

```
try {
    a[2] = 10;
    D.1952 = a[2];
    a[1] = D.1952;
    D.1953 = a[1];
    D.1954 = a[2];
    x = D.1953 + D.1954;
    D.1955 = x + 1;
    D.1956 = a[1];
    D.1957 = D.1955 * D.1956;
    a[0] = D.1957;
}
finally {
    a = {CLOBBER};
}
```

GIMPLE: 1-D Array Accesses

test.c

```
int main()
{
  int a[3], x;
  a[1] = a[2] = 10;
  x = a[1] + a[2];
  a[0] = a[1] + a[1]*x;
}
```

test.c.004t.gimple

```
try {
    a[2] = 10;
    D.1952 = a[2];
    a[1] = D.1952;
    D.1953 = a[1];
    D.1954 = a[2];
    x = D.1953 + D.1954;
    D.1955 = x + 1;
    D.1956 = a[1];
    D.1957 = D.1955 * D.1956;
    a[0] = D.1957;
}
finally {
    a = {CLOBBER};
}
```

GIMPLE: 2-D Array Accesses

test.c

```
int main()
{
  int a[3][3], x, y;
  a[0][0] = 7;
  a[1][1] = 8;
  a[2][2] = 9;
  x = a[0][0] / a[1][1];
  y = a[1][1] % a[2][2];
}
```

test.c.004t.gimple

```
try {
      a[0][0] = 7;
      a[1][1] = 8;
      a[2][2] = 9;
      D.1953 = a[0][0];
      D.1954 = a[1][1];
      x = D.1953 / D.1954;
      D.1955 = a[1][1];
      D.1956 = a[2][2];
      y = D.1955 % D.1956;
}
finally {
      a = {CLOBBER};
}
```


GIMPLE: Use of Pointers

test.c

```
int main()
{
  int **a,*b,c;
  b = &c;
  a = &b;
  **a = 10;  /* c = 10 */
}
```

test.c.004t.gimple

```
main () {
    int * D.1953;
    int * * a;
    int * b;
    int c;
    try
    {
      b = &c;
      a = &b;
      D.1953 = *a;
      *D.1953 = 10;
    }
    finally {
        b = {CLOBBER};
        c = {CLOBBER};
    }
}
```

GIMPLE: Use of Pointers

test.c

```
int main()
{
  int **a,*b,c;
  b = &c;
  a = &b;
  **a = 10;  /* c = 10 */
}
```

test.c.004t.gimple

```
main () {
    int * D.1953;
    int * * a;
    int * b;
    int c;
    try
    {
      b = &c;
      a = &b;
      D.1953 = *a;
      *D.1953 = 10;
    }
    finally {
        b = {CLOBBER};
        c = {CLOBBER};
    }
}
```

Memory and Registers in GIMPLE

- Memory: Gobals, address taken variables, arrays
  - Scalar memory values must be explicitly loaded into registers

    ```
    a.0 = a;
    ```

  - No "addressable" memory within arrays
    - No base + offset modelling of arrays
    - Array reference is a single operation in GIMPLE
  - Since "memory" survives the lifetime of a given scope, locals are marked as clobbered at the end of the scope
- Registers: Locals, formals
  - Restricted visibility
  - Cannot be modified by function calls or a concurrent process
  - Can be freely rearranged

GIMPLE: Use of Structures

test.c

```
typedef struct address
{ char *name;
} ad;

typedef struct student
{ int roll;
  ad *ct;
} st;

int main()
{ st *s;
  s = malloc(sizeof(st));
  s->roll = 1;
  s->ct=malloc(sizeof(ad));
  s->ct->name = "Mumbai";
}
```

test.c.004t.gimple

```
main ()
{
  void * D.1957;
  struct ad * D.1958;
  struct st * s;
  extern void * malloc (unsigned int);

  s = malloc (8);
  s->roll = 1;
  D.1957 = malloc (4);
  s->ct = D.1957;
  D.1958 = s->ct;
  D.1958->name = "Mumbai";
}
```


GIMPLE: Use of Structures

test.c

```
typedef struct address
{ char *name;
} ad;

typedef struct student
{ int roll;
  ad *ct;
} st;

int main()
{ st *s;
  s = malloc(sizeof(st));
  s->roll = 1;
  s->ct=malloc(sizeof(ad));
  s->ct->name = "Mumbai";
}
```

test.c.004t.gimple

```
main ()
{
  void * D.1957;
  struct ad * D.1958;
  struct st * s;
  extern void * malloc (unsigned int);

  s = malloc (8);
  s->roll = 1;
  D.1957 = malloc (4);
  s->ct = D.1957;
  D.1958 = s->ct;
  D.1958->name = "Mumbai";
}
```


GIMPLE: Use of Structures

test.c

```
typedef struct address
{ char *name;
} ad;

typedef struct student
{ int roll;
  ad *ct;
} st;

int main()
{ st *s;
  s = malloc(sizeof(st));
  s->roll = 1;
  s->ct=malloc(sizeof(ad));
  s->ct->name = "Mumbai";
}
```

test.c.004t.gimple

```
main ()
{
  void * D.1957;
  struct ad * D.1958;
  struct st * s;
  extern void * malloc (unsigned int);

  s = malloc (8);
  s->roll = 1;
  D.1957 = malloc (4);
  s->ct = D.1957;
  D.1958 = s->ct;
  D.1958->name = "Mumbai";
}
```

GIMPLE: Use of Structures

test.c

```
typedef struct address
{ char *name;
} ad;

typedef struct student
{ int roll;
  ad *ct;
} st;

int main()
{ st *s;
  s = malloc(sizeof(st));
  s->roll = 1;
  s->ct=malloc(sizeof(ad));
  s->ct->name = "Mumbai";
}
```

test.c.004t.gimple

```
main ()
{
  void * D.1957;
  struct ad * D.1958;
  struct st * s;
  extern void * malloc (unsigned int);

  s = malloc (8);
  s->roll = 1;
  D.1957 = malloc (4);
  s->ct = D.1957;
  D.1958 = s->ct;
  D.1958->name = "Mumbai";
}
```


GIMPLE: Pointer to Array

test.c

```
int main()
{
  int *p_a, a[3];

  p_a = &a[0];

  *p_a = 10;
  *(p_a+1) = 20;
  *(p_a+2) = 30;
}
```

test.c.004t.gimple

```
main ()
{
  int * D.2048;
  int * D.2049;
  int * p_a;
  int a[3];
  try {
      p_a = &a[0];
      *p_a = 10;
      D.2048 = p_a + 4;
      *D.2048 = 20;
      D.2049 = p_a + 8;
      *D.2049 = 30;
  }
  finally {
      a = {CLOBBER};
  }
}
```

GIMPLE: Pointer to Array

test.c

```
int main()
{
  int *p_a, a[3];

  p_a = &a[0];

  *p_a = 10;
  *(p_a+1) = 20;
  *(p_a+2) = 30;
}
```

test.c.004t.gimple

```
main ()
{
  int * D.2048;
  int * D.2049;
  int * p_a;
  int a[3];
  try {
      p_a = &a[0];
      *p_a = 10;
      D.2048 = p_a + 4;
      *D.2048 = 20;
      D.2049 = p_a + 8;
      *D.2049 = 30;
  }
  finally {
      a = {CLOBBER};
  }
}
```

GIMPLE: Pointer to Array

test.c

```
int main()
{
  int *p_a, a[3];

  p_a = &a[0];

  *p_a = 10;
  *(p_a+1) = 20;
  *(p_a+2) = 30;
}
```

test.c.004t.gimple

```
main ()
{
  int * D.2048;
  int * D.2049;
  int * p_a;
  int a[3];
  try {
      p_a = &a[0];
      *p_a = 10;
      D.2048 = p_a + 4;
      *D.2048 = 20;
      D.2049 = p_a + 8;
      *D.2049 = 30;
  }
  finally {
      a = {CLOBBER};
  }
}
```


GIMPLE: Pointer to Array

test.c

```
int main()
{
  int *p_a, a[3];

  p_a = &a[0];

  *p_a = 10;
  *(p_a+1) = 20;
  *(p_a+2) = 30;
}
```

test.c.004t.gimple

```
main ()
{
  int * D.2048;
  int * D.2049;
  int * p_a;
  int a[3];
  try {
      p_a = &a[0];
      *p_a = 10;
      D.2048 = p_a + 4;
      *D.2048 = 20;
      D.2049 = p_a + 8;
      *D.2049 = 30;
  }
  finally {
      a = {CLOBBER};
  }
}
```

GIMPLE: Translation of Conditional Statements

test.c

```
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
     if (a<=12)
          a = a+b+c;

}
```

test.c.004t.gimple

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

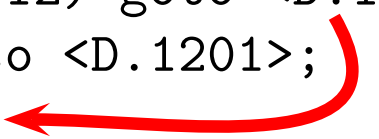

GIMPLE: Translation of Conditional Statements

test.c

```
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
     if (a<=12)
          a = a+b+c;

}
```

test.c.004t.gimple

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```



GIMPLE: Translation of Conditional Statements

test.c

```
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
     if (a<=12)
          a = a+b+c;

}
```

test.c.004t.gimple

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

GIMPLE: Translation of Loops

test.c

```
int main()
{
    int a=2, b=3, c=4;
     while (a<=7)
     {
          a = a+1;
     }
    if (a<=12)
        a = a+b+c;
}
```

test.c.004t.gimple

```
goto <D.1197>;
<D.1196>:
a = a + 1;
<D.1197>:
if (a <= 7) goto <D.1196>;
else goto <D.1198>;
<D.1198>:
```

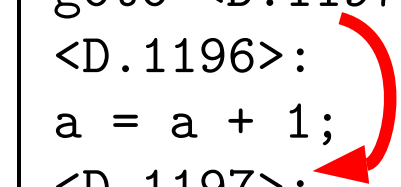
GIMPLE: Translation of Loops

test.c

```
int main()
{
    int a=2, b=3, c=4;
     while (a<=7)
     {
          a = a+1;
     }
    if (a<=12)
        a = a+b+c;
}
```

test.c.004t.gimple

```
goto <D.1197>;
<D.1196>:
a = a + 1;
<D.1197>:
if (a <= 7) goto <D.1196>;
else goto <D.1198>;
<D.1198>:
```



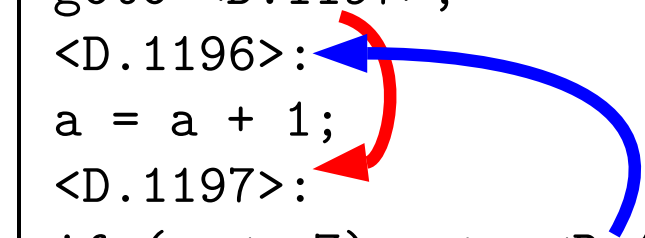
GIMPLE: Translation of Loops

test.c

```
int main()
{
    int a=2, b=3, c=4;
     while (a<=7)
     {
          a = a+1;
     }
    if (a<=12)
        a = a+b+c;
}
```

test.c.004t.gimple

```
goto <D.1197>;
<D.1196>:
a = a + 1;
<D.1197>:
if (a <= 7) goto <D.1196>;
else goto <D.1198>;
<D.1198>:
```



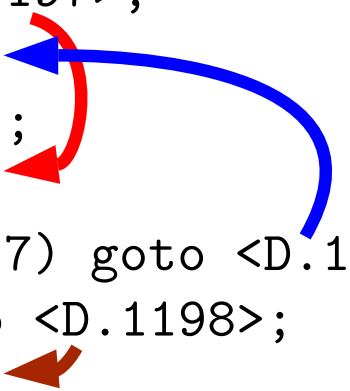
GIMPLE: Translation of Loops

test.c

```
int main()
{
    int a=2, b=3, c=4;
     while (a<=7)
     {
          a = a+1;
     }
    if (a<=12)
        a = a+b+c;
}
```

test.c.004t.gimple

```
goto <D.1197>;
<D.1196>:
a = a + 1;
<D.1197>:
if (a <= 7) goto <D.1196>;
else goto <D.1198>;
<D.1198>:
```



Control Flow Graph: Textual View

test.c.004t.gimple

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

test.c.014t.cfg

```
<bb 5>:
  if (a <= 12)
    goto <bb 6>;
  else
    goto <bb 7>;

<bb 6>:
  D.1199 = a + b;
  a = D.1199 + c;

<bb 7>:
  return;
```

Control Flow Graph: Textual View

test.c.004t.gimple

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

test.c.014t.cfg

```
<bb 5>:
  if (a <= 12)
    goto <bb 6>;
  else
    goto <bb 7>;

<bb 6>:
  D.1199 = a + b;
  a = D.1199 + c;

<bb 7>:
  return;
```


Control Flow Graph: Textual View

test.c.004t.gimple

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

test.c.014t.cfg

```
<bb 5>:
  if (a <= 12)
    goto <bb 6>;
  else
    goto <bb 7>;

<bb 6>:
  D.1199 = a + b;
  a = D.1199 + c;

<bb 7>:
  return;
```


Control Flow Graph: Textual View

test.c.004t.gimple

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

test.c.014t.cfg

```
<bb 5>:
  if (a <= 12)
    goto <bb 6>;
  else
    goto <bb 7>;

<bb 6>:
  D.1199 = a + b;
  a = D.1199 + c;

<bb 7>:
  return;
```


Control Flow Graph: Textual View

test.c.004t.gimple

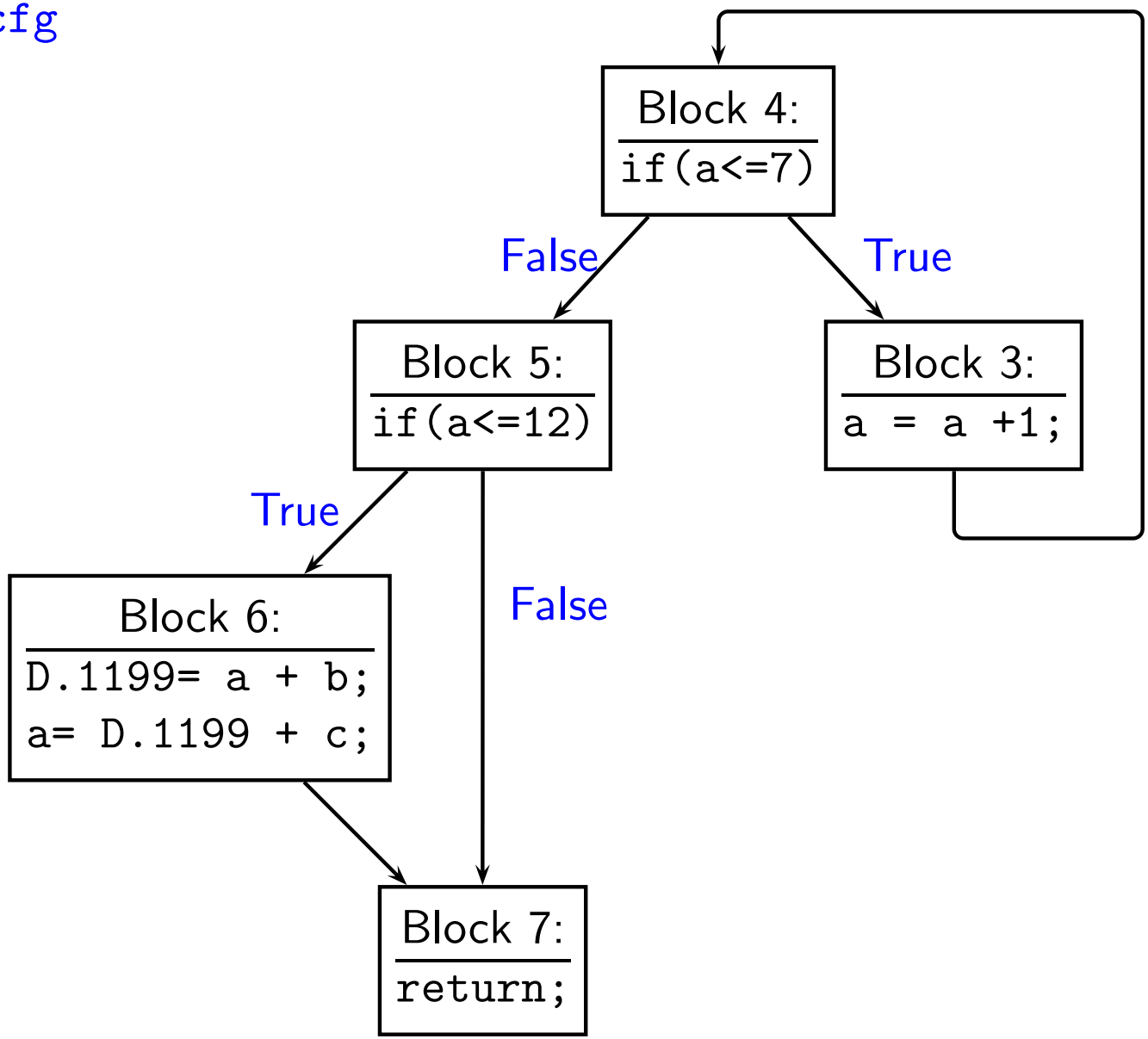
```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

test.c.014t.cfg

```
<bb 5>:
  if (a <= 12)
    goto <bb 6>;
  else
    goto <bb 7>;

<bb 6>:
  D.1199 = a + b;
  a = D.1199 + c;

<bb 7>:
  return;
```

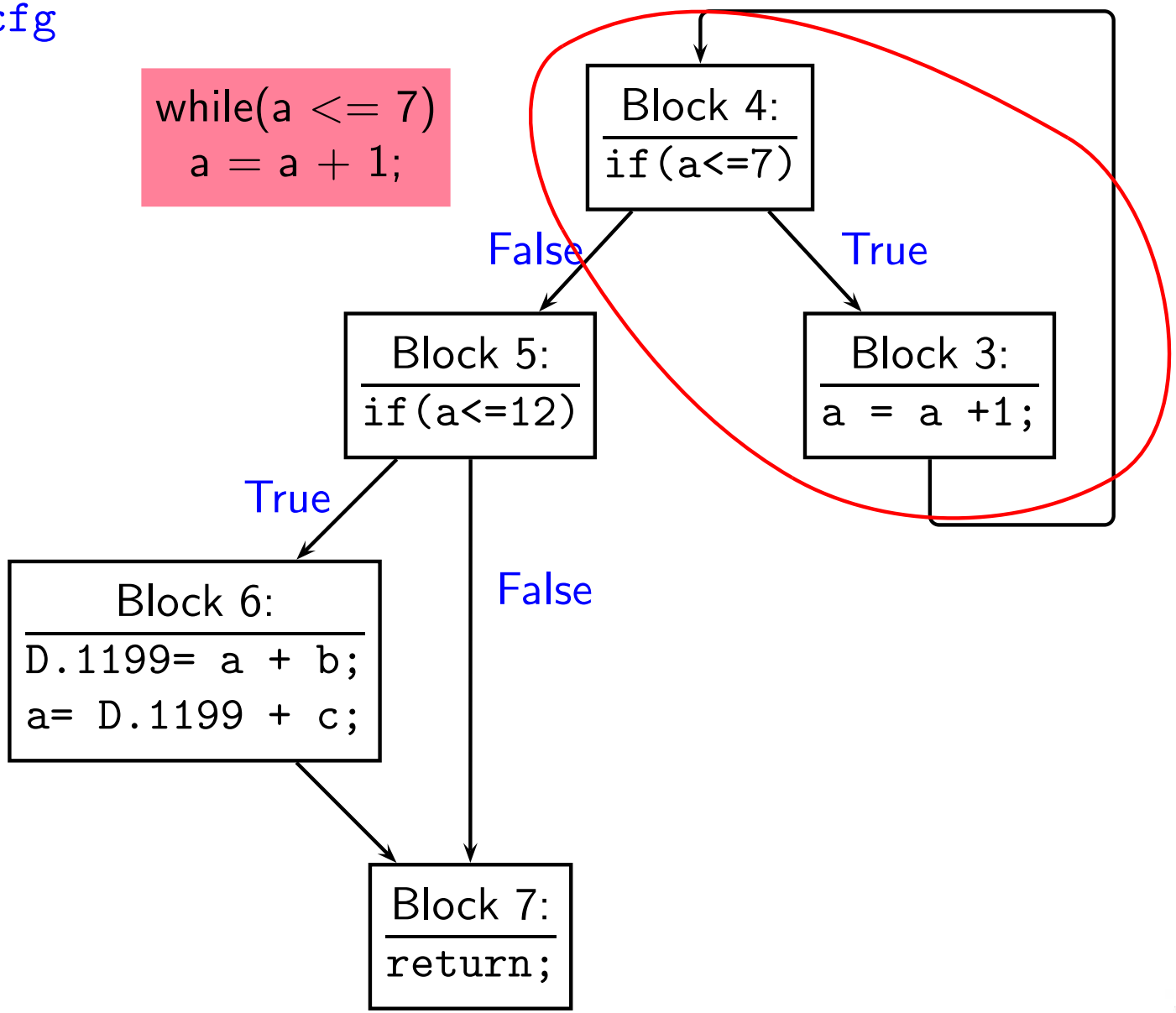

Control Flow Graph: Pictorial View

test.c.014t.cfg



Control Flow Graph: Pictorial View

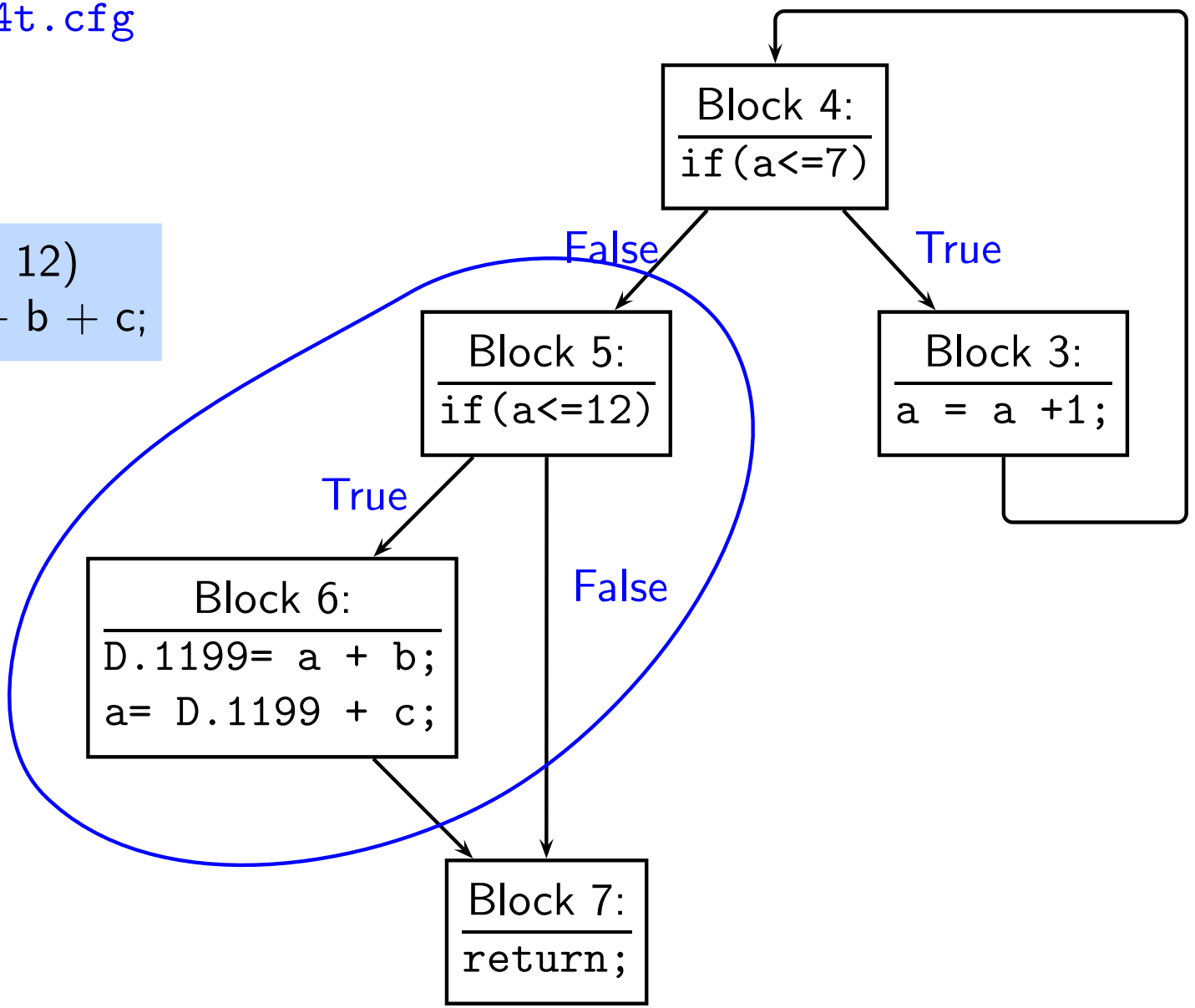
test.c.014t.cfg



Control Flow Graph: Pictorial View

test.c.014t.cfg

```
if(a <= 12)
a = a + b + c;
```



GIMPLE: Function Calls and Call Graph

test.c

```
extern int divide(int, int);
int multiply(int a, int b)
{
   return a*b;
}

int main()
{ int x,y;
  x = divide(20,5);
  y = multiply(x,2);
  printf("%d\n", y);
}
```

test.c.000i.cgraph

```
printf/3  @0x7fd094bbba20 availabilit
  called by: main/1  (1.00 per call)
  calls:
divide/2  @0x7fd094bbb900 availabilit
  called by: main/1  (1.00 per call)
  calls:
main/1  @0x7fd094bbb7e0 (asm: main) a
  called by:
  calls: printf/3  (1.00 per call)
         multiply/0 (1.00 per call)
         divide/2 (1.00 per call)
multiply/0  @0x7fd094bbb6c0 (asm: mul
  called by: main/1  (1.00 per call)
  calls:
```

GIMPLE: Function Calls and Call Graph

test.c

```
extern int divide(int, int);
int multiply(int a, int b)
{
   return a*b;
}

int main()
{ int x,y;
  x = divide(20,5);
  y = multiply(x,2);
  printf("%d\n", y);
}
```

test.c.000i.cgraph

```
printf/3  @0x7fd094bbba20 availabilit
  called by: main/1  (1.00 per call)
  calls:
divide/2  @0x7fd094bbb900 availabilit
  called by: main/1  (1.00 per call)
  calls:
main/1  @0x7fd094bbb7e0 (asm: main) a
  called by:
  calls: printf/3  (1.00 per call)
         multiply/0 (1.00 per call)
         divide/2 (1.00 per call)
multiply/0  @0x7fd094bbb6c0 (asm: mul
  called by: main/1  (1.00 per call)
  calls:
```

GIMPLE: Function Calls and Call Graph

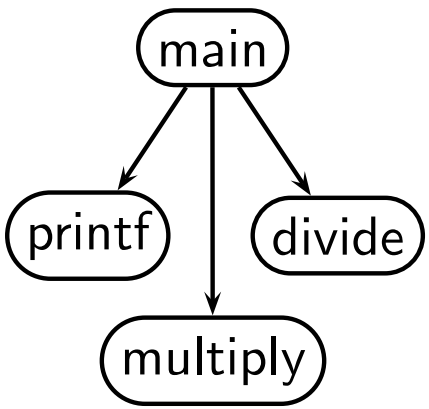
test.c

```
extern int divide(int, int);
int multiply(int a, int b)
{
   return a*b;
}

int main()
{ int x,y;
  x = divide(20,5);
  y = multiply(x,2);
  printf("%d\n", y);
}
```

test.c.000i.cgraph

```
printf/3
  called by: main/1
  calls:
divide/2
  called by: main/1
  calls:
main/1
  called by:
  calls: printf/3
         multiply/0
         divide/2
multiply/0
  called by: main/1
  calls:
```

call graph



GIMPLE: Function Calls and Call Graph

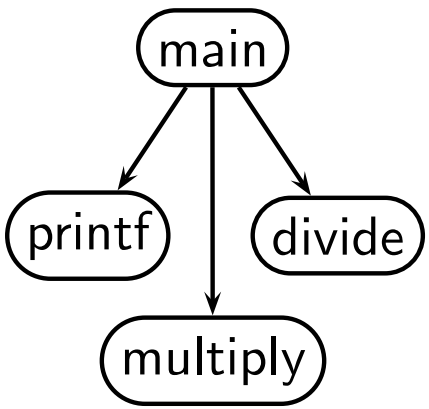
test.c

```
extern int divide(int, int);
int multiply(int a, int b)
{
   return a*b;
}

int main()
{ int x,y;
  x = divide(20,5);
  y = multiply(x,2);
  printf("%d\n", y);
}
```

test.c.000i.cgraph

```
printf/3
  called by: main/1
  calls:
divide/2
  called by: main/1
  calls:
main/1
  called by:
  calls: printf/3
         multiply/0
         divide/2
multiply/0
  called by: main/1
  calls:
```

call graph

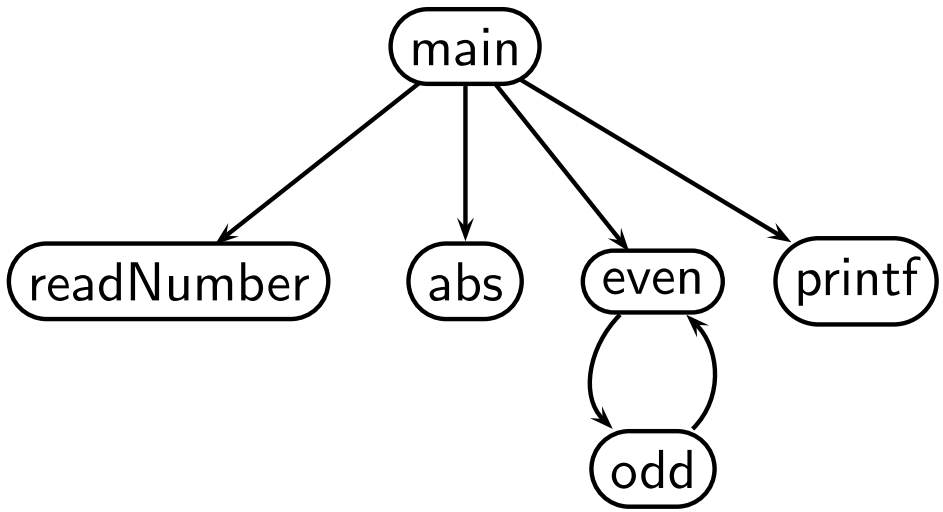


GIMPLE: Call Graphs for Recursive Functions

test.c

```
int even(int n)
{  if (n == 0) return 1;
   else return (!odd(n-1));
}

int odd(int n)
{  if (n == 1) return 1;
   else return (!even(n-1));
}

main()
{ int n;

  n = abs(readNumber());
  if (even(n))
    printf ("n is even\n");
  else printf ("n is odd\n");
}
```

call graph



Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

$x = 10$, $y = 5$

Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

$x = 10, y = 5$

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```

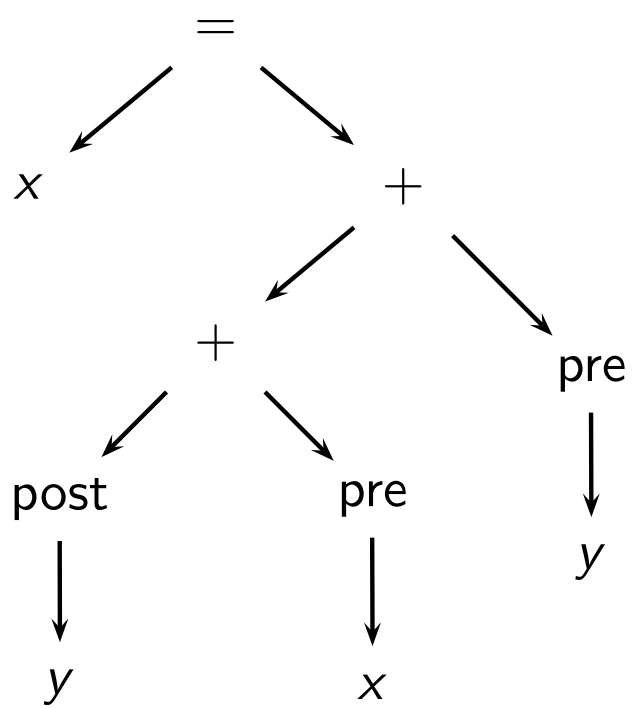

Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

$x = 10, y = 5$

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



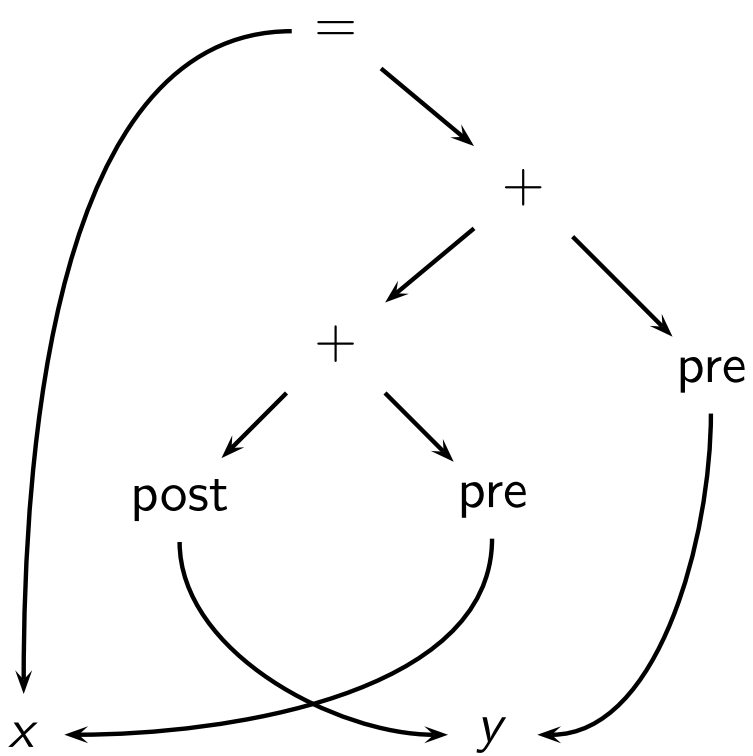
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

$x = 10, y = 5$

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



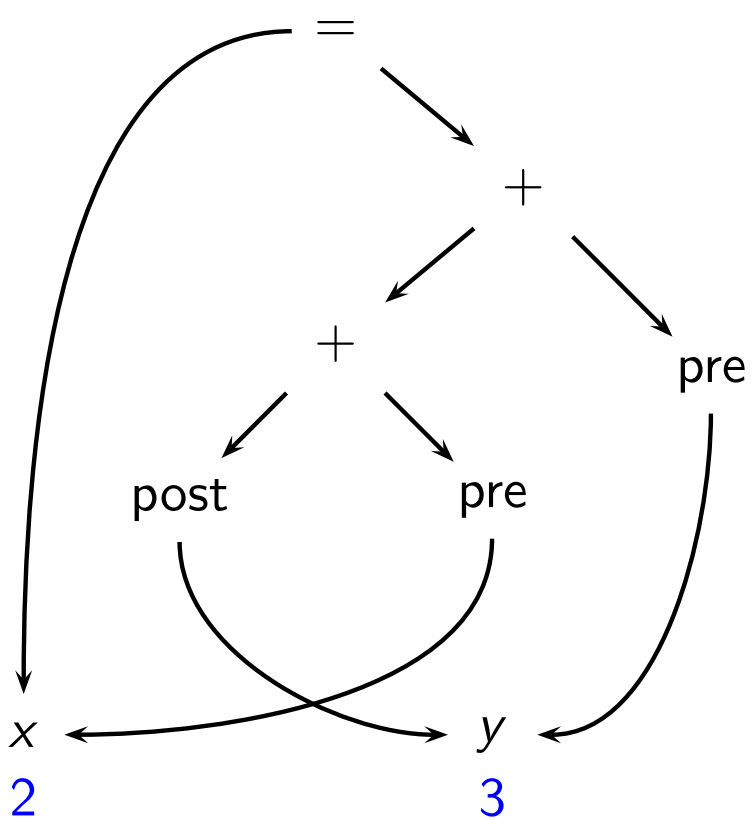
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



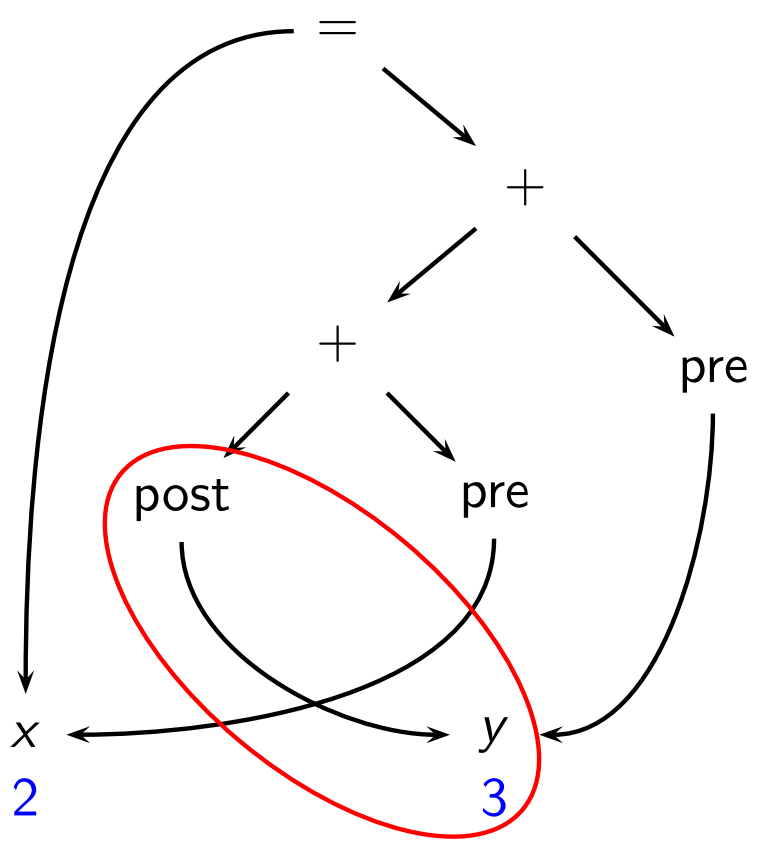
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



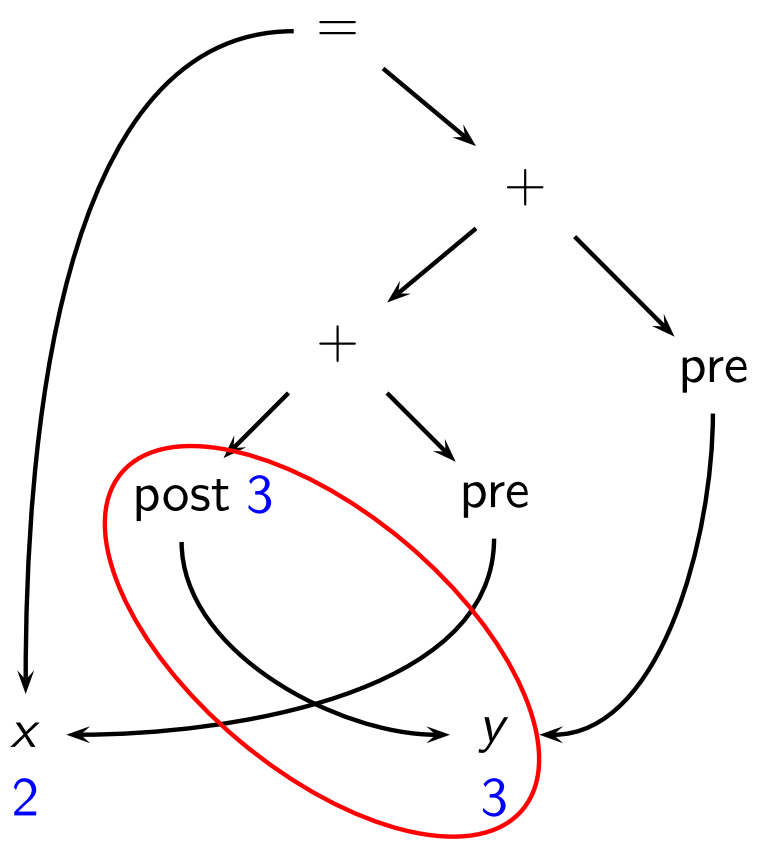
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



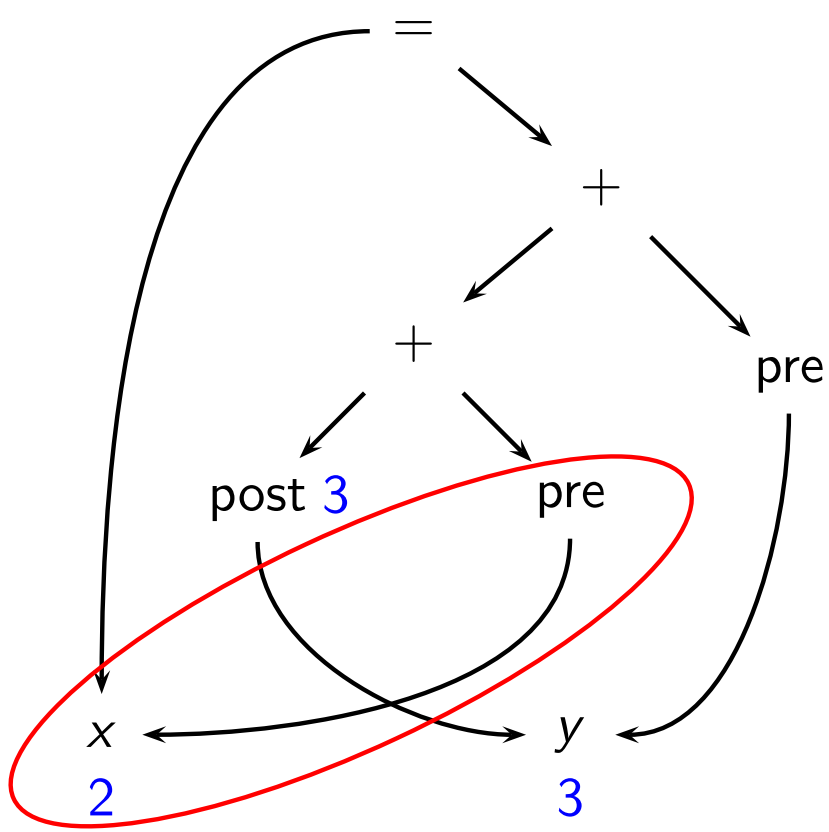
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



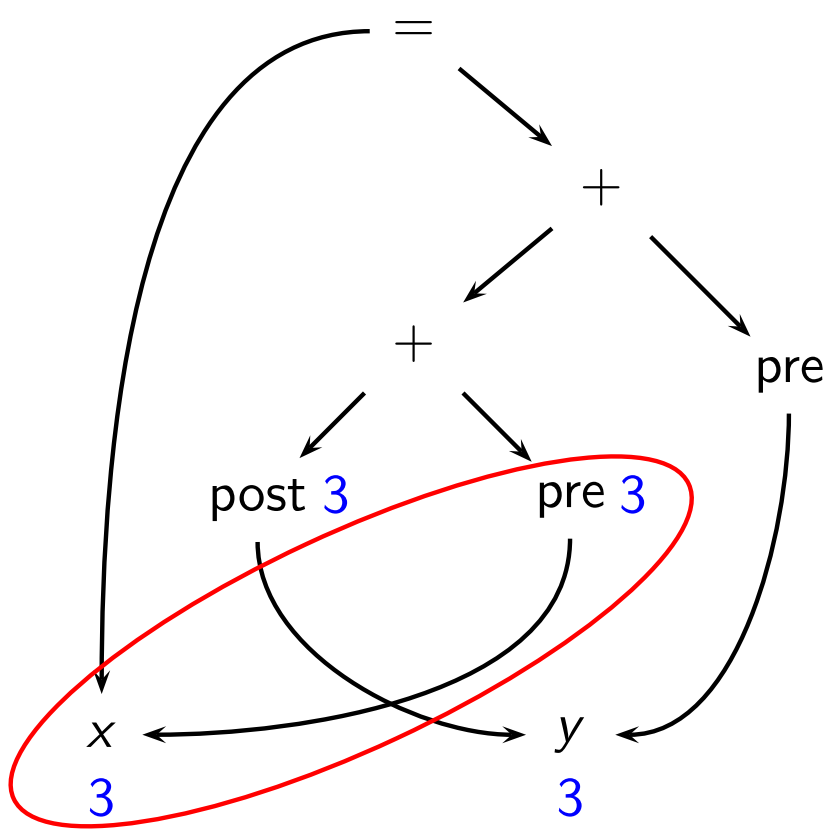
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



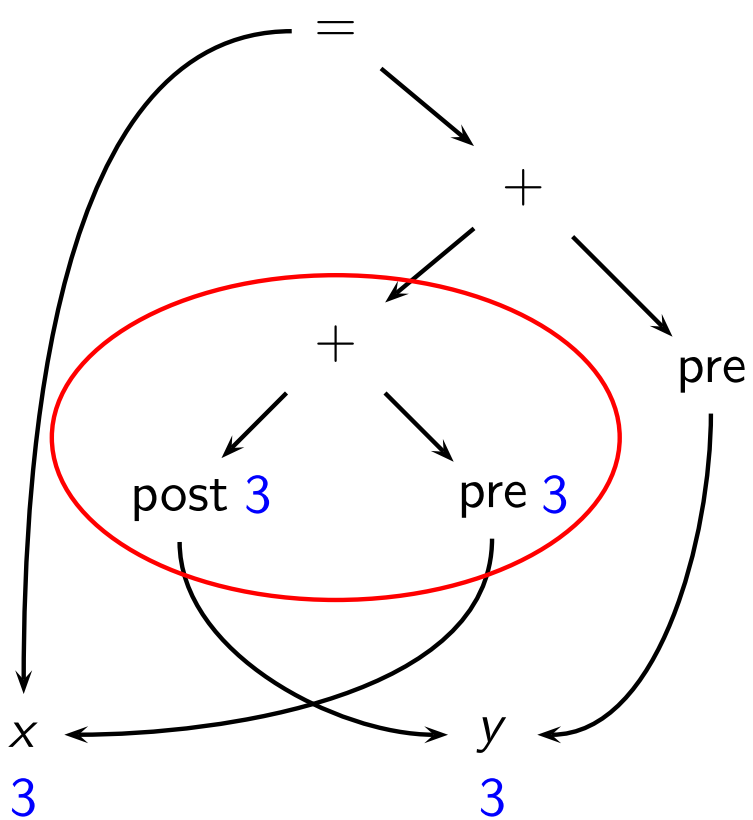
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



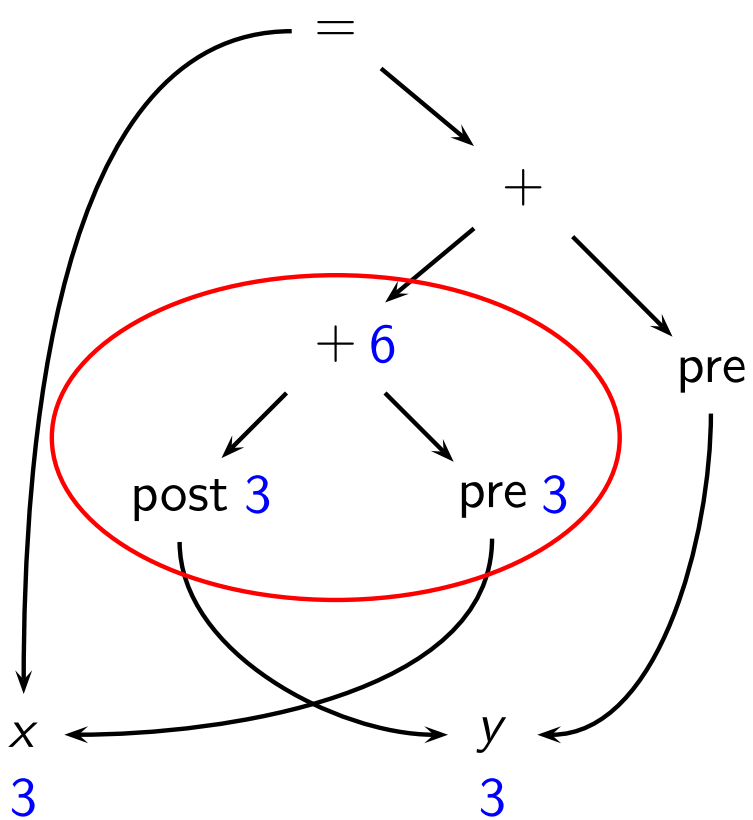
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



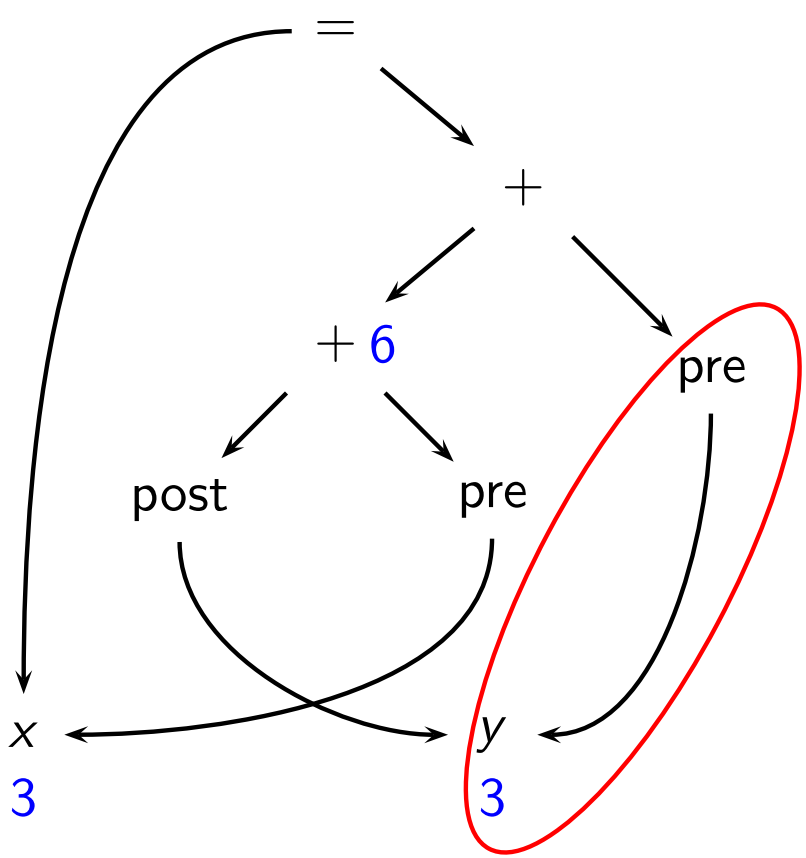
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



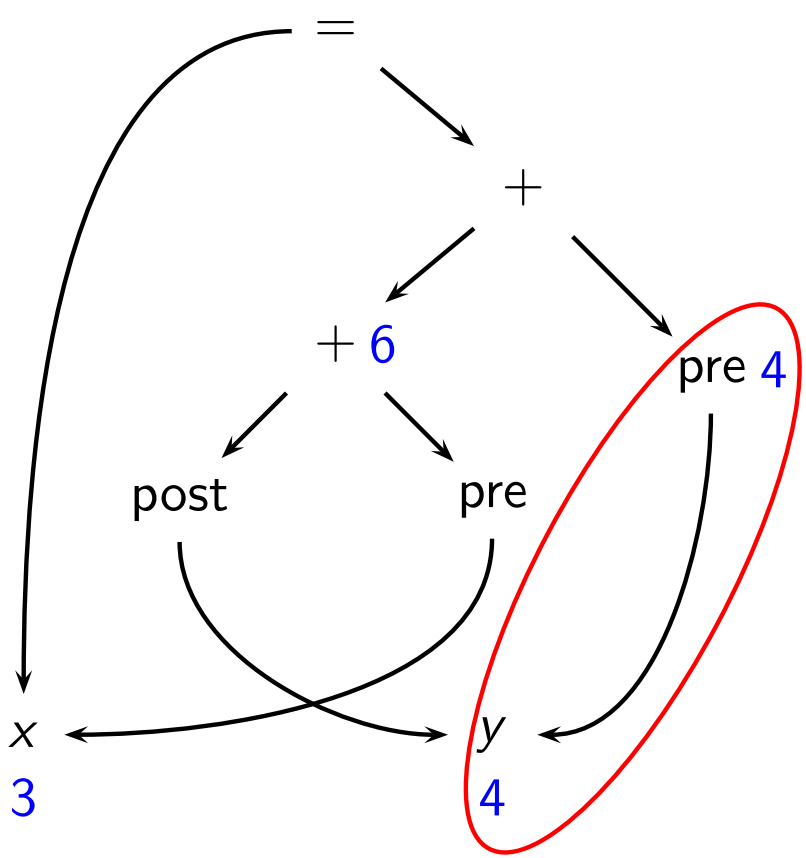
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



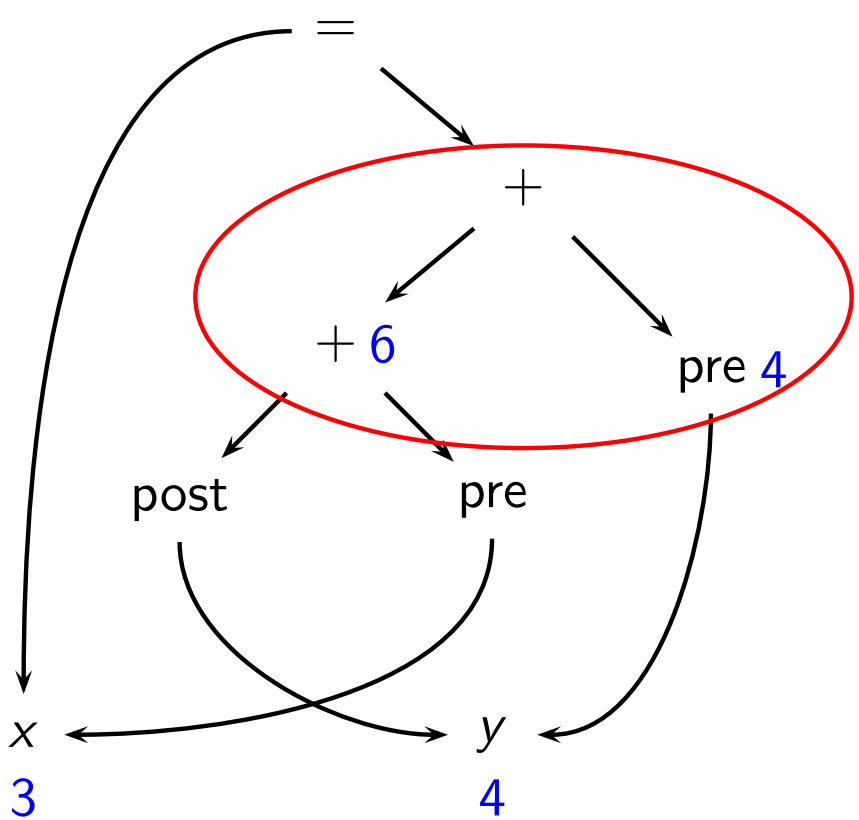
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



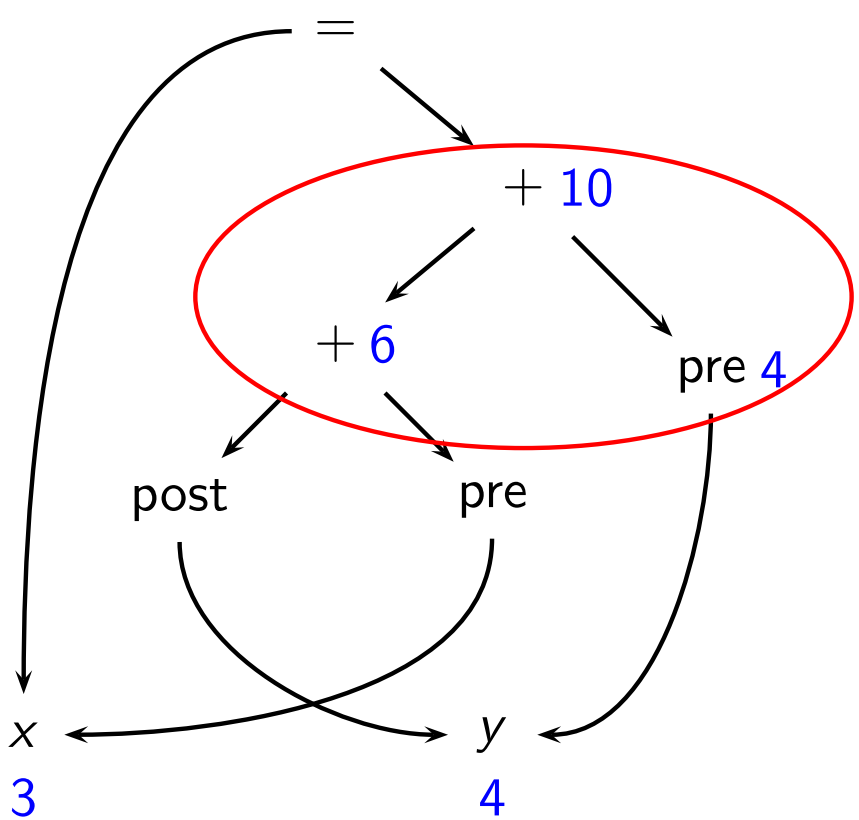
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



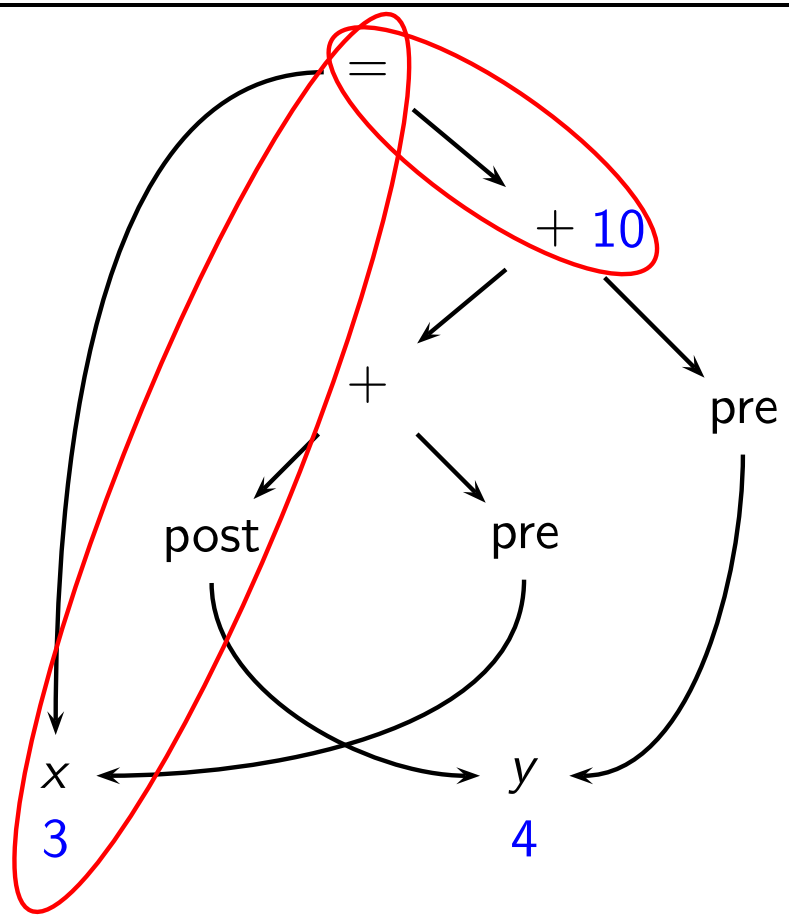
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



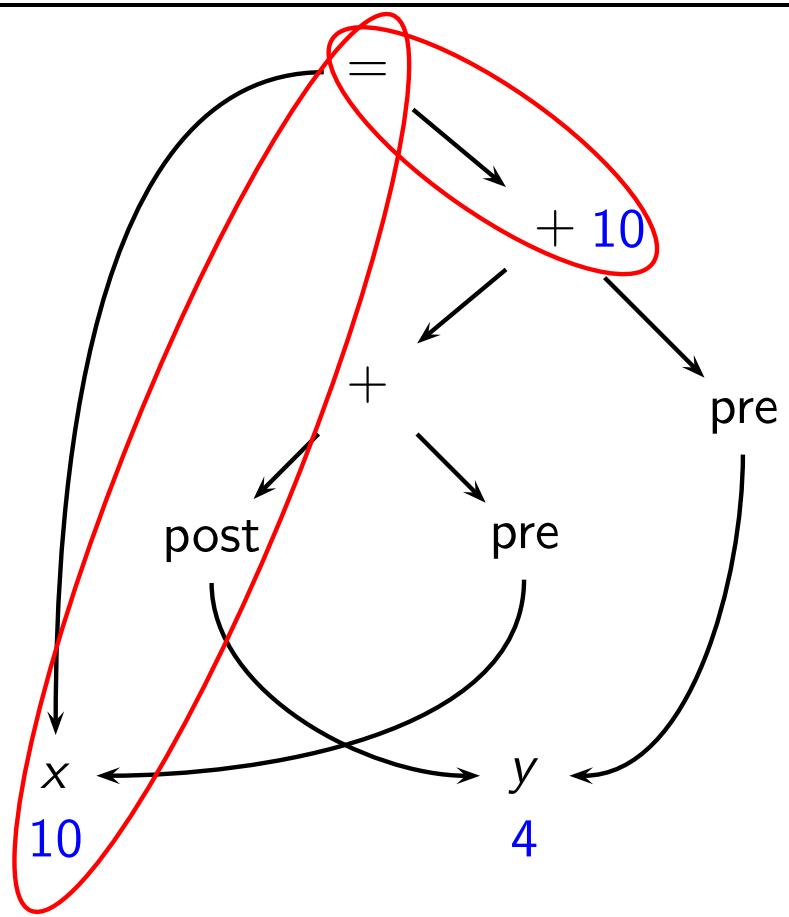
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



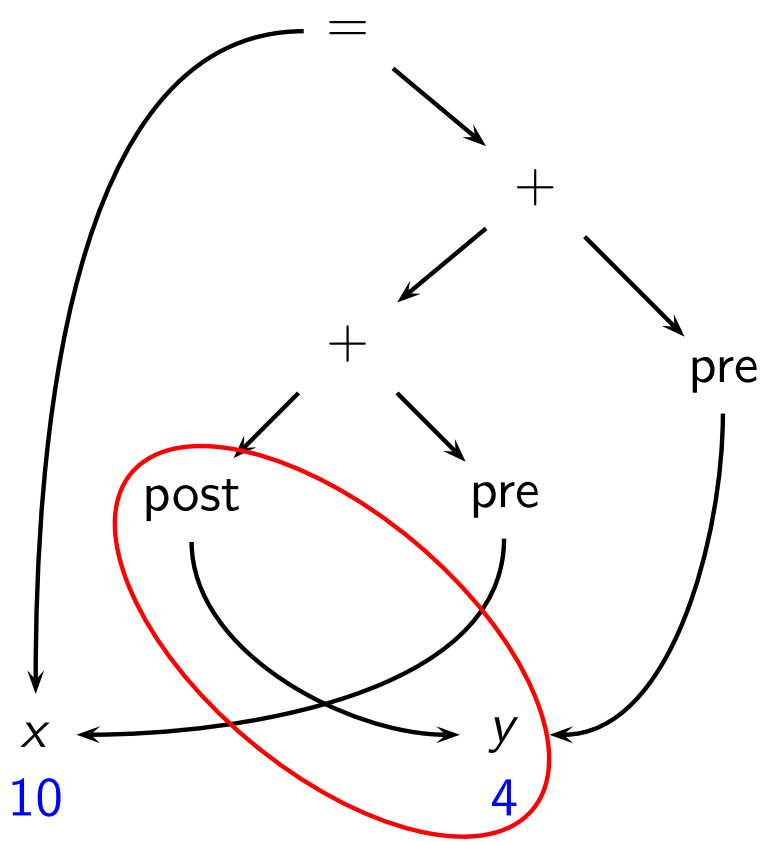
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



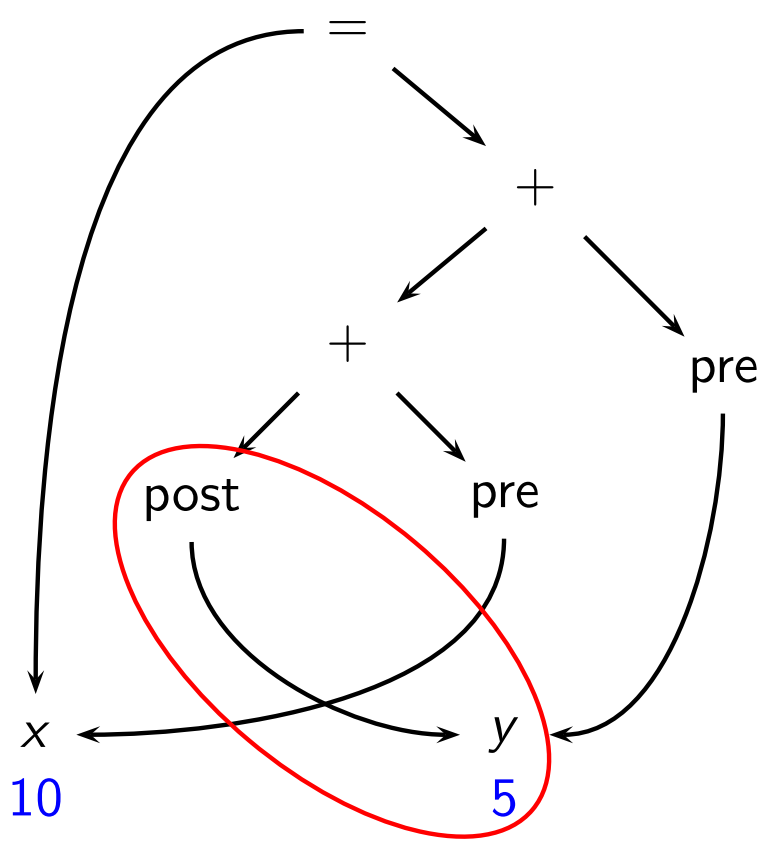
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



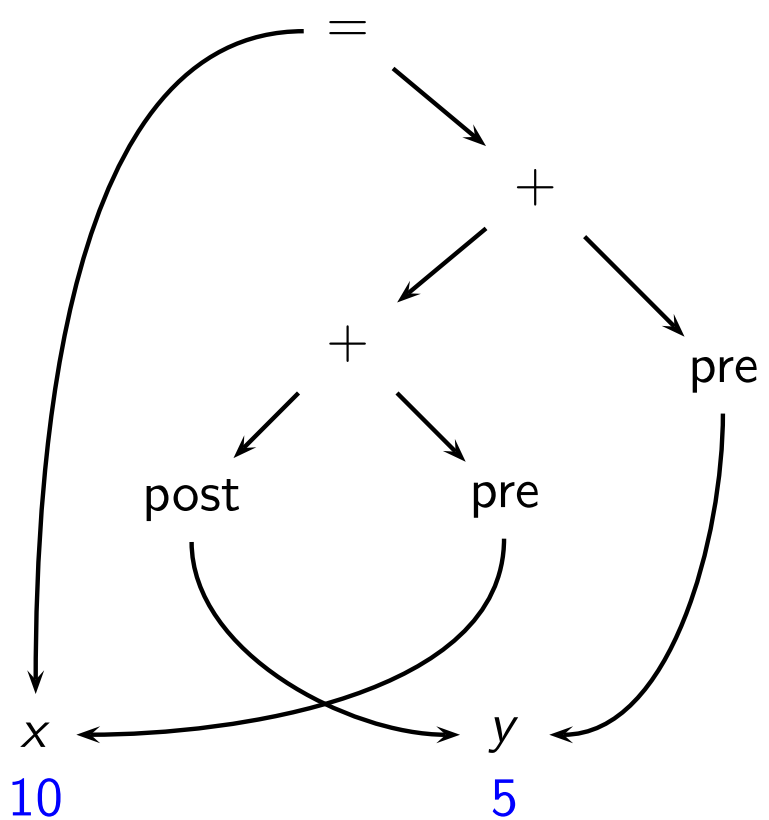
Inspect GIMPLE When in Doubt (1)

```
int x=2, y=3;
x = y++ + ++x + ++y;
```

What are the values of x and y?

x = 10, y = 5

```
x = 2;
y = 3;
x = x + 1;
D.1572 = y + x;
y = y + 1;
x = D.1572 + y;
y = y + 1;
```



Inspect GIMPLE When in Doubt (2)

- How is `a[i] = i++` handled?

  This is an undefined behaviour as per C standards.

- What is the order of parameter evaluation?

  For a call `f(getX(),getY())`, is the order left to right? arbitrary?

  Is the evaluation order in GCC consistent?

- Understanding complicated declarations in C can be difficult

  What does the following declaration mean :

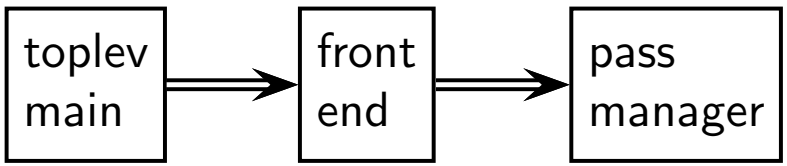
  ```
  int * (* (*MYVAR) (int) ) [10];
  ```

  Hint: Use `-fdump-tree-original-raw-verbose` option. The dump to see is `003t.original`

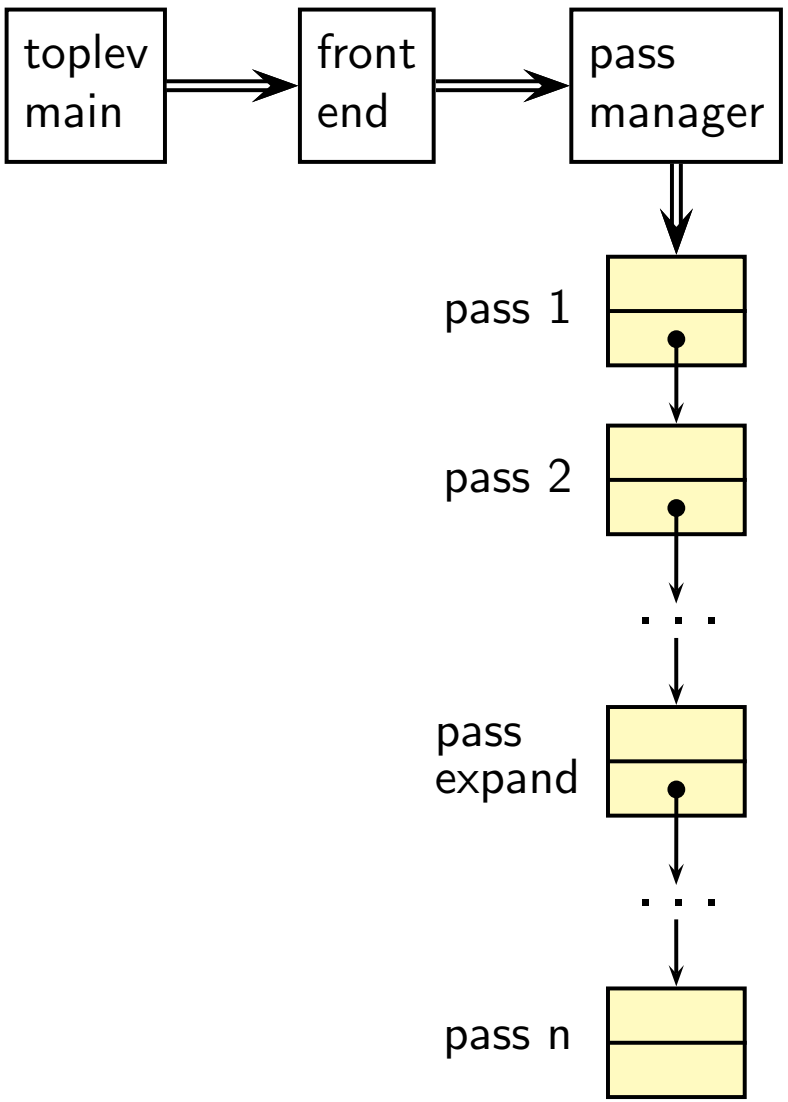
Part 4

Plugins in GCC

Plugin Structure in `cc1`



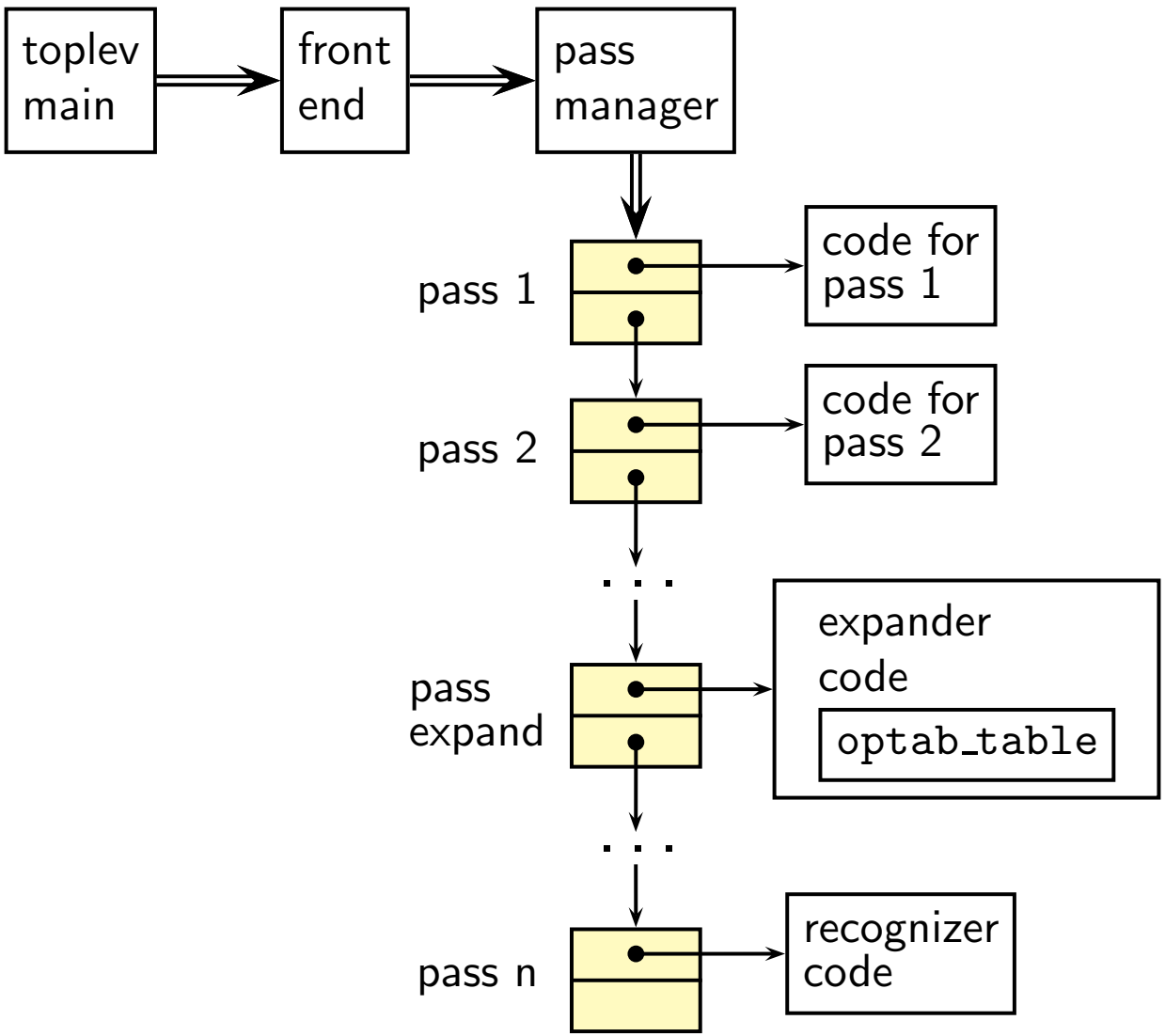
Plugin Structure in `cc1`



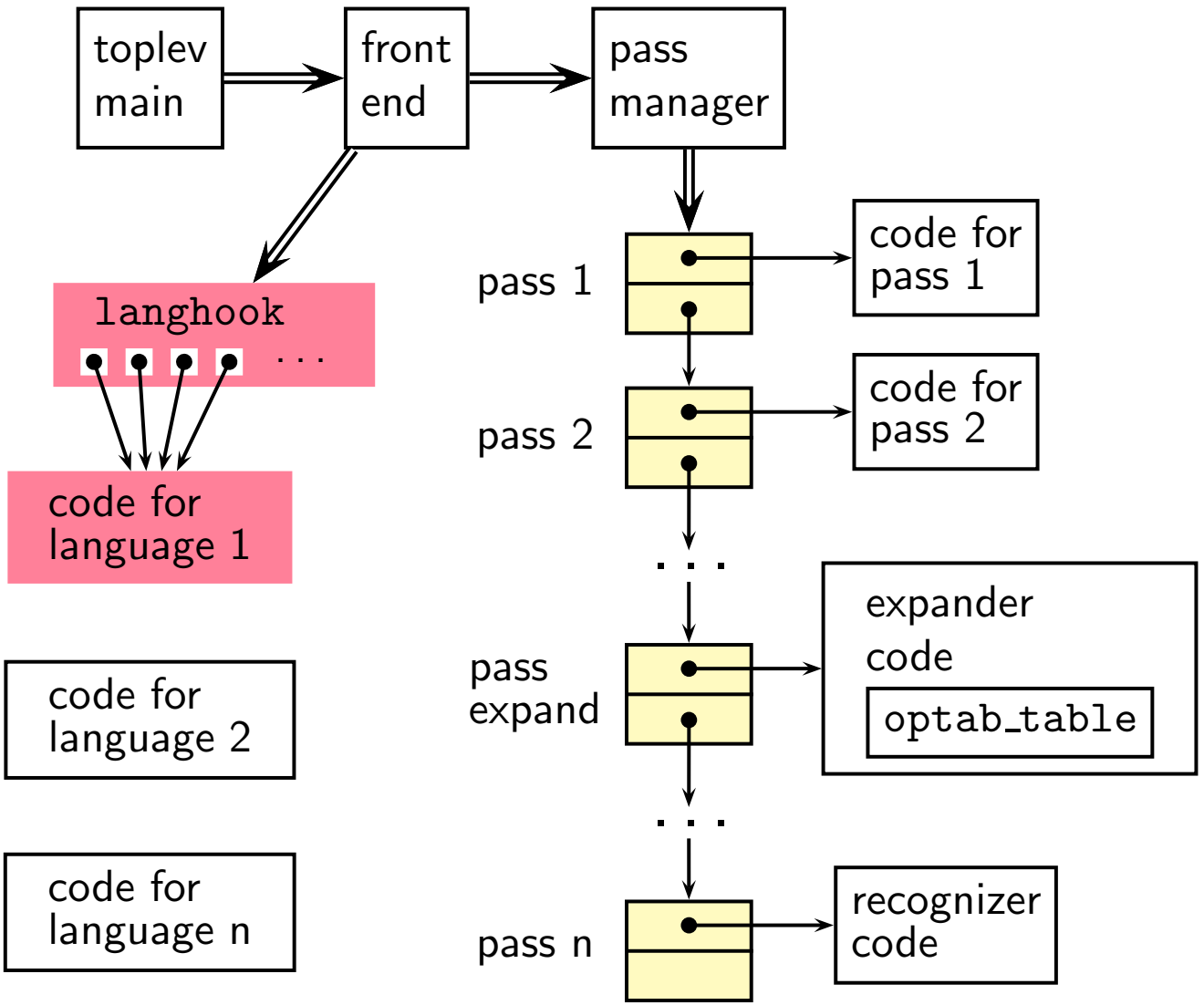
Double arrow represents control flow whereas single arrow represents pointer or index

For simplicity, we have included all passes in a single list. Actually passes are organized into five lists and are invoked as five different sequences

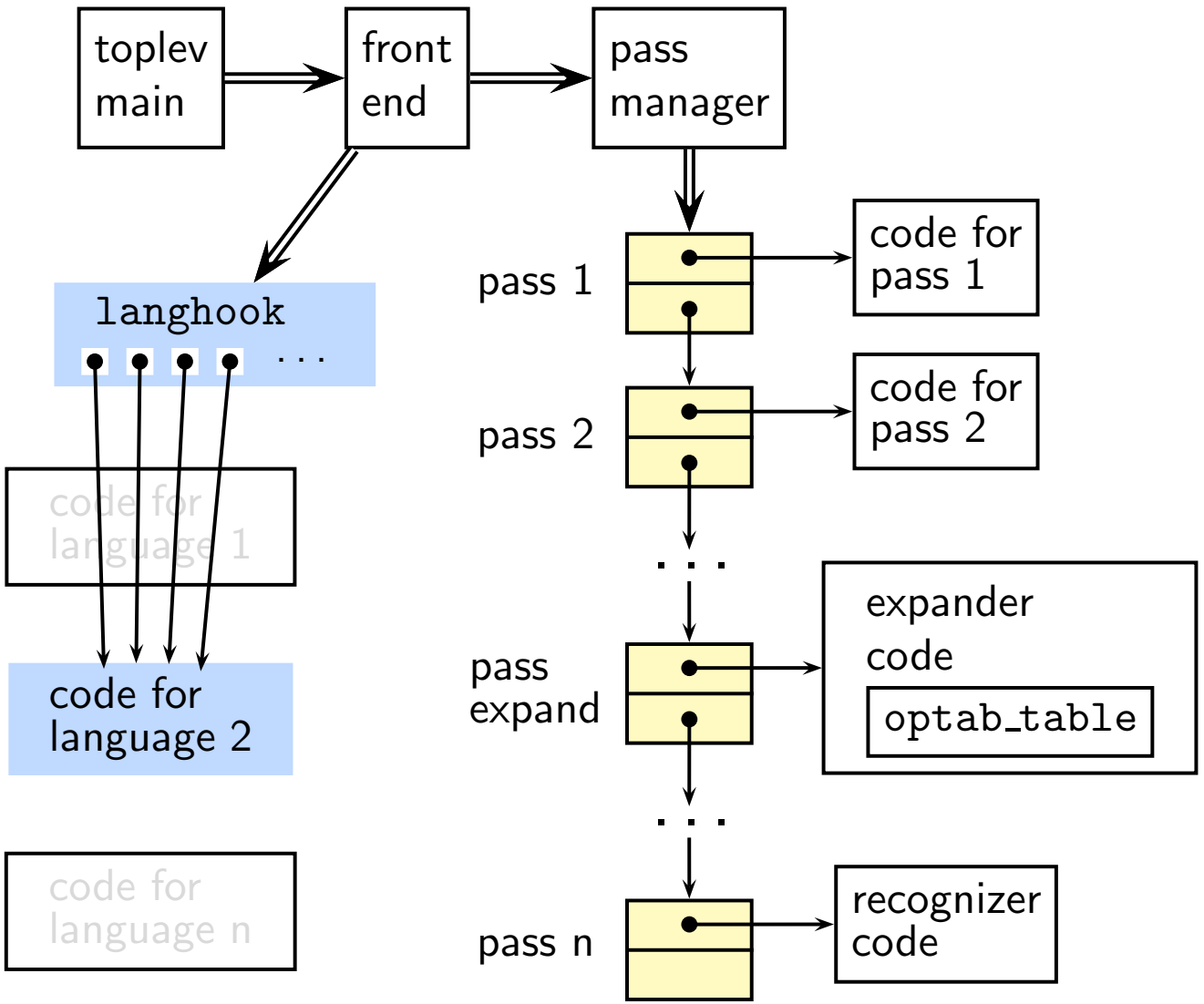
Plugin Structure in `cc1`



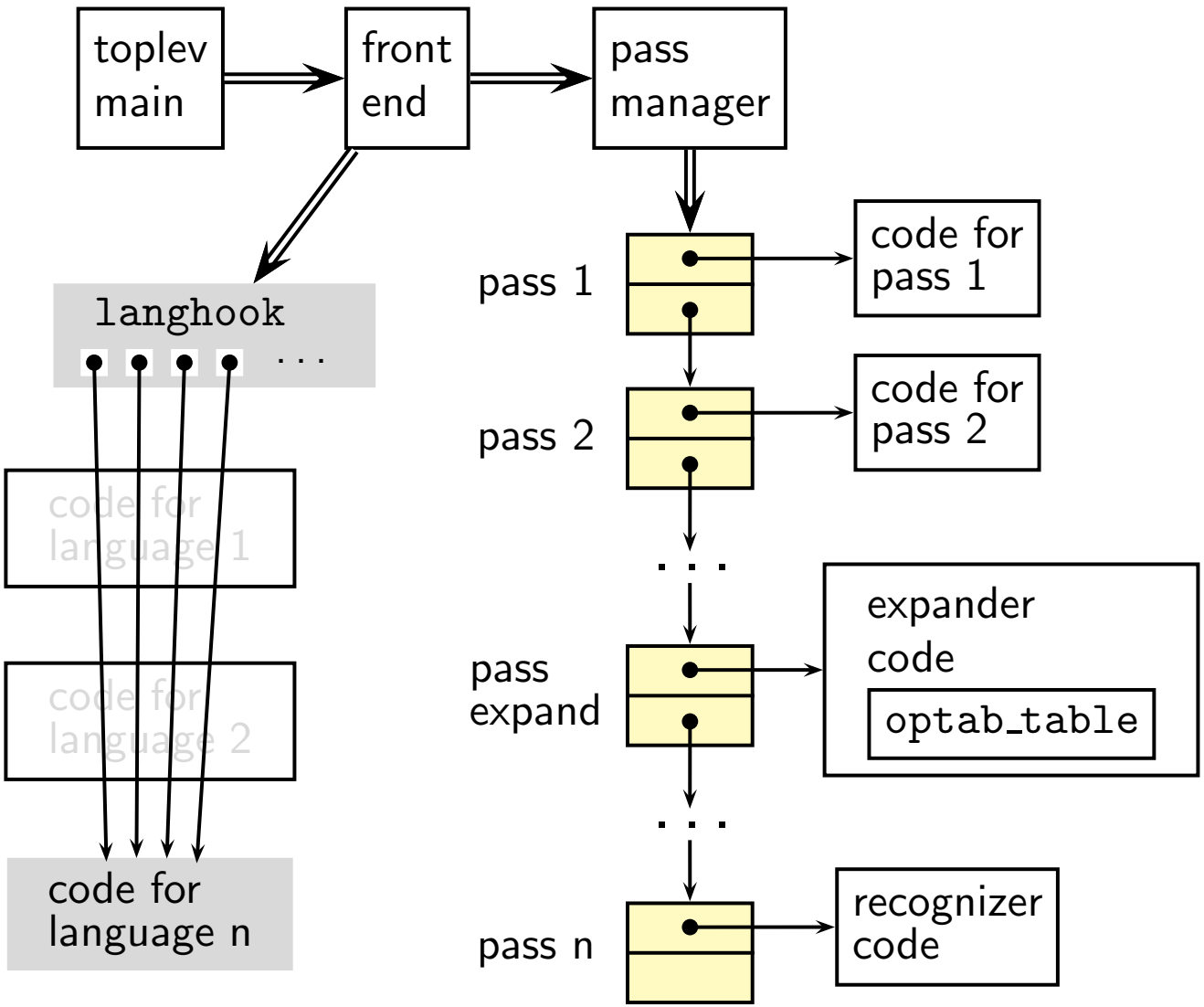
Plugin Structure in `cc1`



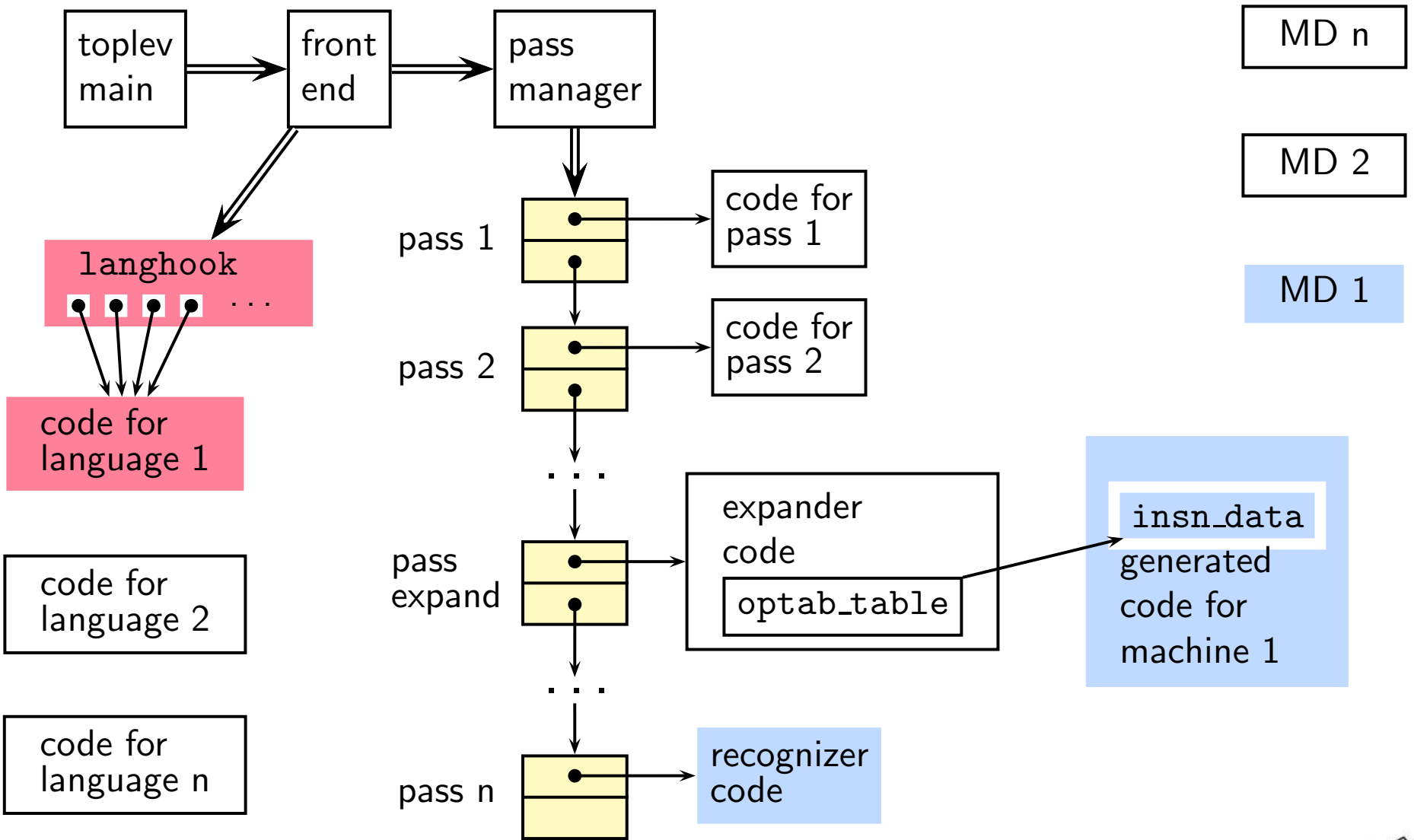
Plugin Structure in `cc1`



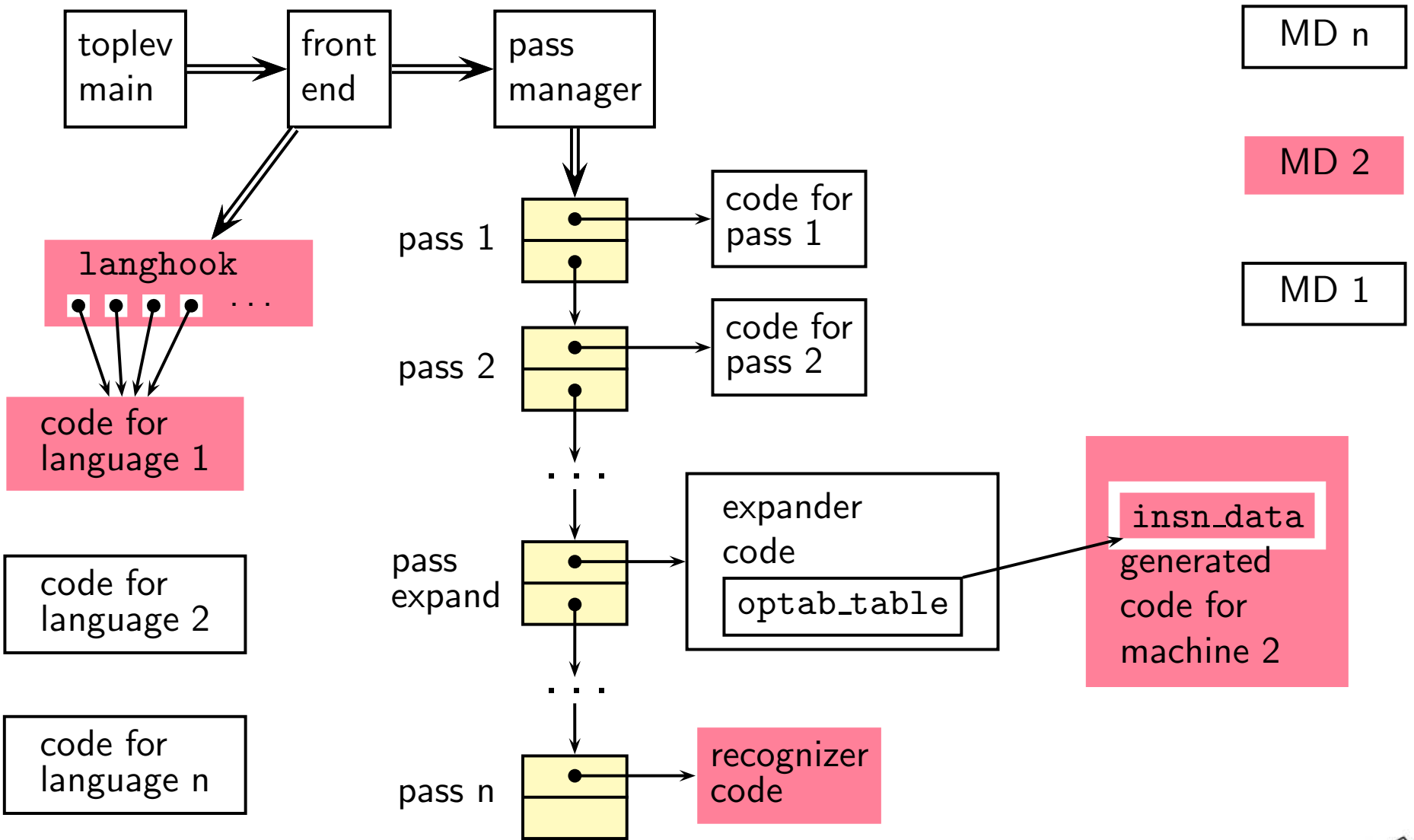
Plugin Structure in `cc1`



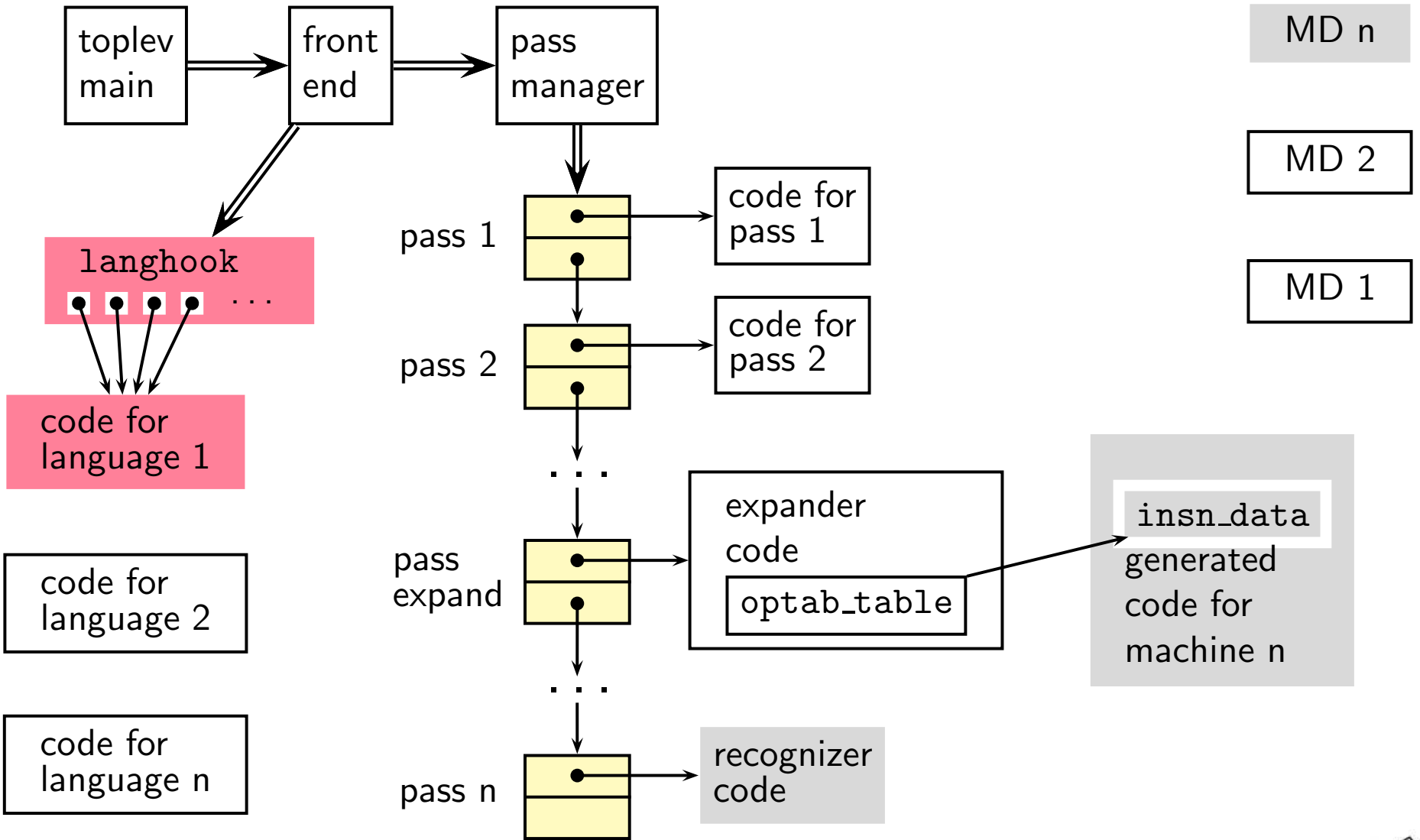
Plugin Structure in `cc1`



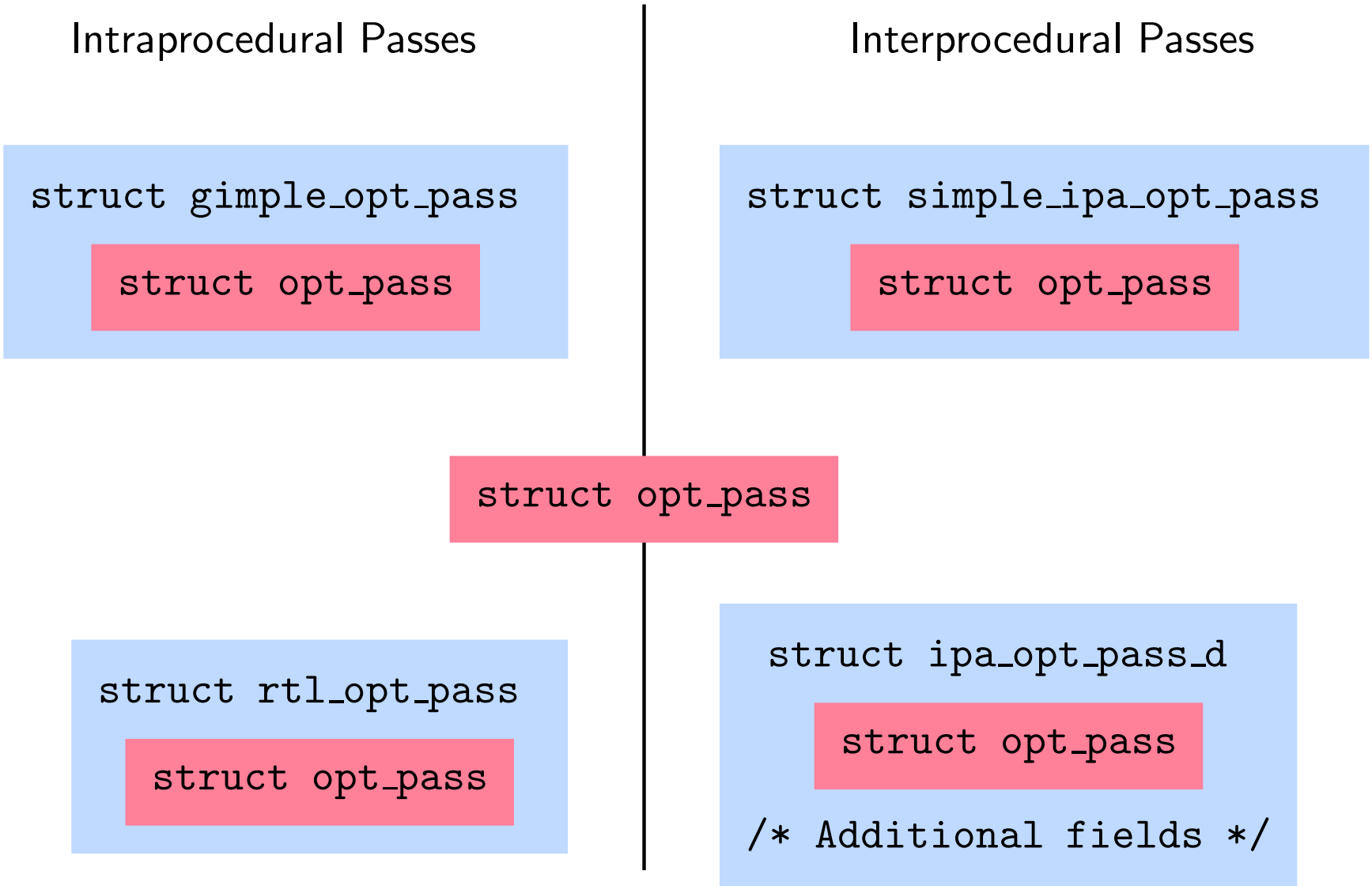
Plugin Structure in `cc1`



Plugin Structure in `cc1`



Plugins for Pass Specifications



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 on a Translation Unit

Pass variable: `all_simple_ipa_passes`

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


Plugins for Interprocedural Passes across a Translation Unit

Pass variable: `all_regular_ipa_passes`

```
struct ipa_opt_pass_d
{
  struct opt_pass pass;
  void (*generate_summary) (void);
  void (*read_summary) (void);
  void (*write_summary) (struct cgraph_node_set_def *,
                         struct varpool_node_set_def *);
  void (*write_optimization_summary)(struct cgraph_node_set_def *,
                                     struct varpool_node_set_def *);
  void (*read_optimization_summary) (void);
  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 *);
};
```

Predefined Pass Lists

Pass List	Purpose
`all_lowering_passes`	AST to CFG translation
`all_small_ipa_passes`	Interprocedural passes restricted to a single translation unit
`all_regular_ipa_passes`	Interprocedural passes on a translation unit as well as across translation units (during WPA/IPA of LTO)
`all_late_ipa_passes`	Interprocedural passes on partitions created by LTO (after WPA/IPA)
`all_lto_gen_passes`	Passes to encode program for LTO
`all_passes`	Intraprocedural passes on GIMPLE and RTL

Registering a Pass as a Static Plugin

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
   (For debuging `cc1` from within `gcc`, see:
   `http://gcc.gnu.org/ml/gcc/2004-03/msg01195.html`)

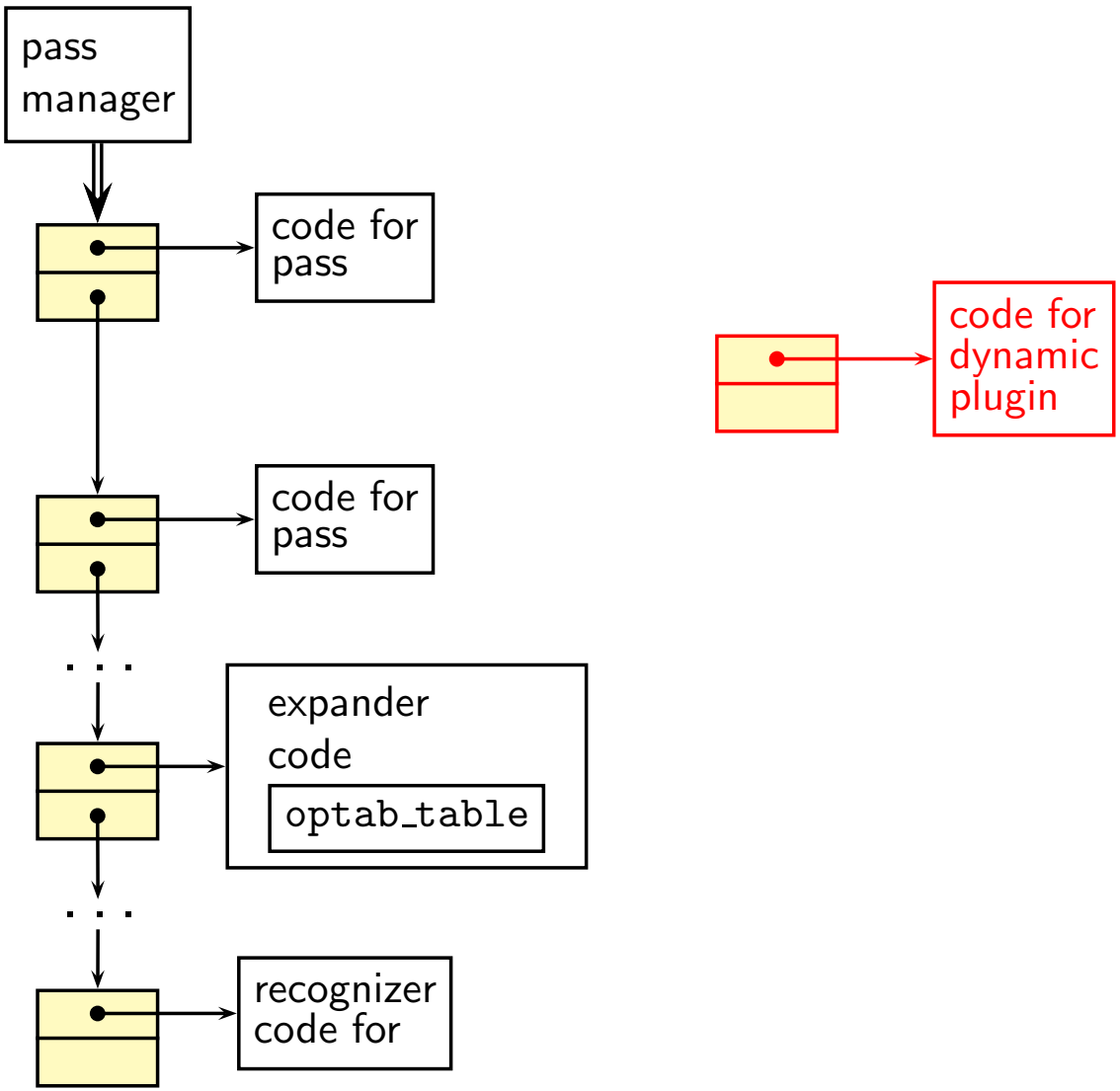
Dynamic Plugins

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

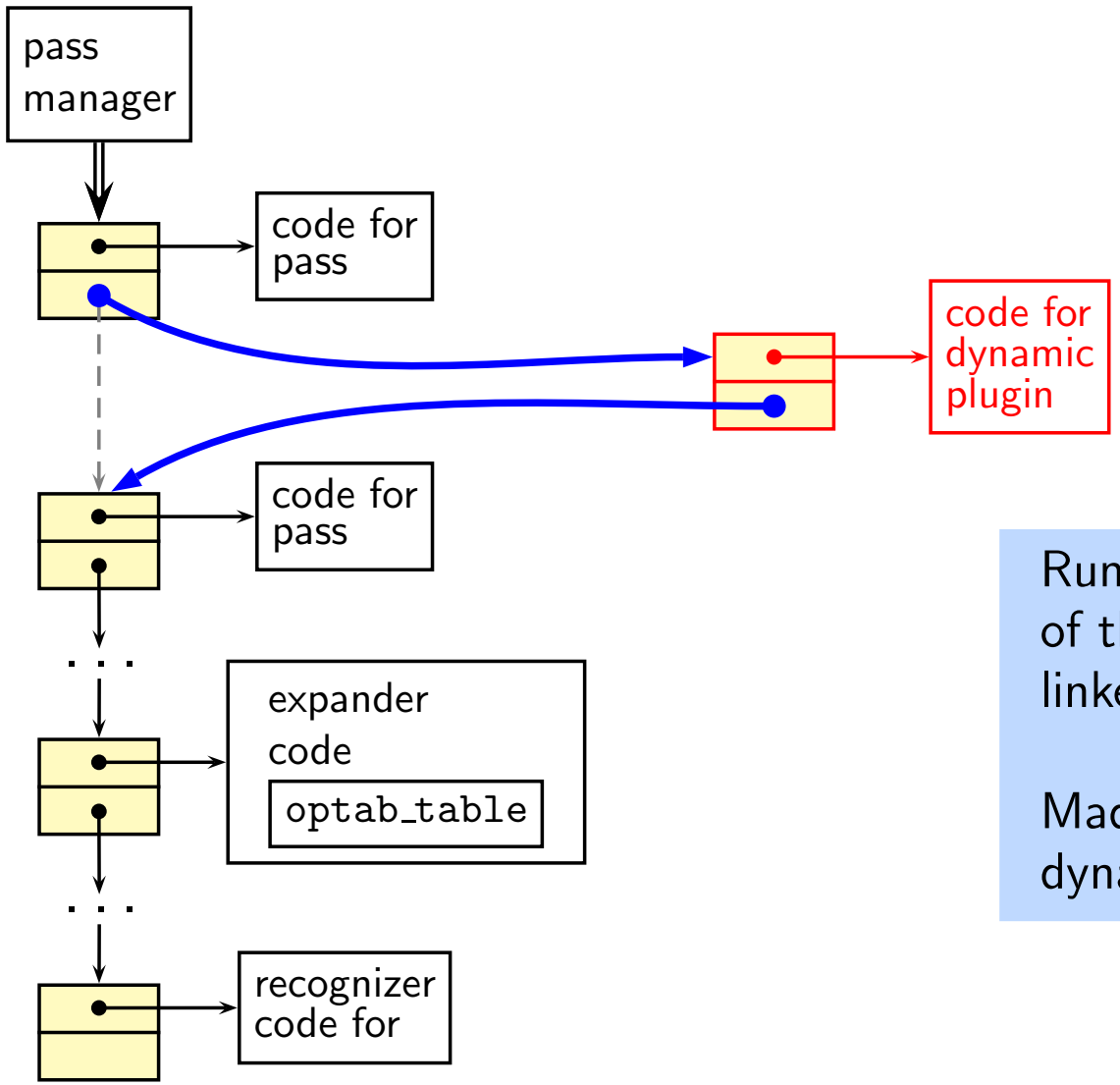
  `-fplugin=/path/to/name.so`

  Arguments required can be supplied as name-value pairs

The Mechanism of Dynamic Plugin



The Mechanism of Dynamic Plugin



Runtime initialization of the appropriate linked list of passes

Made possible by dynamic linking

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 (string
                              in the structure specification) 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;
}
```


Makefile for Creating and Using a Dynamic Plugin

```
CC = $(INSTALL_D)/bin/g++
PLUGIN_SOURCES = new-pass.c
PLUGIN_OBJECTS = $(patsubst %.c,%.o,$(PLUGIN_SOURCES ))
GCCPLUGINS_DIR = $(shell $(CC) -print-file-name=plugin)
CFLAGS+= -fPIC -O2
INCLUDE = -Iplugin/include

%.o : %.c
$(CC) $(CFLAGS) $(INCLUDE) -c $<

new-pass.so: $(PLUGIN_OBJECTS)
        $(CC) $(CFLAGS) $(INCLUDE) -shared $^ -o $@

test_plugin: test.c
        $(CC) -fplugin=./new-pass.so $^ -o $@ -fdump-tree-all
```

Part 5

Manipulating GIMPLE

What is GIMPLE ?

- GIMPLE is influenced by SIMPLE IR of McCat compiler
- But GIMPLE is not same as SIMPLE (GIMPLE supports GOTO)
- It is a simplified subset of GENERIC
  - 3 address representation
  - Control flow lowering
  - Cleanups and simplification, restricted grammar
- Benefit : Optimizations become easier

GIMPLE Goals

The Goals of GIMPLE are

- Lower control flow
  Sequenced statements + conditional and unconditional jumps
- Simplify expressions
  Typically one operator and at most two operands
- Simplify scope
  Move local scope to block begin, including temporaries

Observing Internal Form of GIMPLE

`test.c.004t.gimple` with compilation option `-fdump-tree-all`

```
x = 10;
y = 5;
D.1954 = x * y;
a.0 = a;
x = D.1954 + a.0;
a.1 = a;
D.1957 = a.1 * x;
y = y - D.1957;
```

`test.c.004t.gimple` with compilation option `-fdump-tree-all-raw`

```
gimple_assign <integer_cst, x, 10, NULL>
gimple_assign <integer_cst, y, 5, NULL>
gimple_assign <mult_expr, D.1954, x, y>
gimple_assign <var_decl, a.0, a, NULL>
gimple_assign <plus_expr, x, D.1954, a.0>
gimple_assign <var_decl, a.1, a, NULL>
gimple_assign <mult_expr, D.1957, a.1, x>
gimple_assign <minus_expr, y, y, D.1957>
```

Observing Internal Form of GIMPLE

test.c.004t.gimple with compilation option -fdump-tree-all

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

test.c.004t.gimple with compilation option -fdump-tree-all-raw

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```


Observing Internal Form of GIMPLE

test.c.004t.gimple with compilation option -fdump-tree-all

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

test.c.004t.gimple with compilation option -fdump-tree-all-raw

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```


Observing Internal Form of GIMPLE

test.c.004t.gimple with compilation option -fdump-tree-all

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

test.c.004t.gimple with compilation option -fdump-tree-all-raw

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```

Observing Internal Form of GIMPLE

test.c.004t.gimple with compilation option -fdump-tree-all

```
  if (a < c)
    goto <D.1953>;
  else
    goto <D.1954>;
<D.1953>:
  a = b + c;
  goto <D.1955>;
<D.1954>:
  a = b - c;
<D.1955>:
```

test.c.004t.gimple with compilation option -fdump-tree-all-raw

```
gimple_cond <lt_expr, a,c,<D.1953>, <D.1954>>
gimple_label <<D.1953>>
gimple_assign <plus_expr, a, b, c>
gimple_goto <<D.1955>>
gimple_label <<D.1954>>
gimple_assign <minus_expr, a, b, c>
gimple_label <<D.1955>>
```


Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                    gsi_next (&gsi))
        find_pointer_assignments(gsi_stmt (gsi));
}
```


Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                    gsi_next (&gsi))
        find_pointer_assignments(gsi_stmt (gsi));
}
```

Basic block iterator

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                    gsi_next (&gsi))
        find_pointer_assignments(gsi_stmt (gsi));
}
```

GIMPLE statement iterator

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                    gsi_next (&gsi))
        find_pointer_assignments(gsi_stmt (gsi));
}
```

Get the first statement of bb

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                    gsi_next (&gsi))
        find_pointer_assignments(gsi_stmt (gsi));
}
```

True if end reached

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                    gsi_next (&gsi))
        find_pointer_assignments(gsi_stmt (gsi));
}
```

Advance iterator to the next GIMPLE stmt

Iterating Over GIMPLE Statements

- A basic block contains a doubly linked-list of GIMPLE statements
- The statements are represented as GIMPLE tuples, and the operands are represented by tree data structure
- Processing of statements can be done through iterators

```
basic_block bb;
gimple_stmt_iterator gsi;

FOR_EACH_BB (bb)
{
    for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                    gsi_next (&gsi))
        find_pointer_assignments(gsi_stmt (gsi));
}
```

Return the current statement

Printing Successors of a Basic Block

```
edge e;
edge_iterator ei;
basic_block bb;

FOR_EACH_BB_FN (bb, cfun)
{
   fprintf(dump_file, "\n Successor(s) of basic block bb%d: ",
                          bb->index);
   FOR_EACH_EDGE (e, ei, bb->succs)
   {
        basic_block succ_bb = e->dest;
        fprintf(dump_file, "bb%d\t ", succ_bb->index);
   }
   fprintf(dump_file, "\n");
}
```

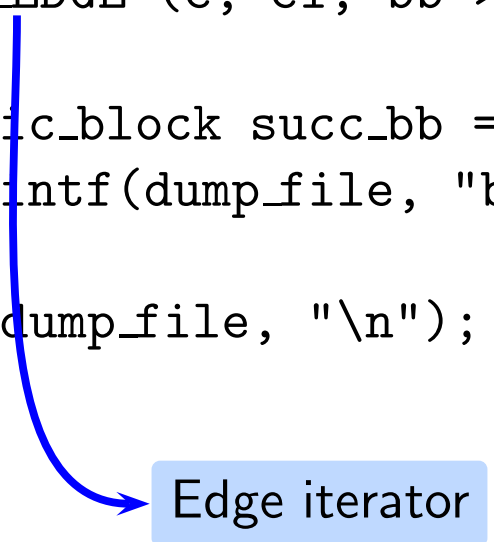

Printing Successors of a Basic Block

```
edge e;
edge_iterator ei;
basic_block bb;

FOR_EACH_BB_FN (bb, cfun)
{
   fprintf(dump_file, "\n Successor(s) of basic block bb%d: ",
                           bb->index);
   FOR_EACH_EDGE (e, ei, bb->succs)
   {
        basic_block succ_bb = e->dest;
        fprintf(dump_file, "bb%d\t ", succ_bb->index);
   }
   fprintf(dump_file, "\n");
}
```

Basic block iterator for current function represented by cfun

Printing Successors of a Basic Block

```
edge e;
edge_iterator ei;
basic_block bb;

FOR_EACH_BB_FN (bb, cfun)
{
   fprintf(dump_file, "\n Successor(s) of basic block bb%d: ",
                          bb->index);
   FOR_EACH_EDGE (e, ei, bb->succs)
   {
        basic_block succ_bb = e->dest;
        fprintf(dump_file, "bb%d\t ", succ_bb->index);
   }
   fprintf(dump_file, "\n");
}
```



Other Useful APIs for Manipulating GIMPLE

Extracting parts of GIMPLE statements:

- `gimple_assign_lhs`: left hand side
- `gimple_assign_rhs1`: left operand of the right hand side
- `gimple_assign_rhs2`: right operand of the right hand side
- `gimple_assign_rhs_code`: operator on the right hand side

A complete list can be found in the file `gimple.h`

Discovering More Information from GIMPLE

- Discovering local variables
- Discovering global variables
- Discovering pointer variables
- Discovering assignment statements involving pointers
  (i.e. either the result or an operand is a pointer variable)

Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;
     unsigned int u;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```


Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;
     unsigned int u;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```

Local variable iterator

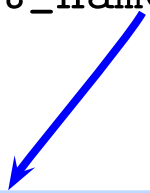
Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;
     unsigned int u;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```

Exclude variables that do not appear in the source

Discovering Local Variables in GIMPLE IR

```
static void gather_local_variables ()
{
     tree list;
     unsigned int u;

     if (!dump_file)
          return;

     fprintf(dump_file,"\nLocal variables : ");
     FOR_EACH_LOCAL_DECL (cfun, u, list)
     {
          if (!DECL_ARTIFICIAL (list))
               fprintf(dump_file, "%s\n", get_name (list));
     }
}
```



Find the name from the TREE node

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

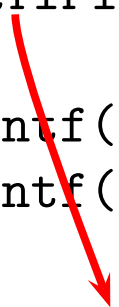

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

List of global variables of the current function

Discovering Global Variables in GIMPLE IR

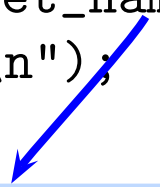
```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```



Exclude variables that do not appear in the source

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```



Find the name from the TREE node

Discovering Global Variables in GIMPLE IR

```
static void gather_global_variables ()
{
        struct varpool_node *node;

        if (!dump_file)
                return;

        fprintf(dump_file,"\nGlobal variables : ");
        for (node = varpool_nodes; node; node = node->next)
        {
                tree var = node->decl;
                if (!DECL_ARTIFICIAL(var))
                {
                        fprintf(dump_file, get_name(var));
                        fprintf(dump_file,"\n");
                }
        }
}
```

Go to the next item in the list

Assignment Statements Involving Pointers

```
int *p, *q;
void callme (int);
int main ()
{
    int  a, b;
    p = &b;
    callme (a);
    return 0;
}
void callme (int a)
{
    a = *(p + 3);
    q = &a;
}
```

```
main ()
{  int D.1965;
    int a;
    int b;

    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   int * p.0;
    int a.1;

    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Discovering Pointers in GIMPLE IR

```
static bool
is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool
is_pointer_type (tree type)
{
     if (POINTER_TYPE_P (type))
         return true;
     if (TREE_CODE (type) == ARRAY_TYPE)
         return is_pointer_type (TREE_TYPE (type));
     /* Return true if it is an aggregate type. */
     return AGGREGATE_TYPE_P (type);
}
```


Discovering Pointers in GIMPLE IR

```
static bool
is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool
is_pointer_type (tree type)
{
     if (POINTER_TYPE_P (type))
         return true;
     if (TREE_CODE (type) == ARRAY_TYPE)
         return is_pointer_type (TREE_TYPE (type));
     /* Return true if it is an aggregate type. */
     return AGGREGATE_TYPE_P (type);
}
```

Data type of the expression

Discovering Pointers in GIMPLE IR

```
static bool
is_pointer_var (tree var)
{
    return is_pointer_type (TREE_TYPE (var));
}

static bool
is_pointer_type (tree type)
{
     if (POINTER_TYPE_P (type))
         return true;
     if (TREE_CODE (type) == ARRAY_TYPE)
         return is_pointer_type (TREE_TYPE (type));
     /* Return true if it is an aggregate type. */
     return AGGREGATE_TYPE_P (type);
}
```

Defines what kind of node it is

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{   if (is_gimple_assign (stmt))
    {    tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
            {    fprintf (dump_file, "Pointer Statement :");
                 print_gimple_stmt (dump_file, stmt, 0, 0);
                 num_ptr_stmts++;
            }
         }
    }
}
```


Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{   if (is_gimple_assign (stmt))
    {    tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
            {    fprintf (dump_file, "Pointer Statement :");
                 print_gimple_stmt (dump_file, stmt, 0, 0);
                 num_ptr_stmts++;
            }
         }
    }
}
```

Extract the LHS of the assignment statement

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{   if (is_gimple_assign (stmt))
    {    tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
            {    fprintf (dump_file, "Pointer Statement :");
                 print_gimple_stmt (dump_file, stmt, 0, 0);
                 num_ptr_stmts++;
            }
         }
    }
}
```

Extract the first operand of the RHS

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{   if (is_gimple_assign (stmt))
    {    tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
            {    fprintf (dump_file, "Pointer Statement :");
                 print_gimple_stmt (dump_file, stmt, 0, 0);
                 num_ptr_stmts++;
            }
         }
    }
}
```

Extract the second operand of the RHS

Discovering Assignment Statements Involving Pointers

```
static void
find_pointer_assignments (gimple stmt)
{   if (is_gimple_assign (stmt))
    {    tree lhsop = gimple_assign_lhs (stmt);
         tree rhsop1 = gimple_assign_rhs1 (stmt);
         tree rhsop2 = gimple_assign_rhs2 (stmt);
         /* Check if either LHS, RHS1 or RHS2 operands
            can be pointers. */
         if ((lhsop && is_pointer_var (lhsop)) ||
             (rhsop1 && is_pointer_var (rhsop1)) ||
             (rhsop2 && is_pointer_var (rhsop2)))
         {  if (dump_file)
            {    fprintf (dump_file, "Pointer Statement :");
                 print_gimple_stmt (dump_file, stmt, 0, 0);
                 num_ptr_stmts++;
            }
         }
    }
}
```

Pretty print the GIMPLE statement

Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                              gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```


Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                          gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```

Basic block iterator parameterized with function

Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                          gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```

Current function (i.e. function being compiled)

Putting it Together at the Intraprocedural Level

```
static unsigned int
intra_gimple_manipulation (void)
{
   basic_block bb;
   gimple_stmt_iterator gsi;

   initialize_var_count ();
   FOR_EACH_BB_FN (bb, cfun)
   {
       for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi);
                                          gsi_next (&gsi))
            find_pointer_assignments (gsi_stmt (gsi));
   }
   print_var_count ();
   return 0;
}
```

GIMPLE statement iterator

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1
- For `callme`: 2

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1
- For `callme`: 2

Why is the pointer in the red statement being missed?

Intraprocedural Analysis Results

```
main ()
{   ...
    p = &b;
    callme (a);
    D.1965 = 0;
    return D.1965;
}
callme (int a)
{   ...
    p.0 = p;
    a.1 = MEM[(int *)p.0 + 12B];
    a = a.1;
    q = &a;
}
```

Information collected by intraprocedural Analysis pass

- For `main`: 1
- For `callme`: 2

Why is the pointer in the red statement being missed?

Because it is deeper in the tree and our program does not search deeper in the tree

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```


Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Iterating over all the callgraph nodes

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Setting the current function in the context

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Basic Block Iterator

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

GIMPLE Statement Iterator

Extending our Pass to Interprocedural Level

```
static unsigned int
inter_gimple_manipulation (void)
{
   struct cgraph_node *node;
   basic_block bb;
   gimple_stmt_iterator gsi;
   initialize_var_count ();
   for (node = cgraph_nodes; node; node=node->next) {
      /* Nodes without a body, and clone nodes are not interesting. */
      if (!gimple_has_body_p (node->decl) || node->clone_of)
            continue;
      push_cfun (DECL_STRUCT_FUNCTION (node->decl));
      FOR_EACH_BB (bb) {
          for (gsi=gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi))
               find_pointer_assignments (gsi_stmt (gsi));
      }
      pop_cfun ();
   }
   print_var_count ();
   return 0;
}
```

Resetting the function context

Interprocedural Results

Number of Pointer Statements = 3

Interprocedural Results

Number of Pointer Statements = 3

Observation:

- Information can be collected for all the functions in a single pass
- Better scope for optimizations

Last but not the least . . .

Thank You!