Code optimization in GCC

Sébastian Pop

Université Louis Pasteur

Strasbourg

FRANCE

Introduction

`GCC : GNU Compiler Collection`

- C, C++, Java, Ada, Frotran, Mercury, . . .

Introduction

`GCC : GNU Compiler Collection`

- C, C++, Java, Ada, Frotran, Mercury, . . .
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, . . .

Introduction

`GCC : GNU Compiler Collection`

- C, C++, Java, Ada, Frotran, Mercury, . . .
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, . . .
- Main compiler in GNU world

Introduction

`GCC : GNU Compiler Collection`

- C, C++, Java, Ada, Frotran, Mercury, . . .
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, . . .
- Main compiler in GNU world
- Apple's system compiler.

Introduction

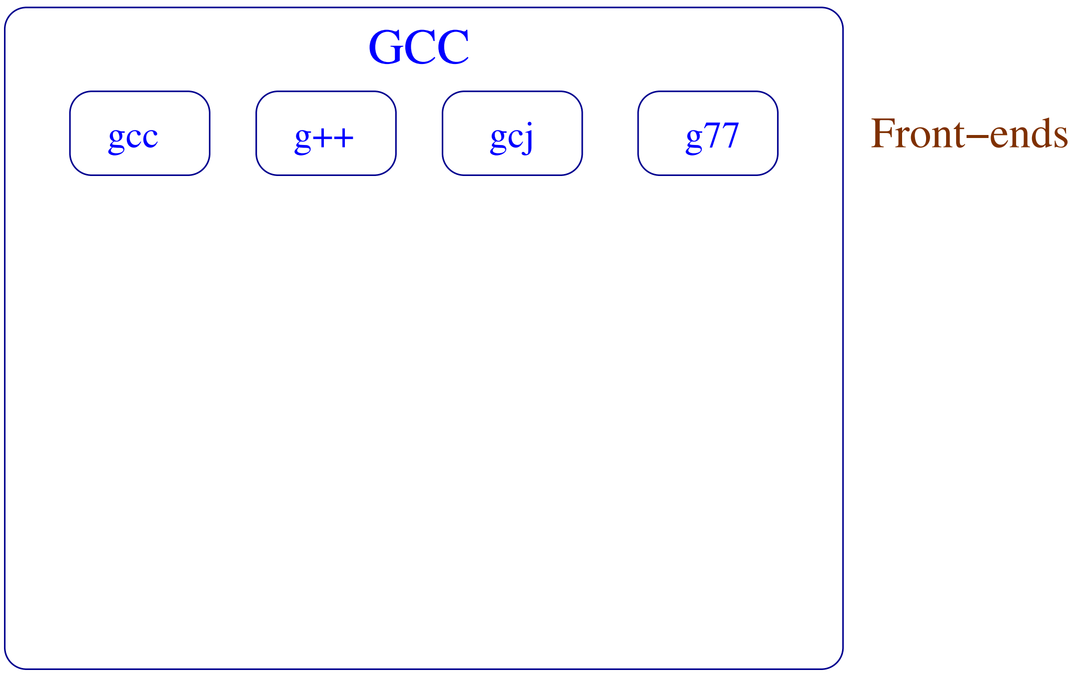
`GCC : GNU Compiler Collection`

- C, C++, Java, Ada, Frotran, Mercury, . . .
- Generates code for 43 different architectures: i386, ia64, m68k, sparc, . . .
- Main compiler in GNU world
- Apple's system compiler.
- Industrial compiler.

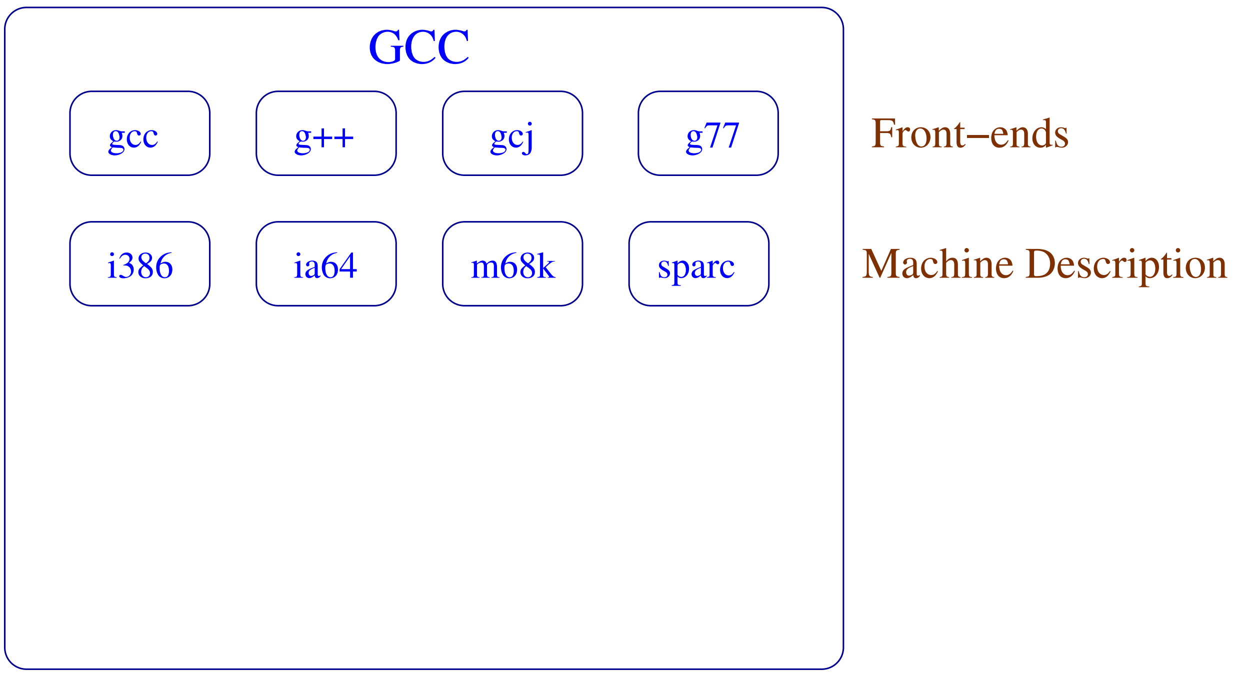
Front-ends / back-end

GCC

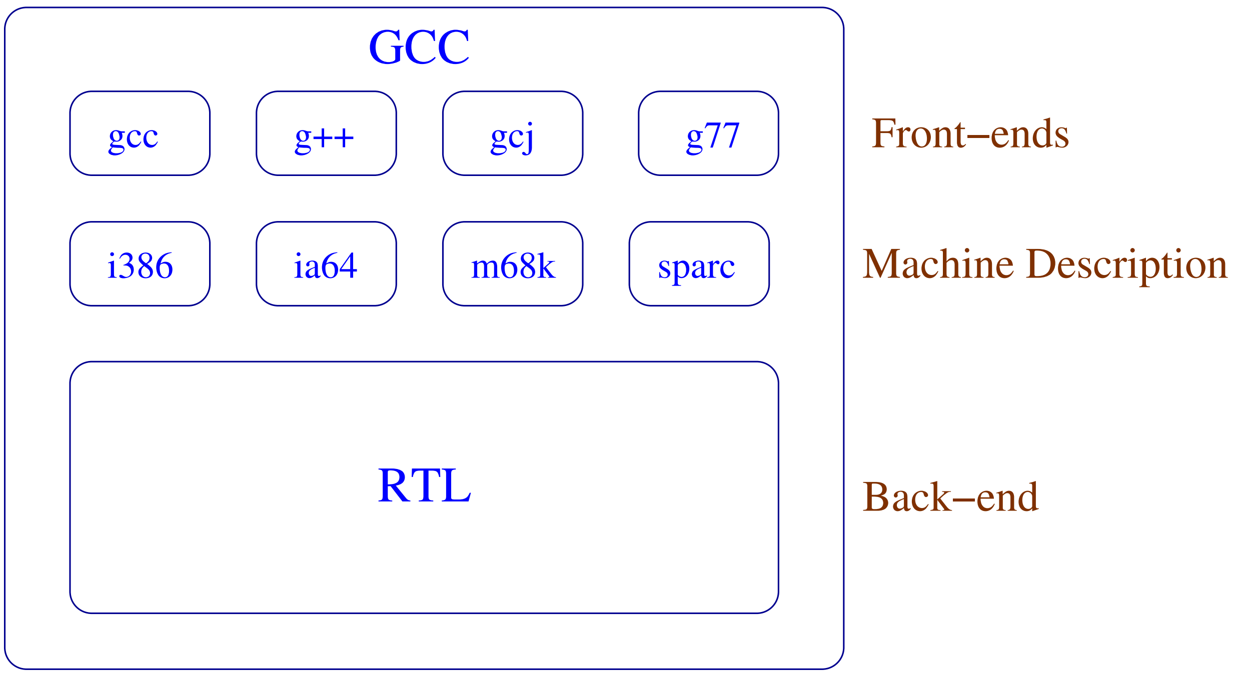
Front-ends / back-end



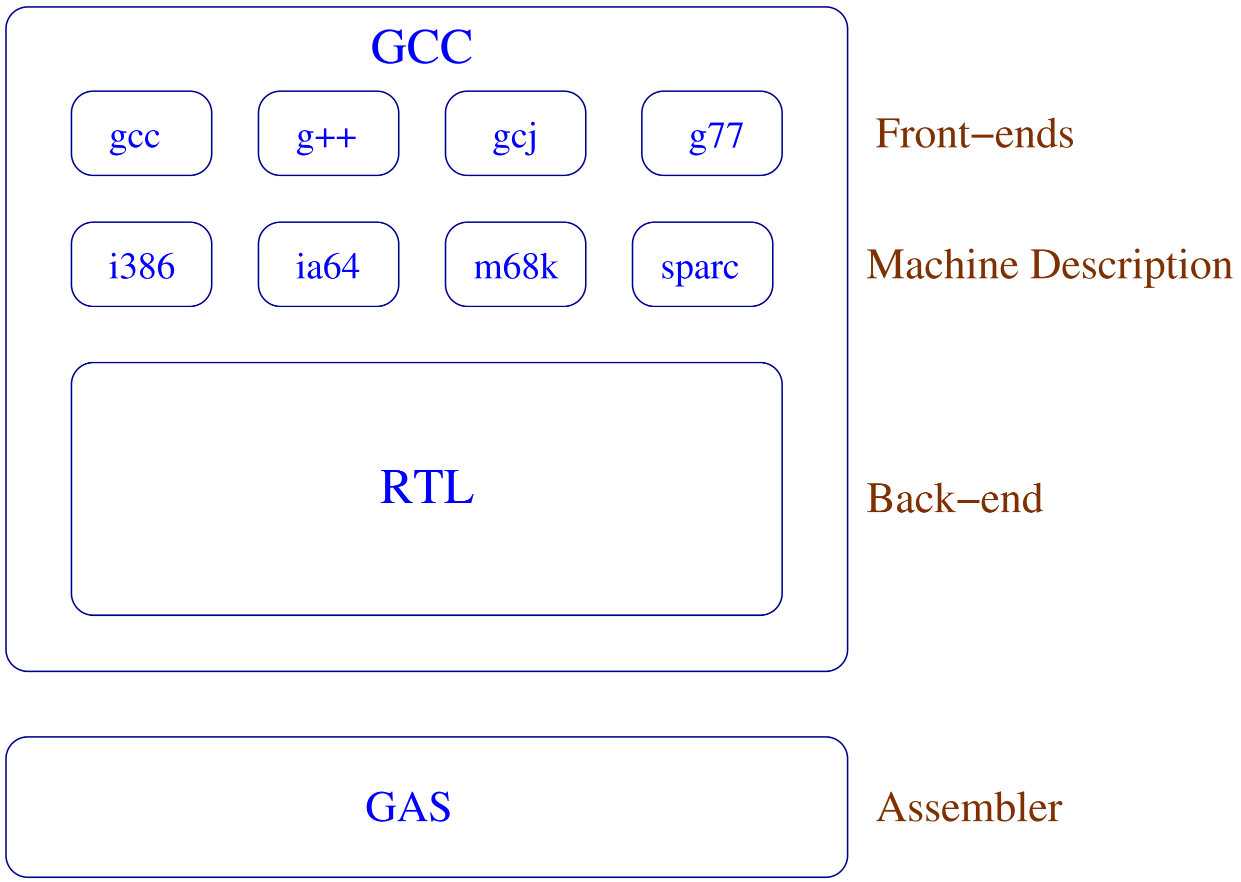
Front-ends / back-end



Front-ends / back-end



Front-ends / back-end



Exemple: cross-compilation

- Suppose that I want to generate Sparc code:

  `-target=sparc`

Exemple: cross-compilation

- Suppose that I want to generate Sparc code: `–target=sparc`
- I build GCC on my laptop: `–build=i586`

Exemple: cross-compilation

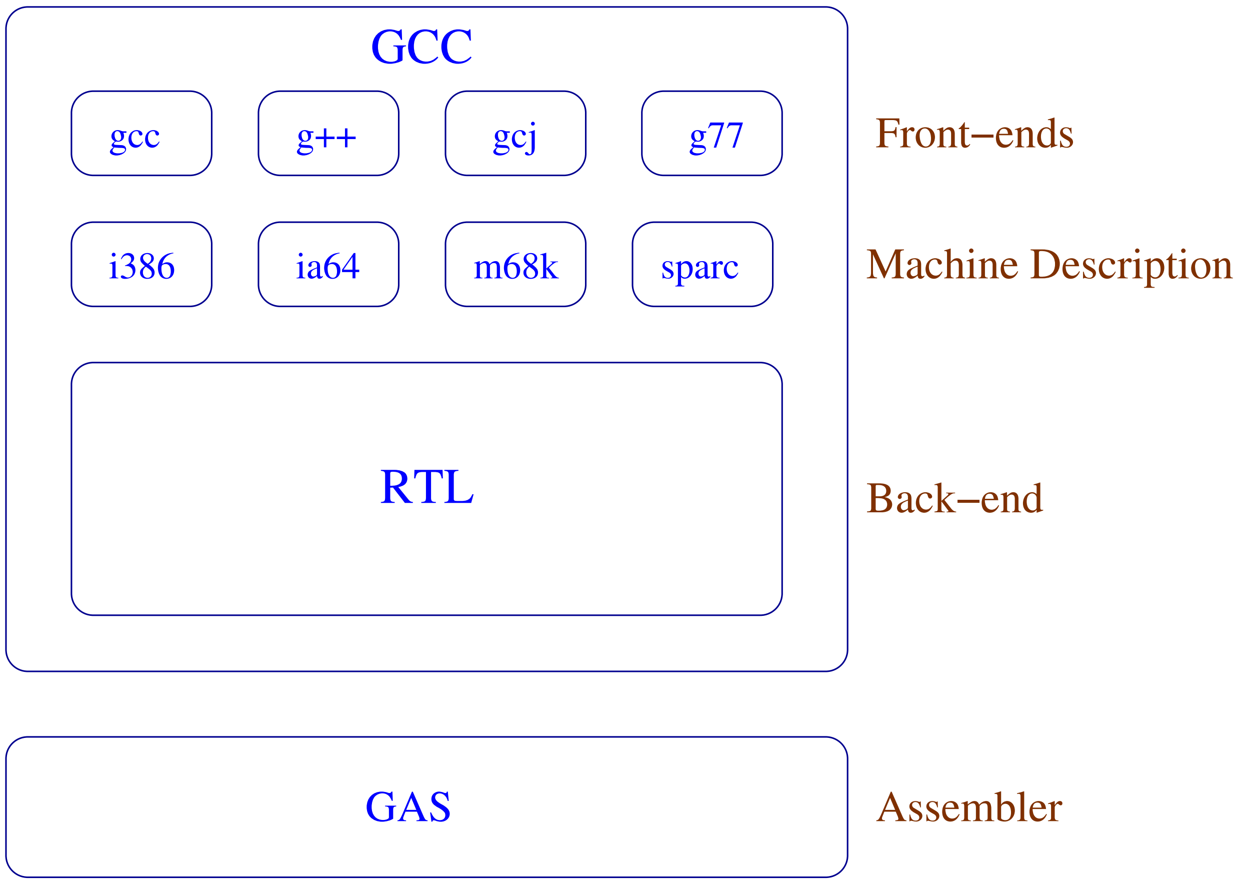
- Suppose that I want to generate Sparc code: `-target=sparc`
- I build GCC on my laptop: `-build=i586`
- and I run the compiler on my laptop: `-host=i586`

Exemple: cross-compilation

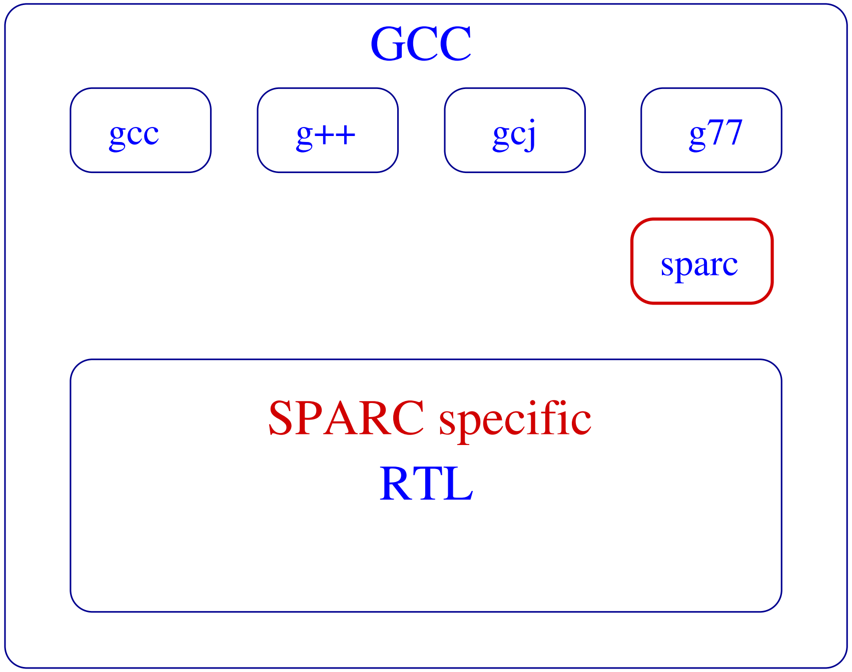
- Suppose that I want to generate Sparc code: `-target=sparc`
- I build GCC on my laptop: `-build=i586`
- and I run the compiler on my laptop: `-host=i586`

```
../gcc/configure -target=sparc -build=i586
-host=i586
```

Exemple: cross-compilation



Exemple: cross-compilation



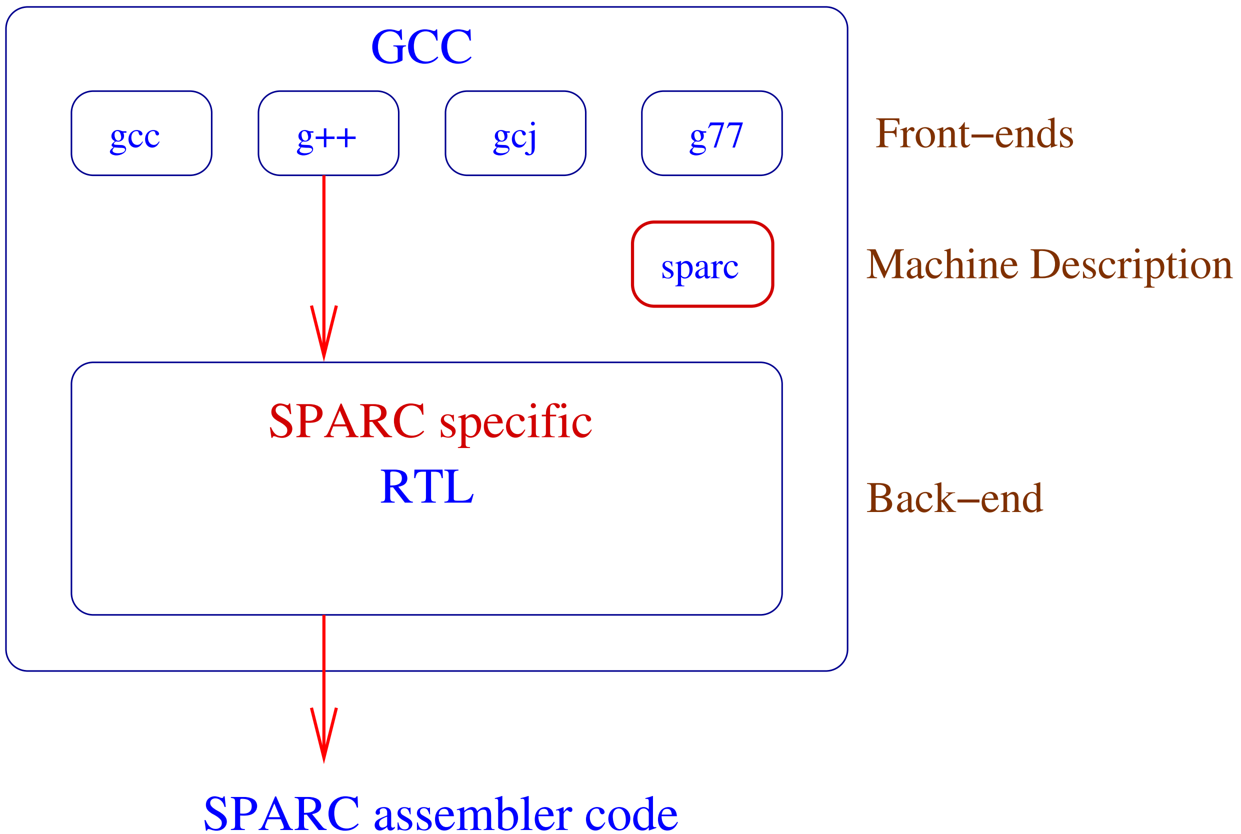
Front–ends

Machine Description

Back–end

1. Select SPARC machine description

Exemple: cross-compilation



1. Select SPARC machine description

2. Compile

RTL Optimizations

- An optimization pass optimizes all front-ends.

RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.

RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
- Types and memory structures after lowering to RTL contain less information.

  Memory accesses are under their canonical form:

  <start adress + offset>

RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
- Types and memory structures after lowering to RTL contain less information.

  Memory accesses are under their canonical form:

  <start adress + offset>
- Idea: we'd like to have

RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
- Types and memory structures after lowering to RTL contain less information.

  Memory accesses are under their canonical form:

  <start adress + offset>
- Idea: we'd like to have
  - architecture independent optimizations.

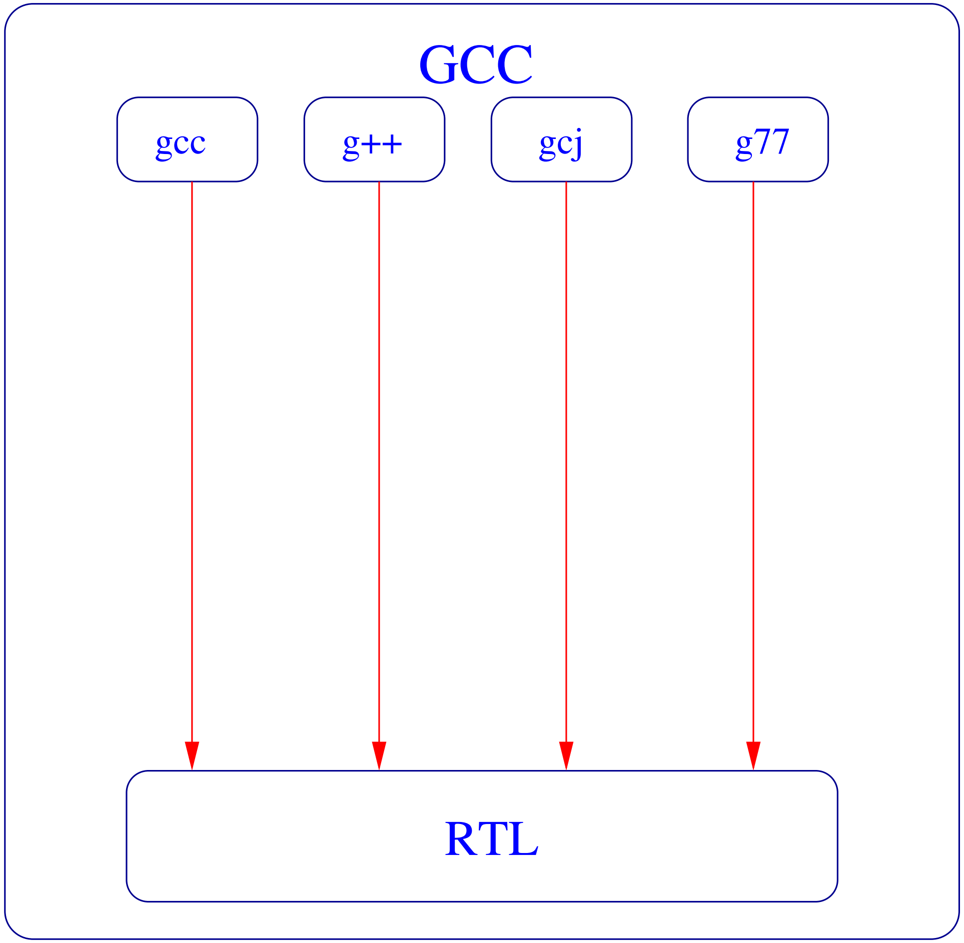
RTL Optimizations

- An optimization pass optimizes all front-ends.
- Machine dependent optimizations.
- Types and memory structures after lowering to RTL contain less information.

  Memory accesses are under their canonical form:

  <start adress + offset>
- Idea: we'd like to have
  - architecture independent optimizations.
  - on high level representations.

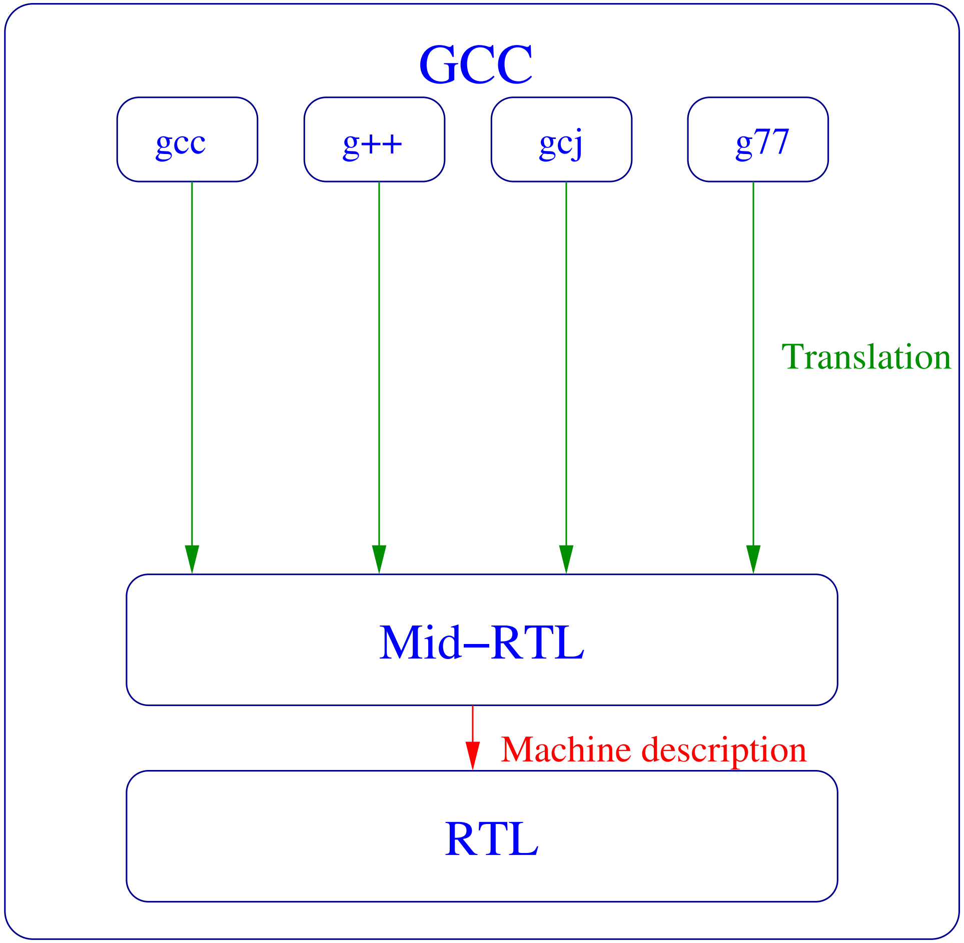
Intermediate Representations



Translation follows machines specificities

Machine description

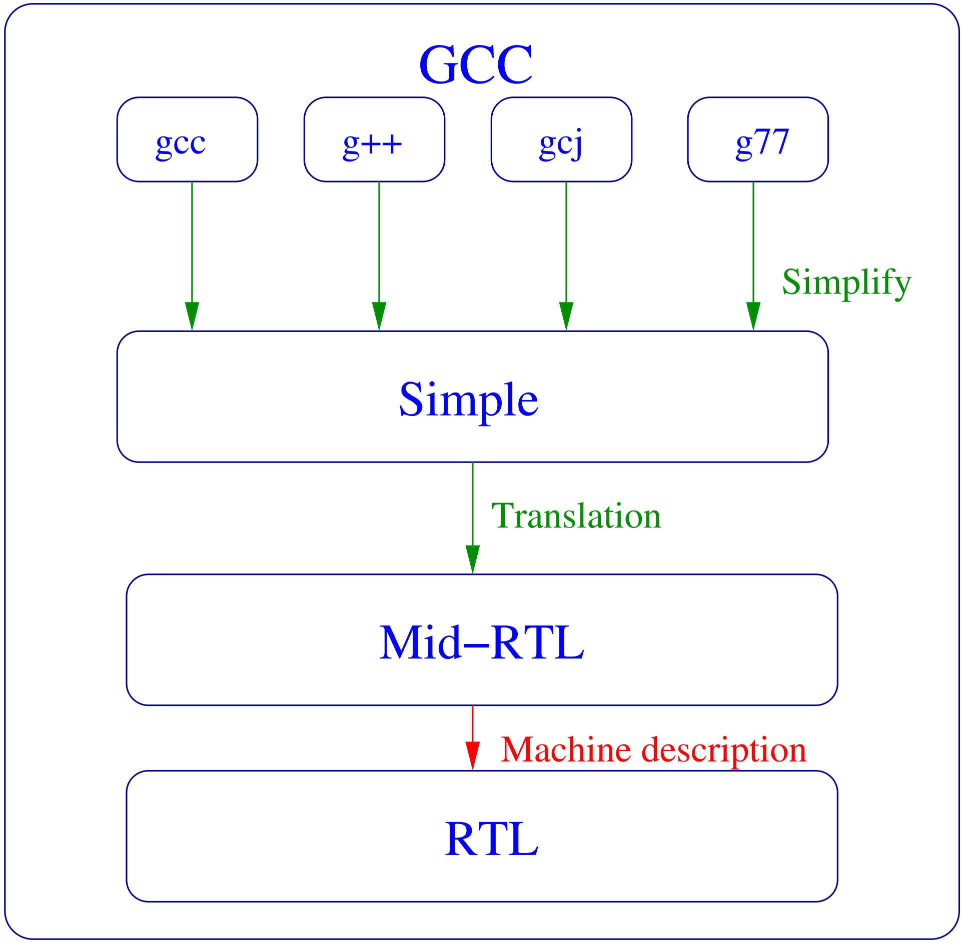
Intermediate Representations



Progressive transition from AST to RTL

Architecture independent IR

Intermediate Representations



Imperative Normal Form

Language independent representation

Progressive transition from AST to RTL

Architecture independent IR

Abstract Syntax Trees

- Simple linked list for statement nodes.

Abstract Syntax Trees

- Simple linked list for statement nodes.
- Manipulation of nodes through a macro interface: `TREE_CHAIN`, `TREE_OPERAND`, `TREE_CODE`, `...`

Abstract Syntax Trees

- Simple linked list for statement nodes.
- Manipulation of nodes through a macro interface: `TREE_CHAIN`, `TREE_OPERAND`, `TREE_CODE`, `...`
- Data structures hidden.

Abstract Syntax Trees

- Simple linked list for statement nodes.
- Manipulation of nodes through a macro interface: `TREE_CHAIN, TREE_OPERAND, TREE_CODE, ...`
- Data structures hidden.
- AST nodes are typed:
  allows tree-checking during development.

AST: example

```
a = (−−b) * 7;
x = y+z;
```

AST: example

AST: example

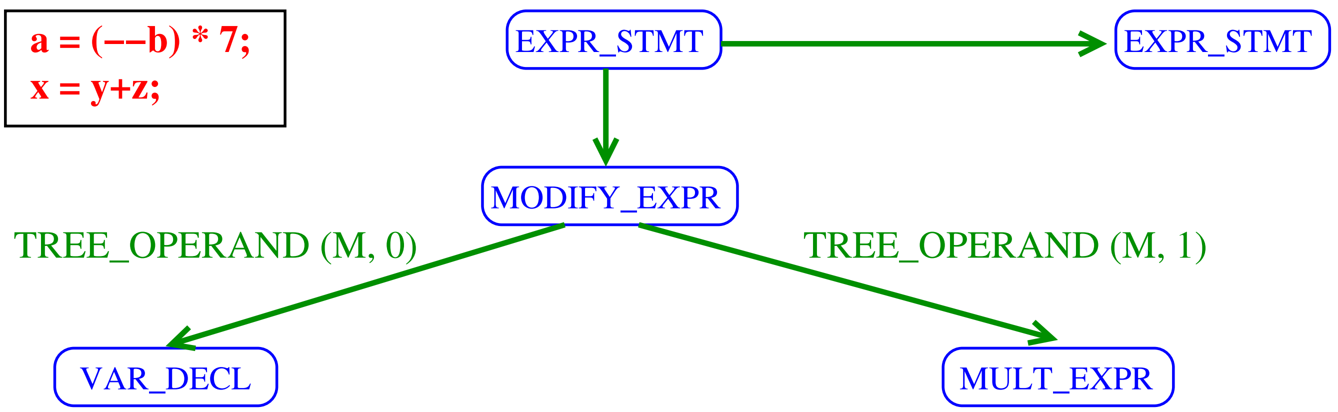
```
a = (−−b) * 7;
x = y+z;
```

AST: example

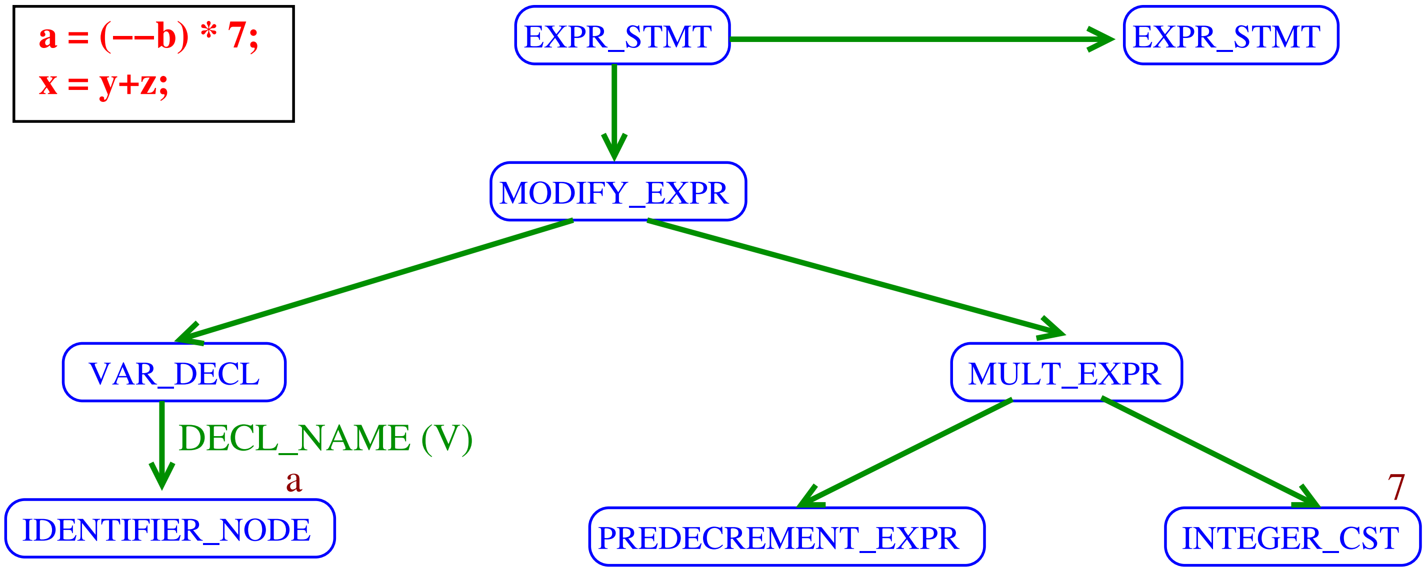
```
a = (−−b) * 7;
x = y+z;
```

AST: example

```
a = (−−b) * 7;
x = y+z;
```

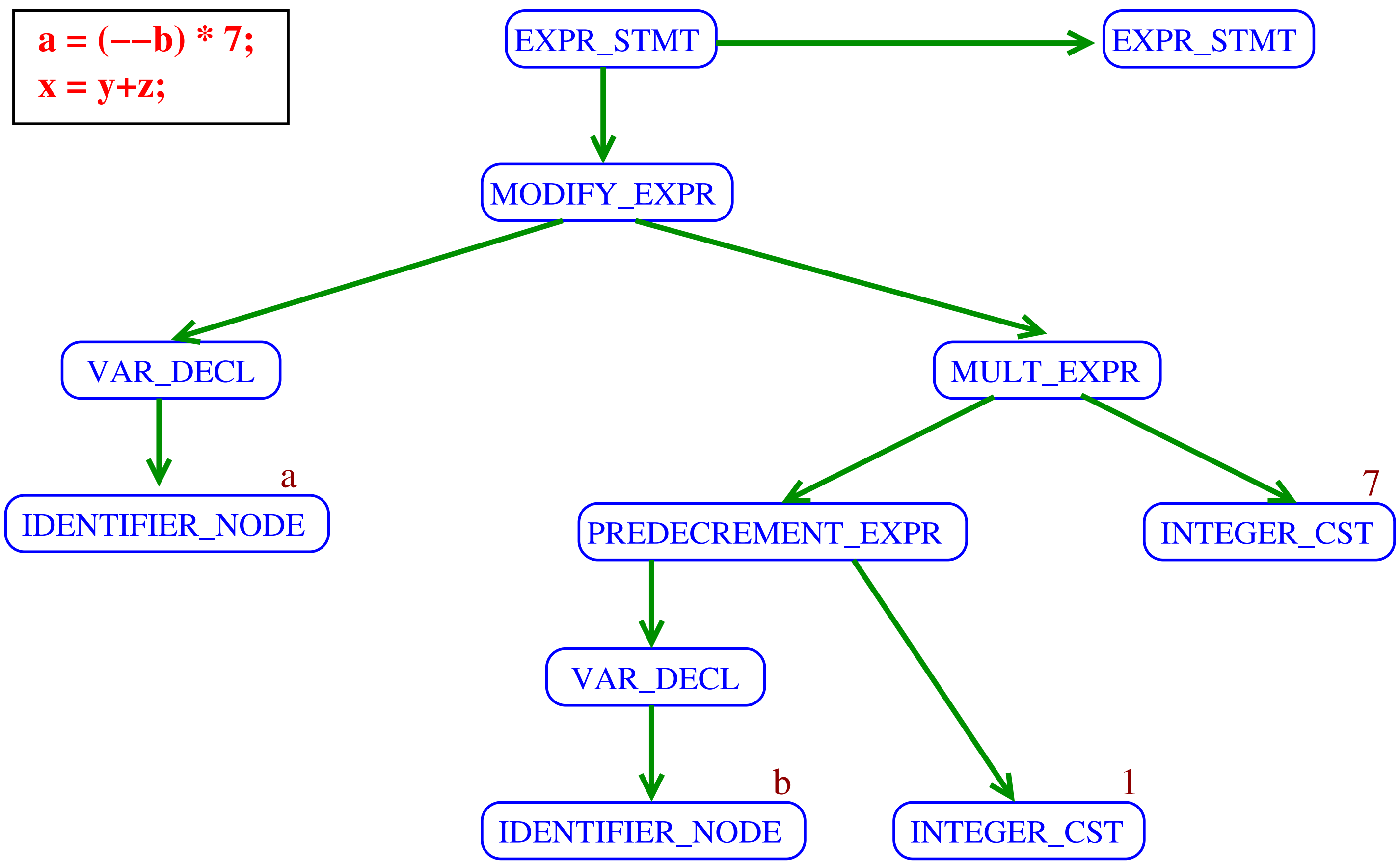


AST: example


a = (−−b) * 7;
x = y+z;
EXPR_STMT
EXPR_STMT
MODIFY_EXPR
VAR_DECL
MULT_EXPR
DECL_NAME (V)
a
7
IDENTIFIER_NODE
PREDECREMENT_EXPR
INTEGER_CST

AST: example

```
a = (−−b) * 7;
x = y+z;
```



Simple: overview

- SIMPLE's grammar defines an imperative normal form:

Simple: overview

- SIMPLE's grammar defines an imperative normal form:
  - Reduced number of expressions.

Simple: overview

- SIMPLE's grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.

Simple: overview

- SIMPLE's grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.
- SIMPLE AST has a regular structure.

Simple: overview

- SIMPLE’s grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.
- SIMPLE AST has a regular structure.
- Systematic AST analysis is possible.

Simple: overview

- SIMPLE's grammar defines an imperative normal form:
  - Reduced number of expressions.
  - Reduced number of control structures.
- SIMPLE AST has a regular structure.
- Systematic AST analysis is possible.
- Common intermediate representation for all front ends.

Simple: exemple

```
a = --b*7;
```

Simple: exemple

`a = --b*7;`	`b=b-1;` `a=b*7;`

Simple: exemple

`a = --b*7;`	`b=b-1;` `a=b*7;`

```
if (i++ && --k)
 {
   j=f(i+3*k);
 }
```

Simple: exemple

`a = --b*7;`	`b=b-1;` `a=b*7;`

<pre>if (i++ && --k)
 {
   j=f(i+3*k);
 }</pre>
	<pre>if (i)
	  {
	   k=k-1;
	   if(k)
	     {
	      i=i+1;
	      T1=3*k;
	      T2=i+T1;
	      j=f(T2);
	     }
	   else
	     i=i+1;
	  }
	 else
	  i=i+1;</pre>

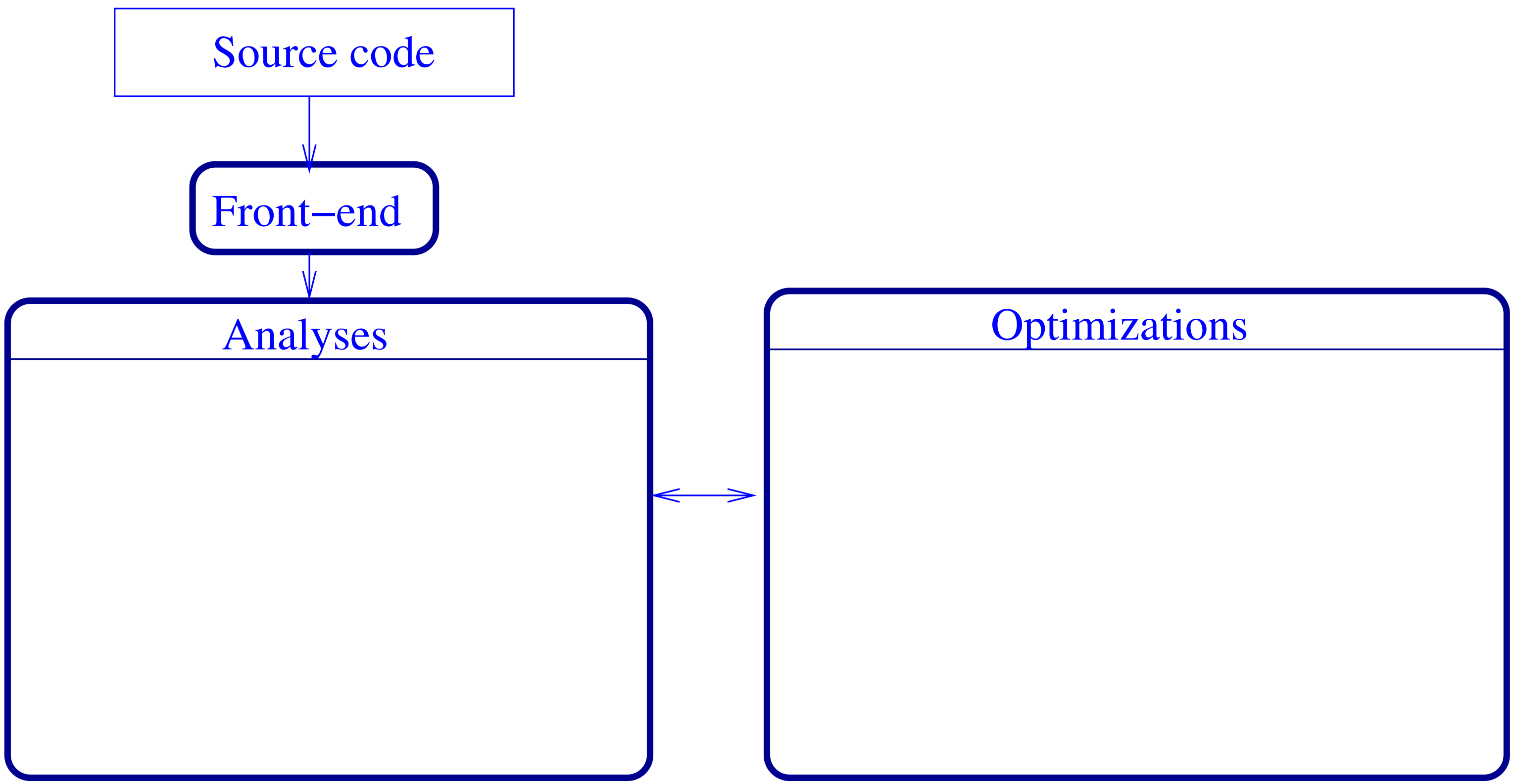
Simple: exemple

```
while(i++ && --k)
 {
  A[i]=A[i+3*k];
 }
```

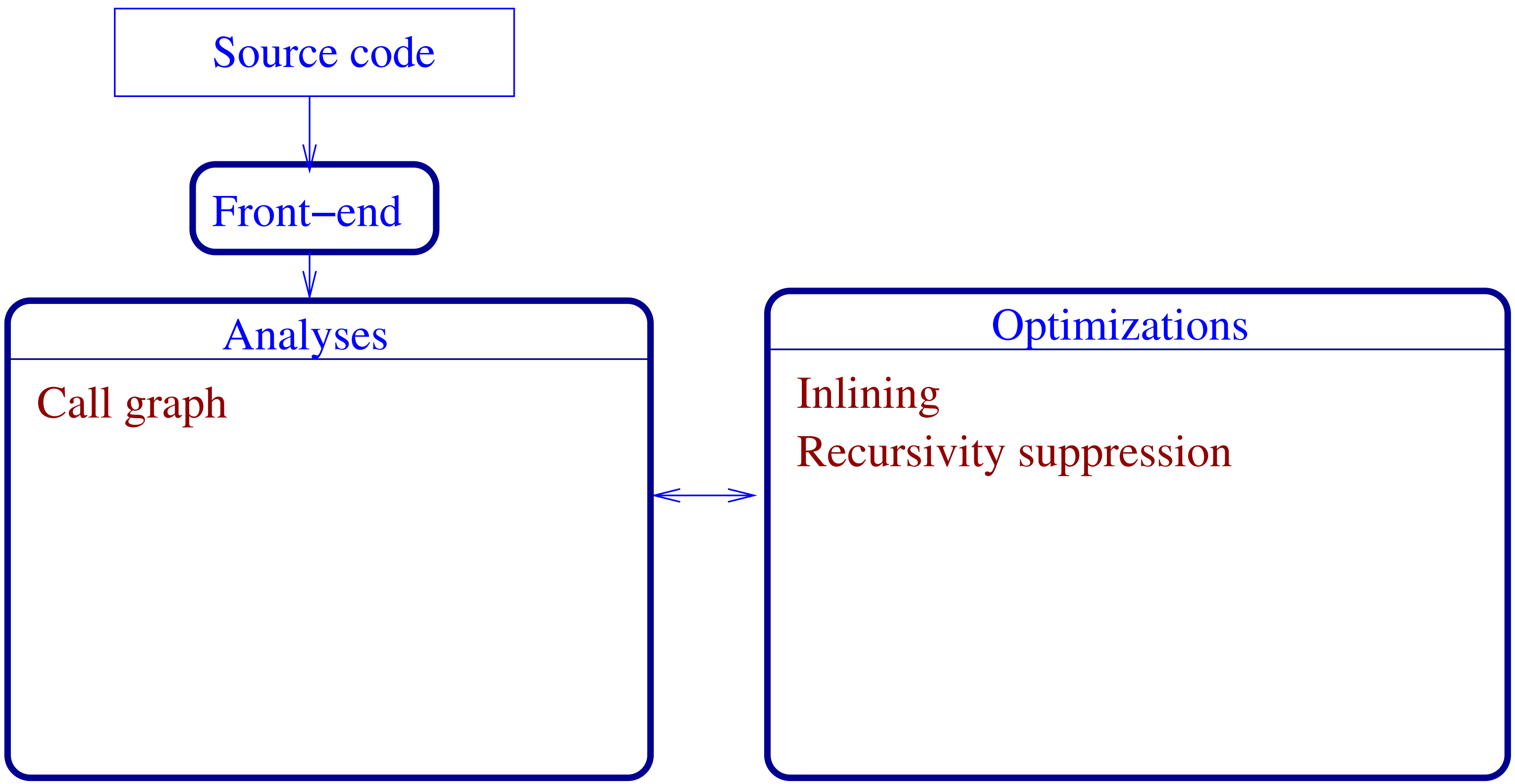
Simple: exemple

`while(i++ && --k)`
` {`
`  A[i]=A[i+3*k];`
` }`
	`if(i)`
	` {`
	`  k=k-1;`
	`  if (k)`
	`   while(1)`
	`    {`
	`     i=i+1;`
	`     T1=3*k;`
	`     T2=i+T1;`
	`     A[i]=A[T2];`
	`     if(i)`
	`      {`
	`       k=k-1;`
	`       if(k)`
	`        i=i+1;`
	`       else`
	`        break;`
	`      }`
	`     else`
	`      break;`
	`    }`
	` }`
	`i=i+1;`

An optimizing compiler



An optimizing compiler



Call Graph

- (node, edge) => (declaration, call)

Call Graph

- (node, edge) => (declaration, call)
- Graph representation:

Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.

Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.
  - in a file under parenthesized form: EXP-Space.

Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.
  - in a file under parenthesized form: EXP-Space.
- Use metrics for controlling inlining.

Call Graph

- (node, edge) => (declaration, call)
- Graph representation:
  - pointers: P-Space.
  - in a file under parenthesized form: EXP-Space.
- Use metrics for controlling inlining.
- GCC's analysis is limited to a single translation unit.

Call Graph : solution

- Perform call graph optimizations outside GCC.

Call Graph : solution

- Perform call graph optimizations outside GCC.
- Problems :

Call Graph : solution

- Perform call graph optimizations outside GCC.
- Problems :
  - Extract information, decide, then apply optimizations: 3 passes.

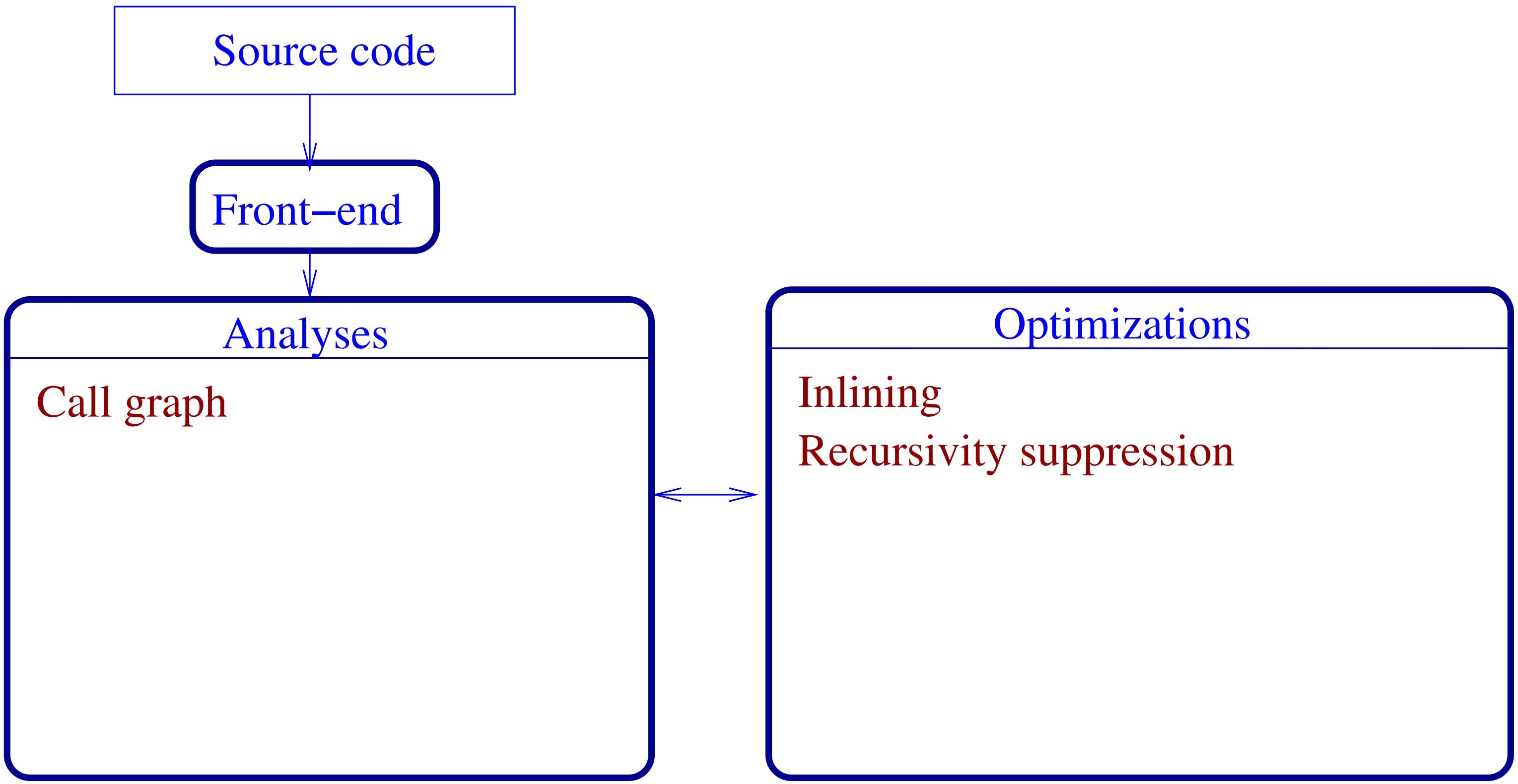
Call Graph : solution

- Perform call graph optimizations outside GCC.
- Problems :
  - Extract information, decide, then apply optimizations: 3 passes.
  - Knowledge base's size.

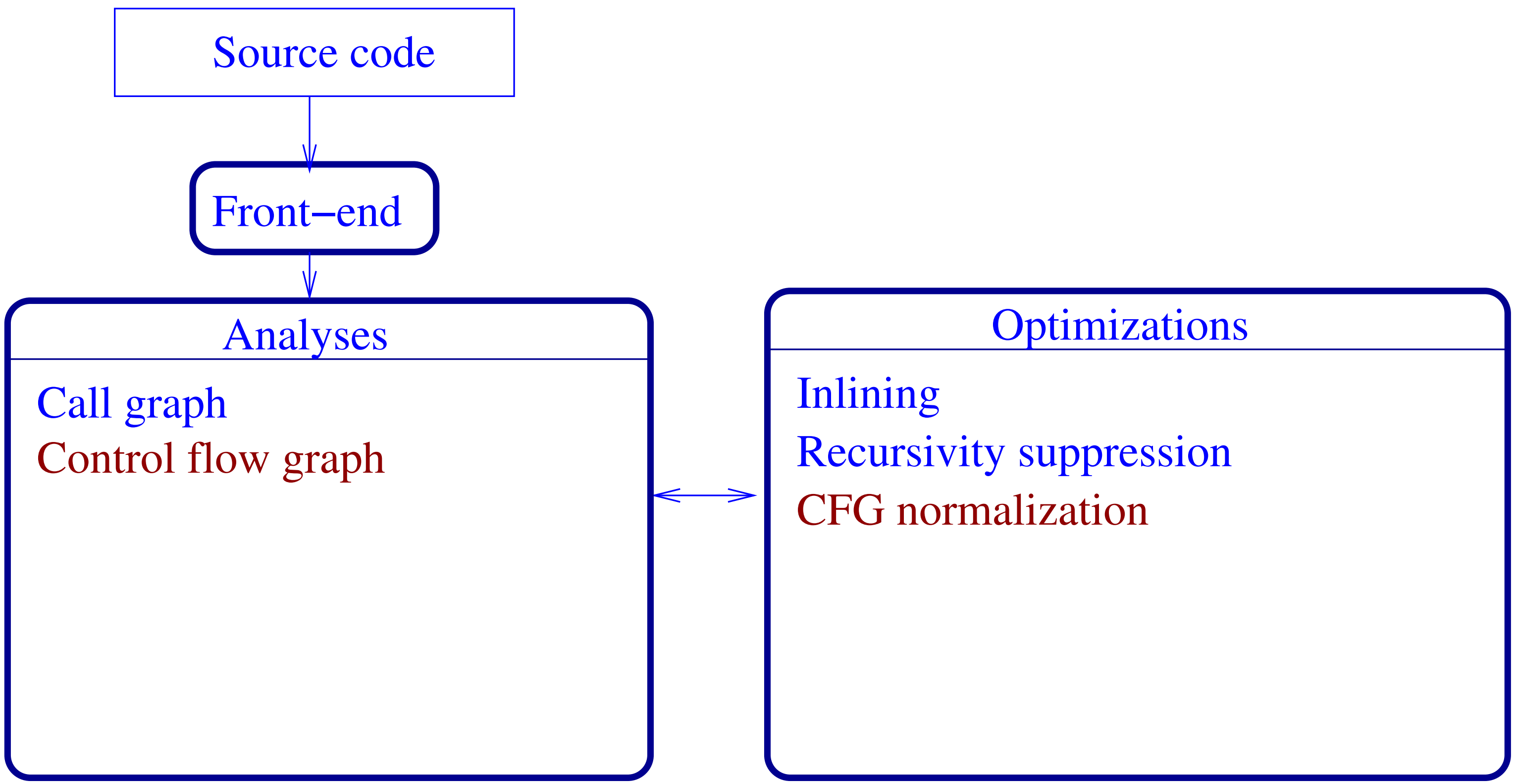
Call Graph : solution

- Perform call graph optimizations outside GCC.
- Problems :
  - Extract information, decide, then apply optimizations: 3 passes.
  - Knowledge base's size.
  - What informations to be stored in KB?

An optimizing compiler



An optimizing compiler



CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.

CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.

CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?

CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?
    - It is difficult to optimize programs containing gotos.

CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?
  - It is difficult to optimize programs containing gotos.
  - Break and continue translation to RTL generates gotos.

CFG Normalization

- Suppress irregularities from control flow: goto, break, continue.
- CFG normalization is based on Simple.
- Why normalizing CFG?
  - It is difficult to optimize programs containing gotos.
  - Break and continue translation to RTL generates gotos.
  - Simplification generates irregular code.

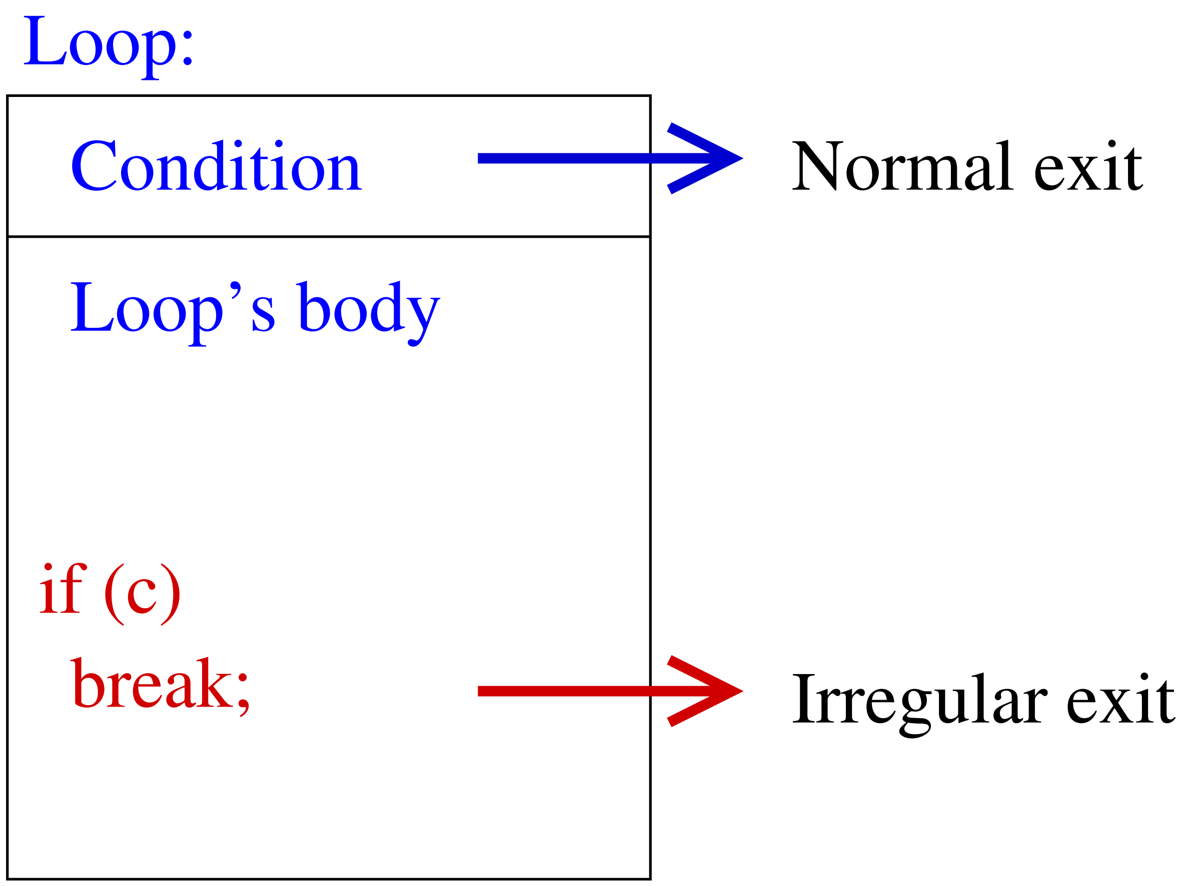
Flow Out

Loop:

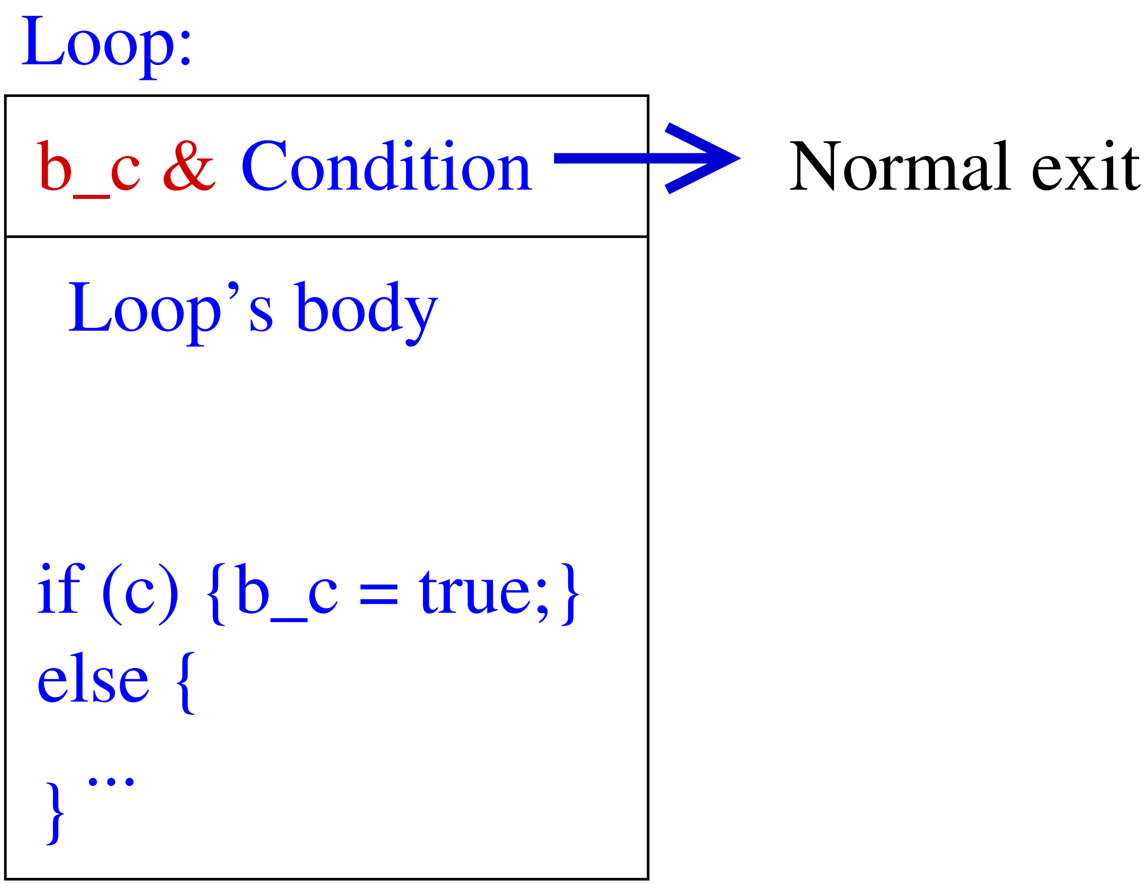
Condition

Loop's body

Flow Out


Loop:
Condition
Normal exit
Loop's body
if (c)
break;
Irregular exit

Flow Out



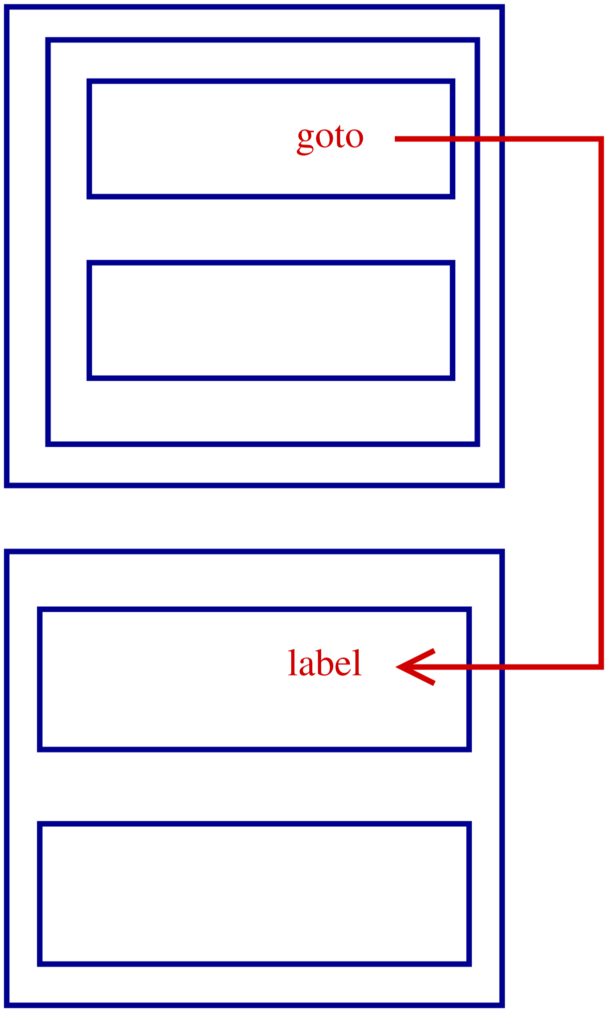
Break Elimination

```
while (a)
  {
    stmt1;
    if (b)
      break;
    stmt2;
  }
```

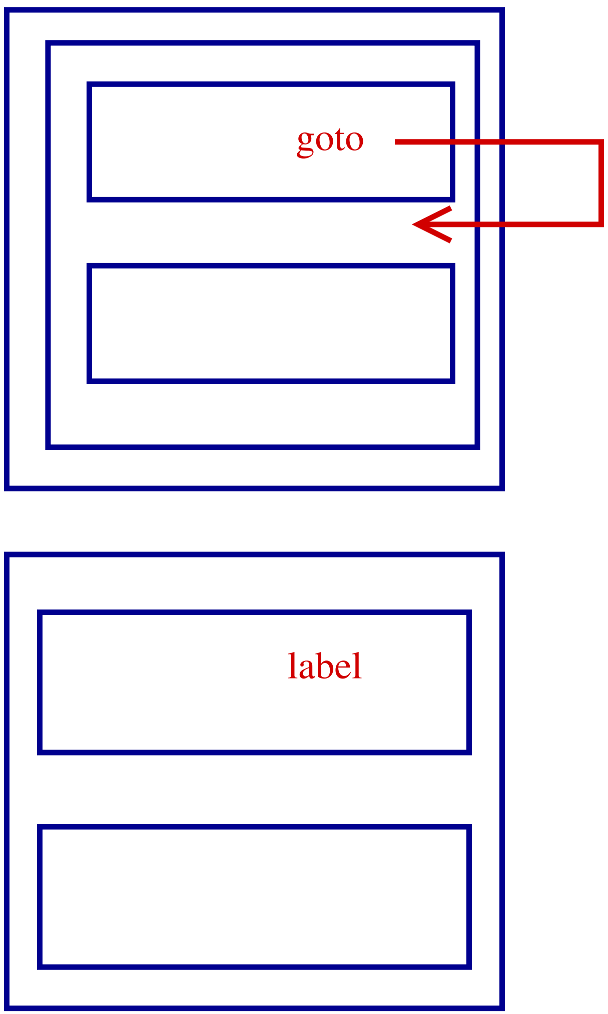
Break Elimination

```
int c_b = 0;
while (c_b == 0 && a)
  {
    stmt1;
    if (b)
      {c_b = 1;}
      else
        {
          stmt2;
        }
  }
```

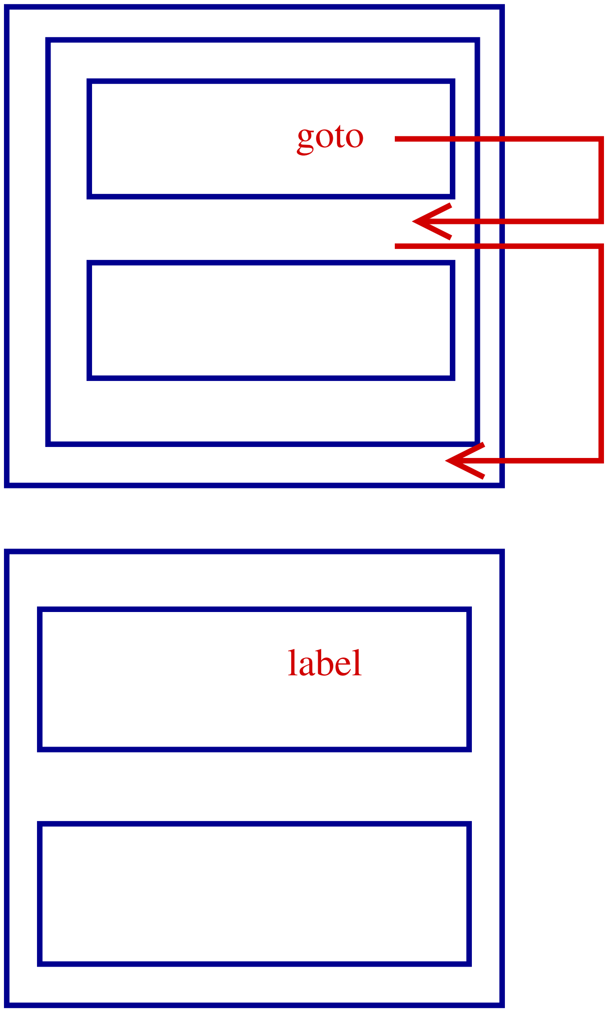
Goto Elimination



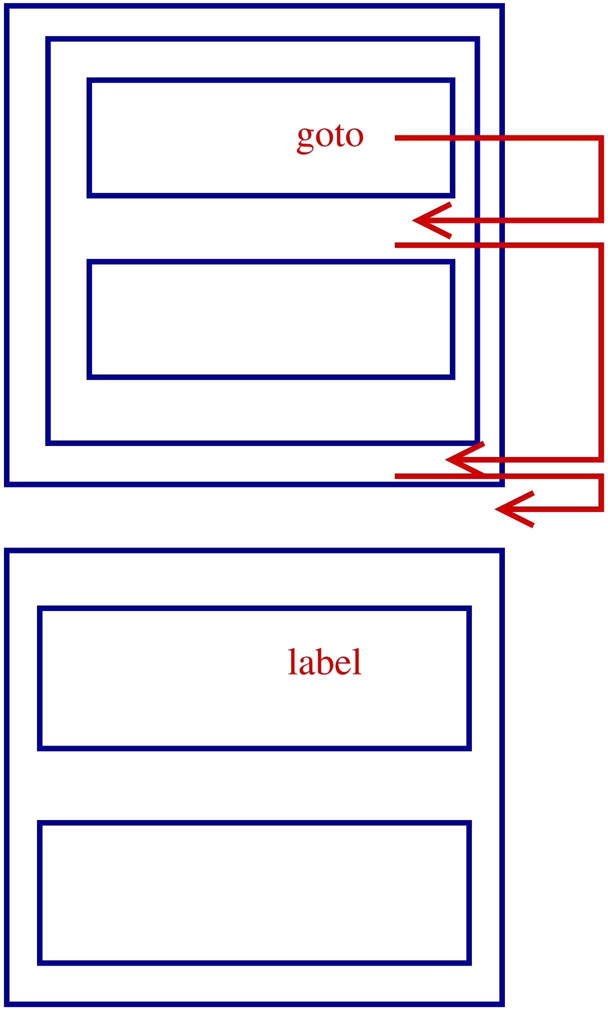
Goto Elimination



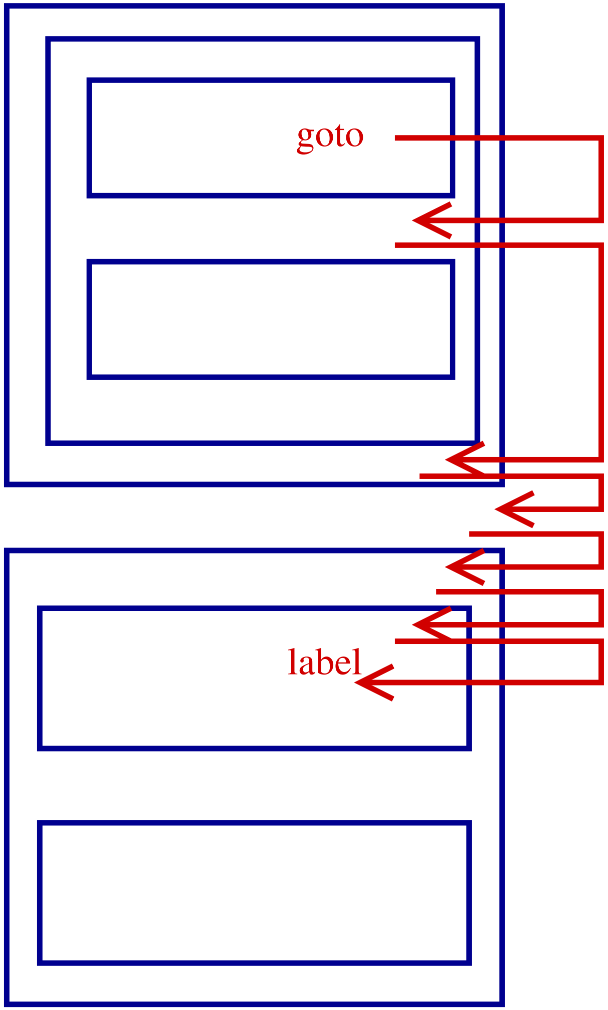
Goto Elimination



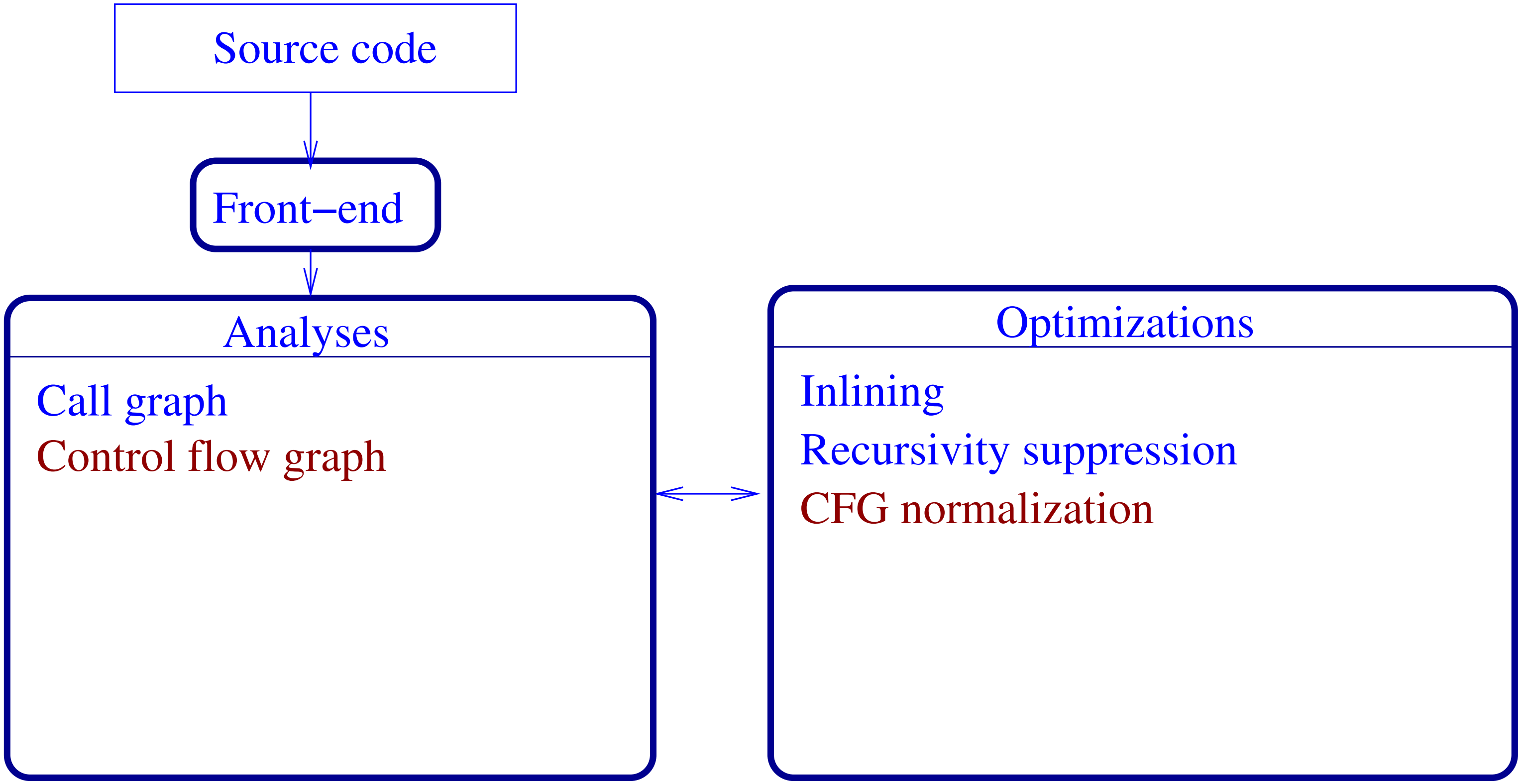
Goto Elimination



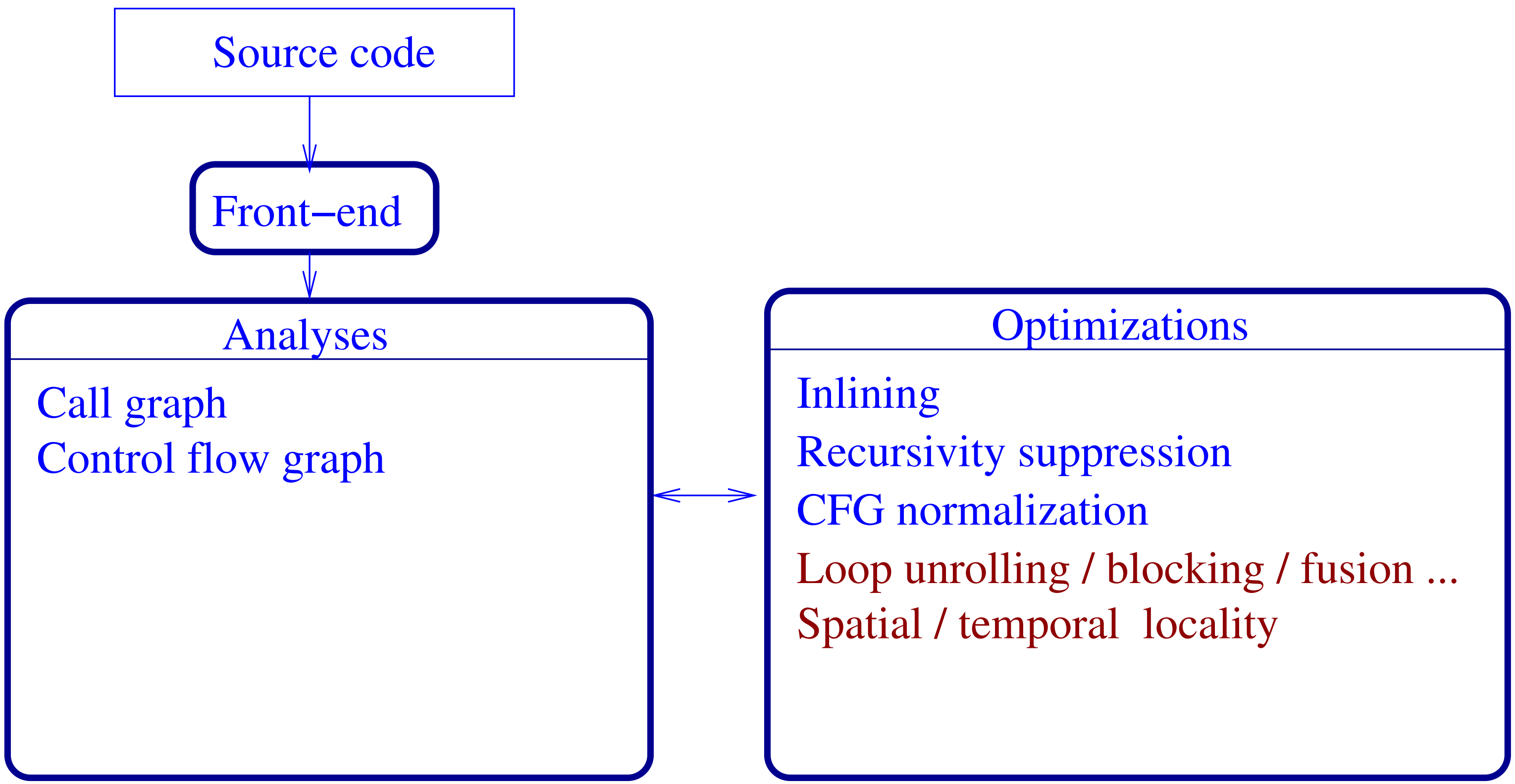
Goto Elimination



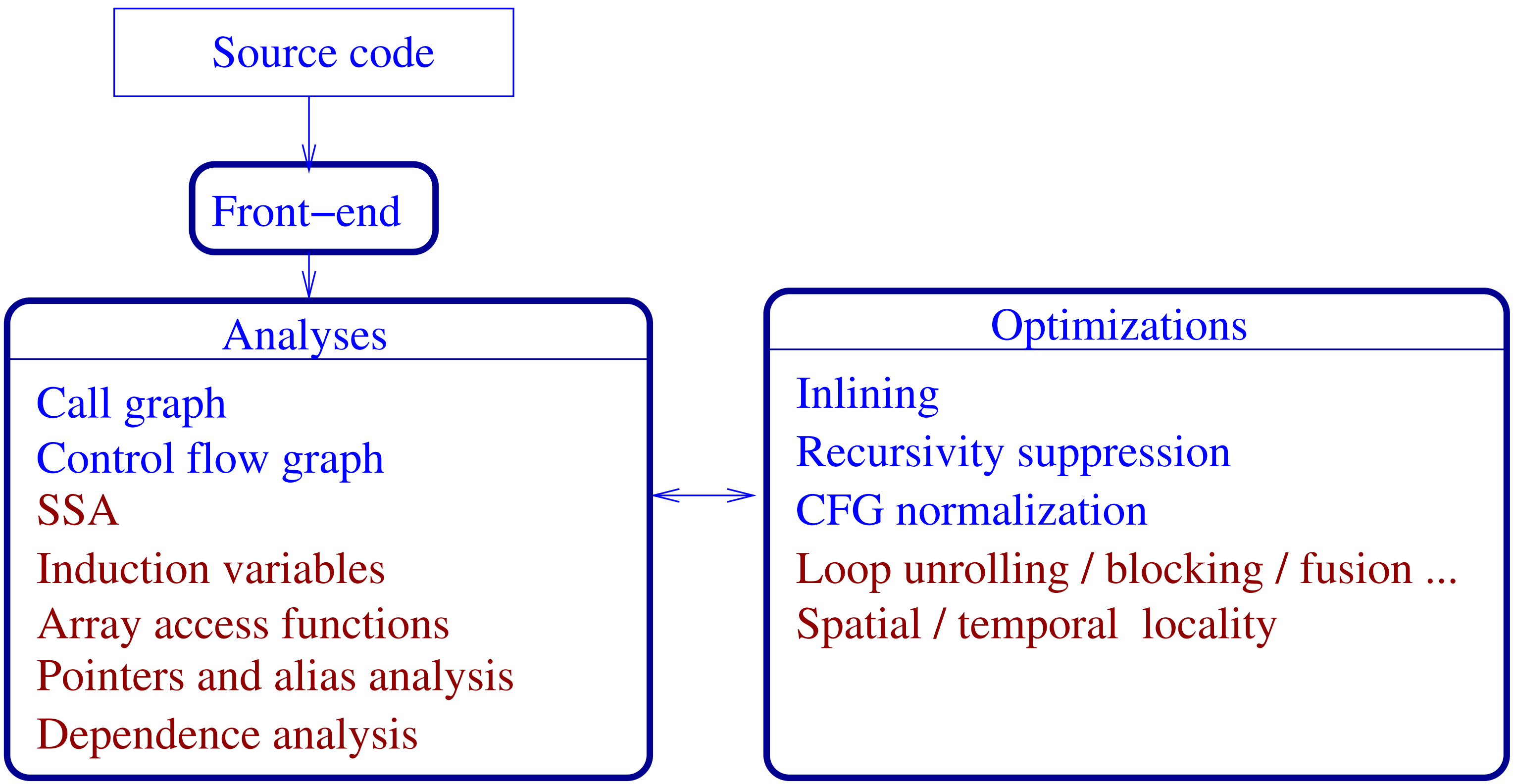
An optimizing compiler



An optimizing compiler



An optimizing compiler



Loop Optimizations

- Loops are normalized after detection of induction variables.

Loop Optimizations

- Loops are normalized after detection of induction variables.
- Geometric representation of array accesses can be then constructed.

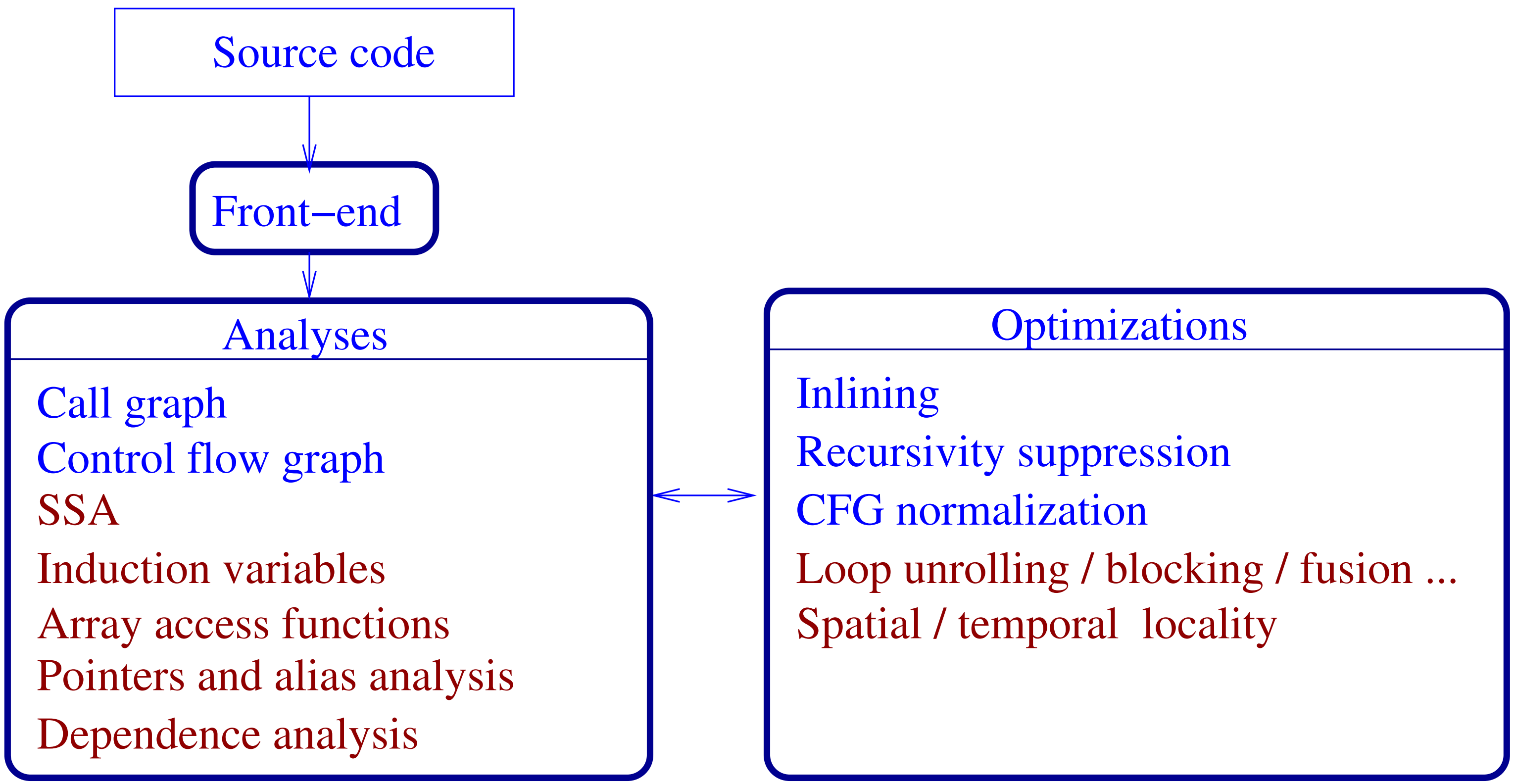
Loop Optimizations

- Loops are normalized after detection of induction variables.
- Geometric representation of array accesses can be then constructed.
- Dependence analysis is necessary for validating loop transformations.

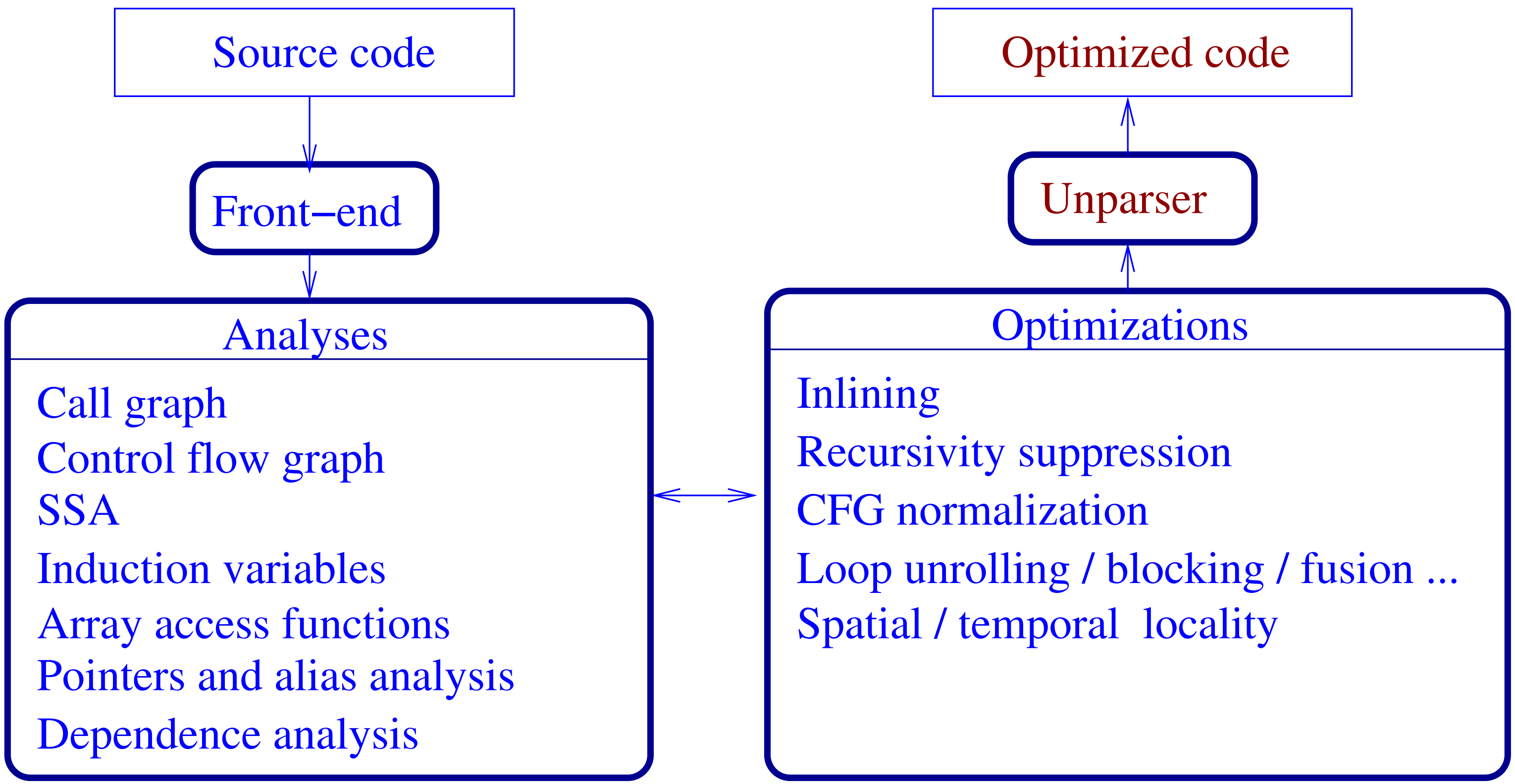
Loop Optimizations

- Loops are normalized after detection of induction variables.
- Geometric representation of array accesses can be then constructed.
- Dependence analysis is necessary for validating loop transformations.
- These points are still under development.

An optimizing compiler



An optimizing compiler



Remerciements

Merci à tous ceux qui ont contribué à la réussite de ce projet :

Remerciements

Merci à tous ceux qui ont contribué à la réussite de ce projet :

- l'équipe ICPS pour l'excellente ambiance,
- Philippe Clauss, Vincent Loechner et Benoît Meister pour leur travail de recherche,
- Catherine Mongenet pour le cours de compil,
- Frédéric Wagner et Diego Novillo pour m'avoir accompagné dans ce projet,
- the FSF for GCC, Debian, Linux and Prosper
- et ma famille.