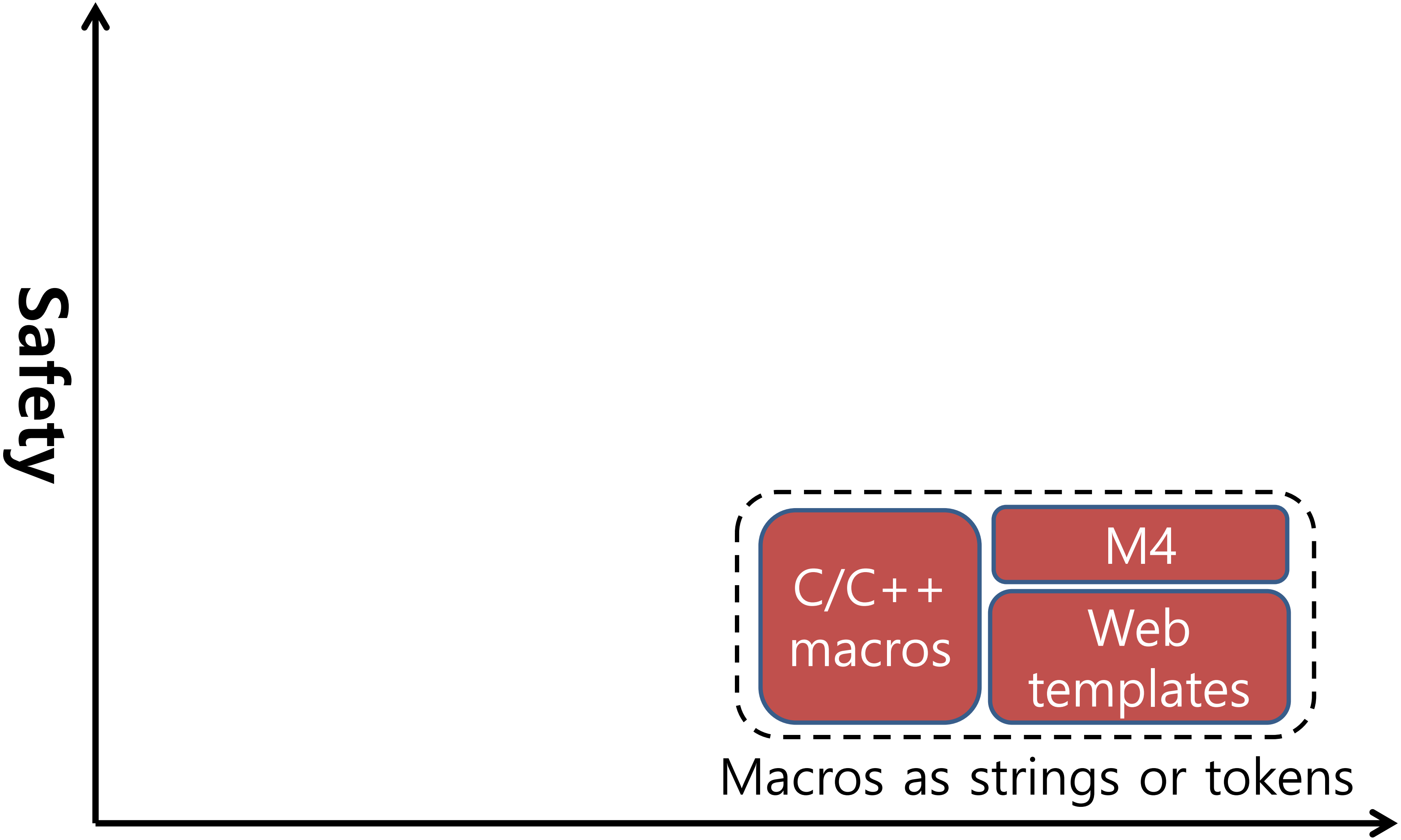
Marco: Safe, Expressive Macros for Any Language

Byeongcheol Lee
Robert Grimm
Martin Hirzel
Kathryn S. McKinley

Macros in programming languages

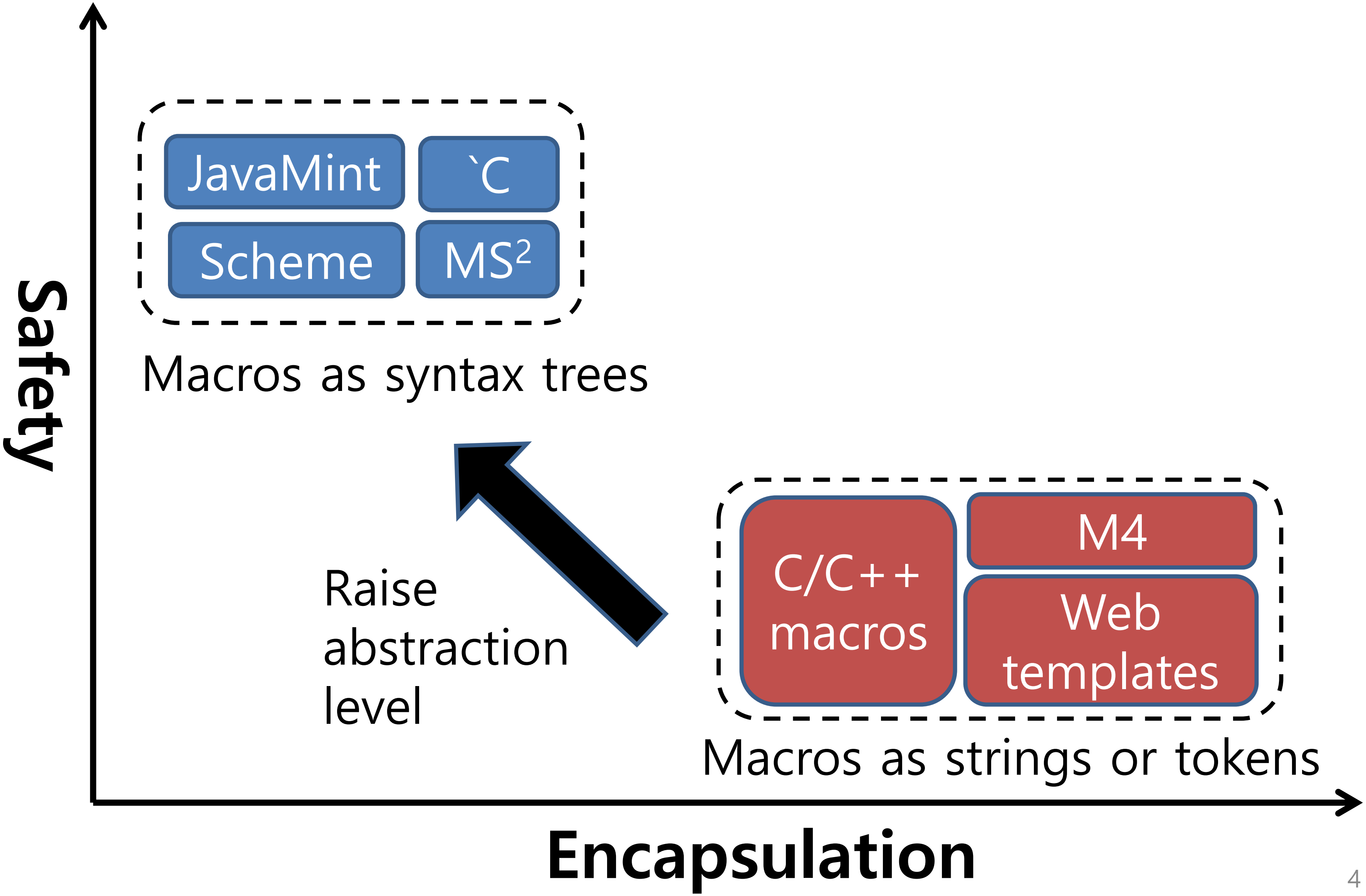
- Abstraction
  - Simple, elegant core languages
  - Macros in C and Scheme
- Language interoperability
  - Target-language code as host-language data
  - Web templates for HTML and SQL code

Unsafe macros for any Language

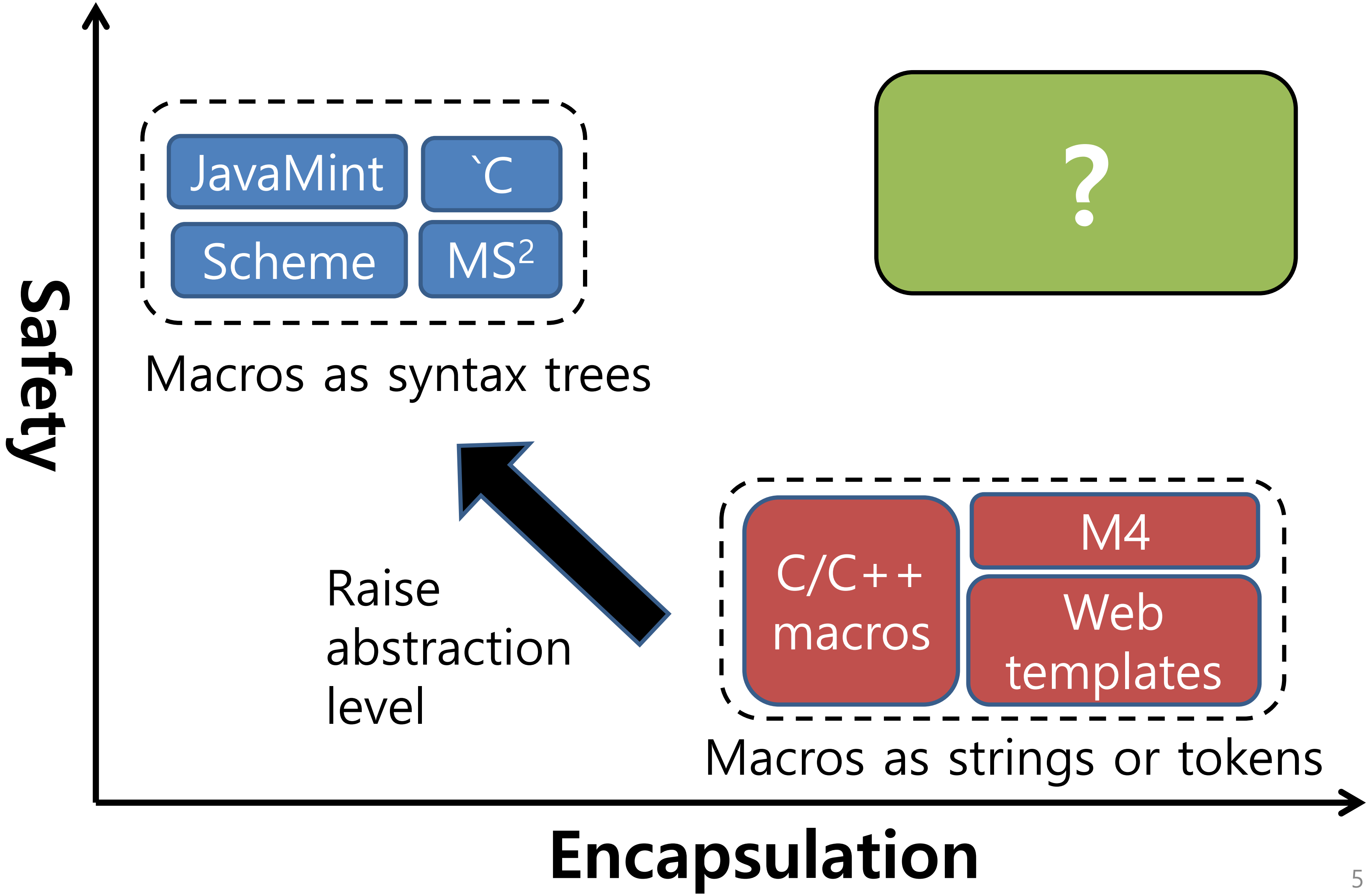


Encapsulation

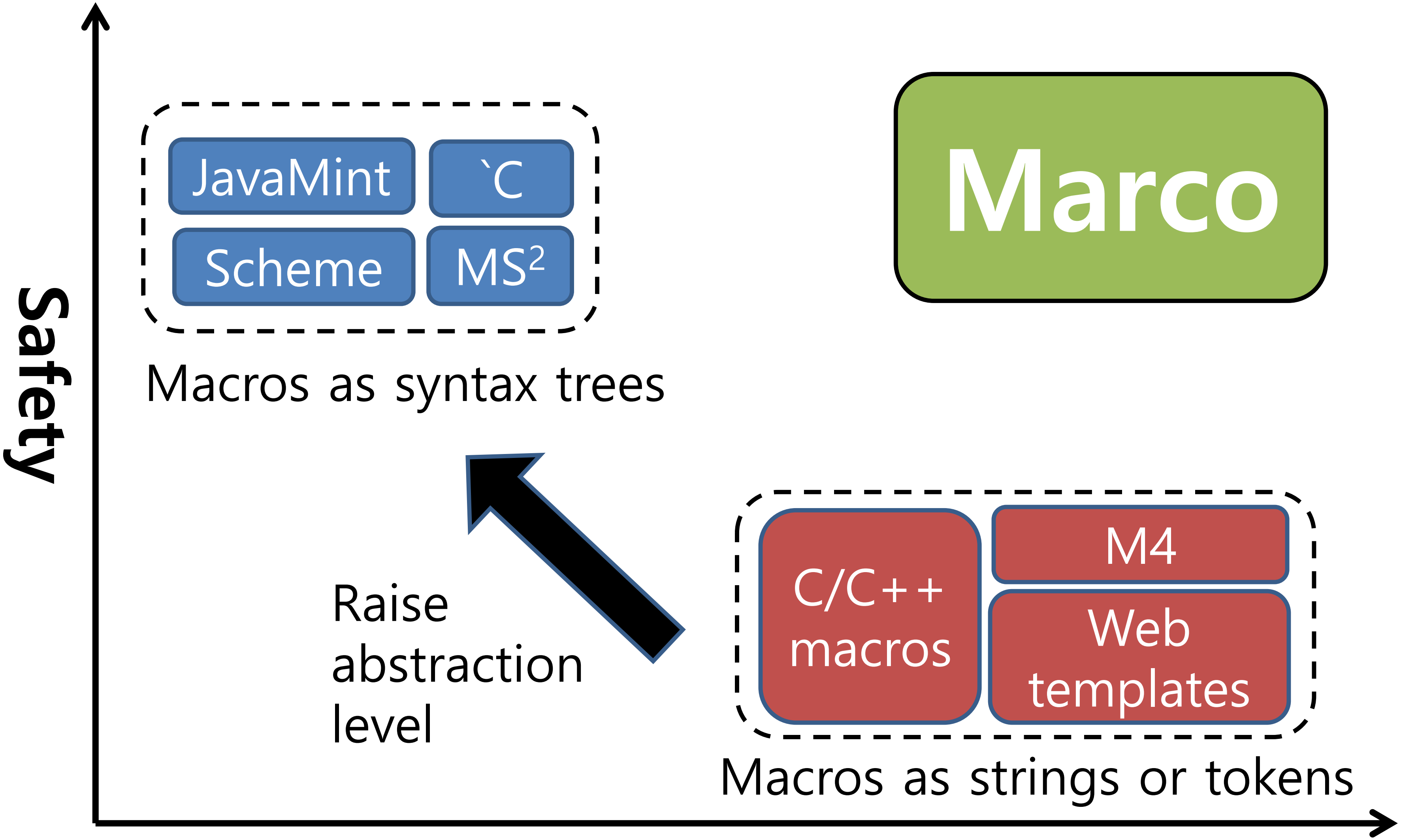
Safe macros for one language



Safe macros for any Language

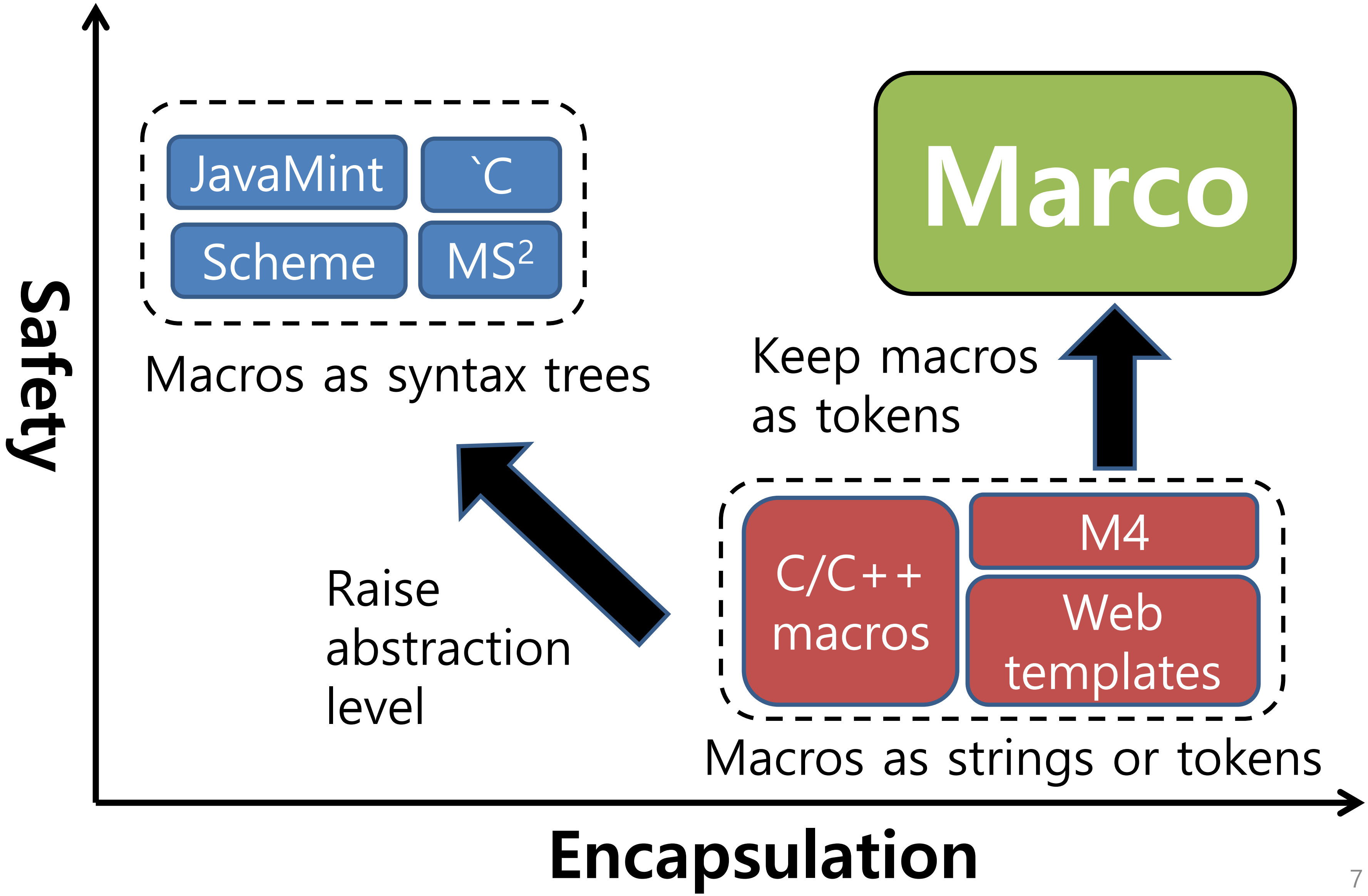


Marco: safe macros for any Language



Encapsulation

Marco: safe macros for any Language



Safety

JavaMint

`C

Scheme

MS²

Macros as syntax trees

Marco

Keep macros as tokens

Raise abstraction level

C/C++ macros

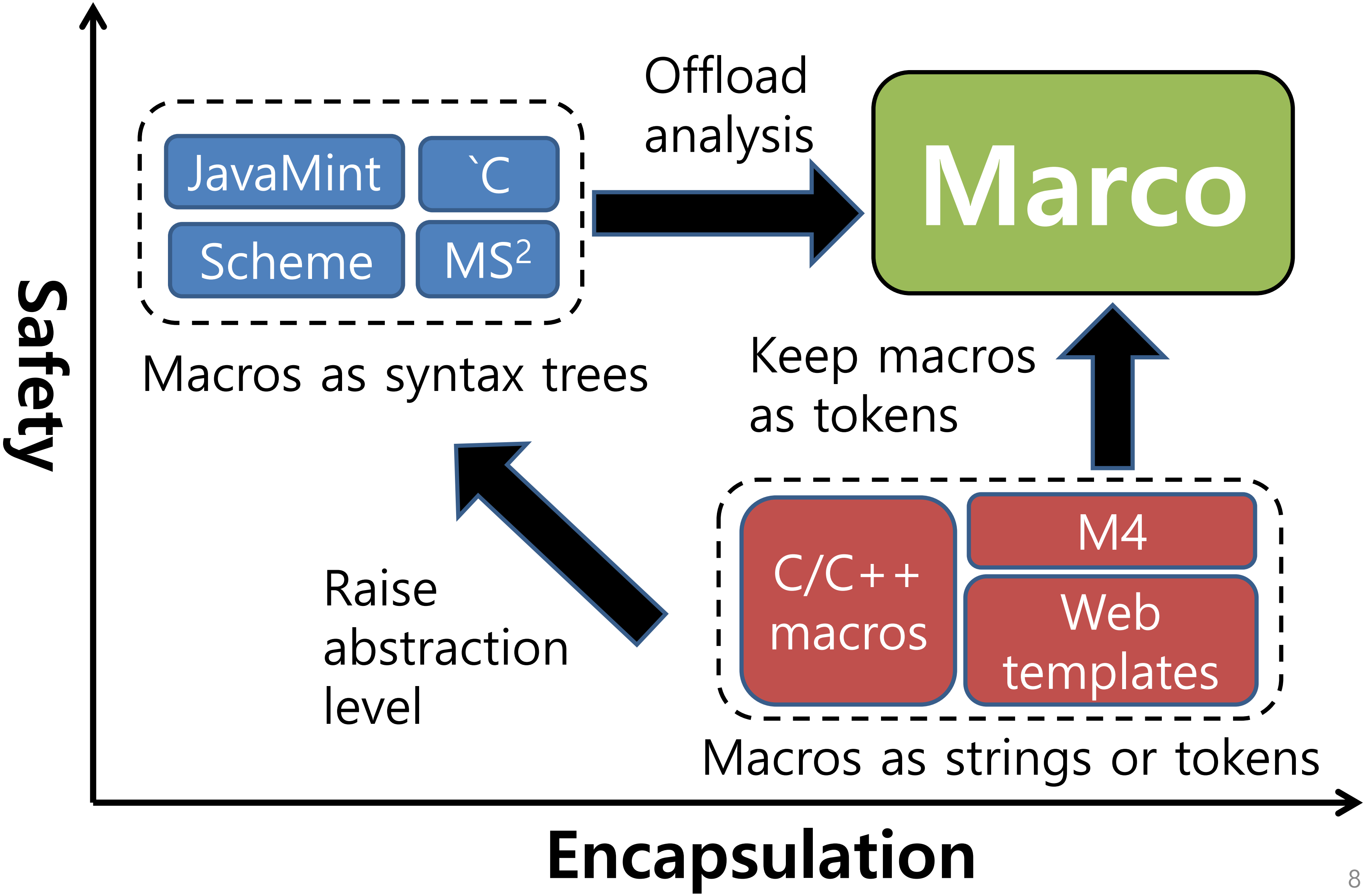
M4

Web templates

Macros as strings or tokens

Encapsulation

Marco: safe macros for any Language



Outline

- Introduction
- Marco language and architecture
  - Expressing macros as Tokens
  - Offloading analysis using oracle queries
- Oracle analysis in practice
- Summary

Expressing macros as tokens

```
#define swap(x, y) {
    int temp = x;
    x = y;
    y = temp;
}
```

C/C++ macro

```
code<cpp,stmt> swap(
  code<cpp,id> x,
  code<cpp,id> y) {
  return `cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }]; }
```

Marco macro

- **Static typing**
  - **code types parametized by language and category**
  - **code<cpp,stmt> and `cpp(stmt) for C++ statement**
- **Explicit blanks**

Multilingual macros in Marco

```
code<cpp,stmt> swap(
  code<cpp,id> x,
  code<cpp,id> y) {
  return `cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }]; }
```

C++

```
code<sql,stmt> select(
  code<sql,expr> cond)
{
  return `sql(stmt) [
   select names
     from employees
     where $cond
  ]; }
```

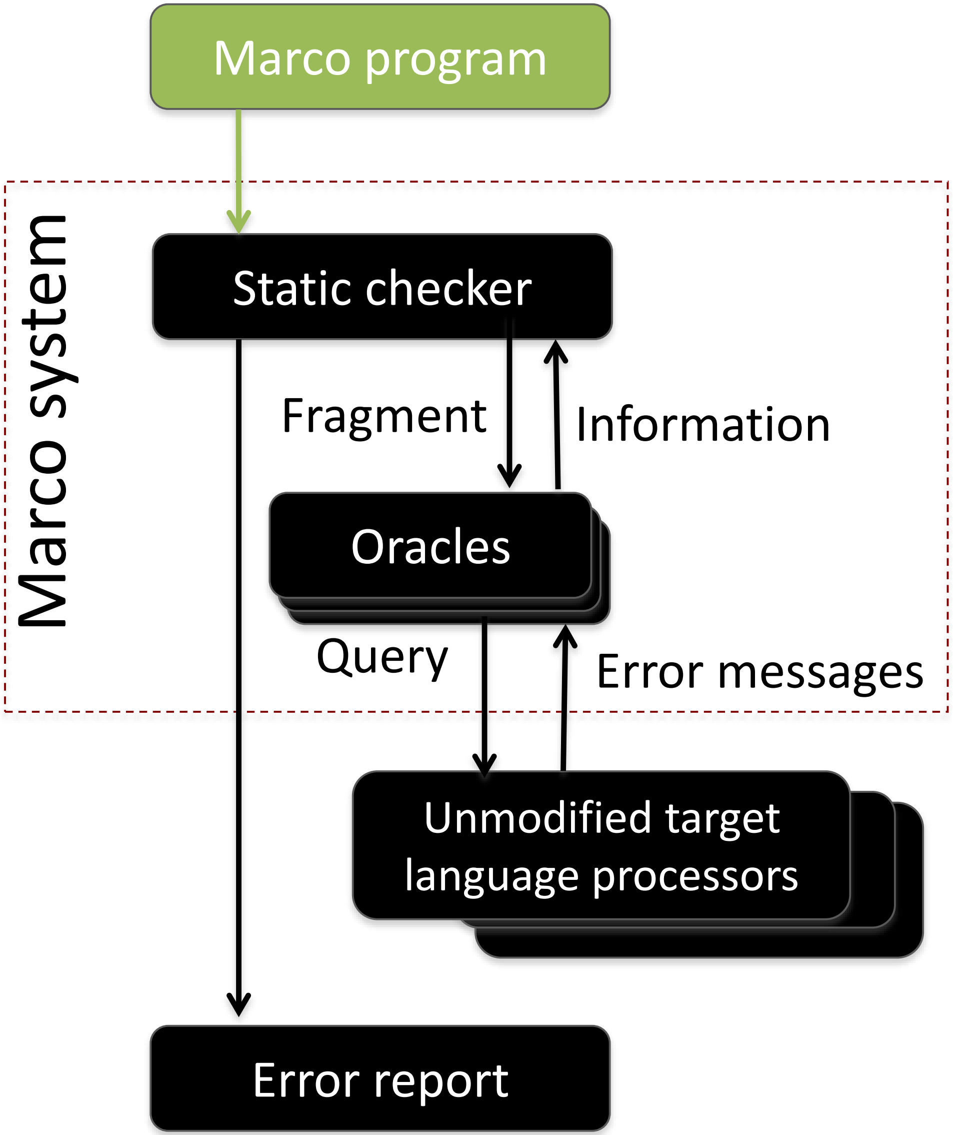
SQL

- Scannerless, extensible parser in ***Rats!***
- **`cpp** selects a C++ lexical analyzer
- **`sql** selects an SQL lexical analyzer

Offloading analysis

```
code<cpp,stmt>
swap(
  code<cpp,id> x,
  code<cpp,id> y) {
  return `cpp(stmt)
[ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }];
}
```

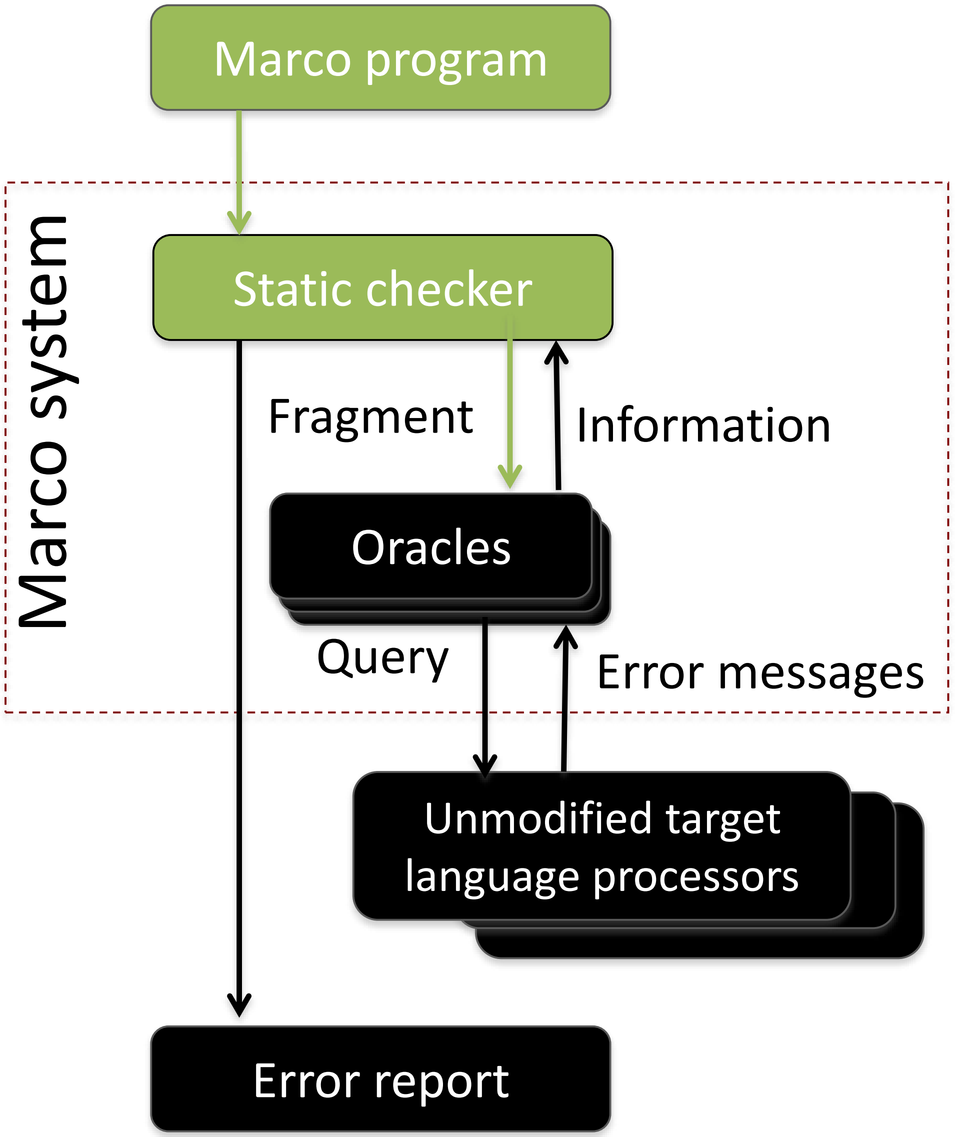
Marco program



Offloading analysis

```
`cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }]
```

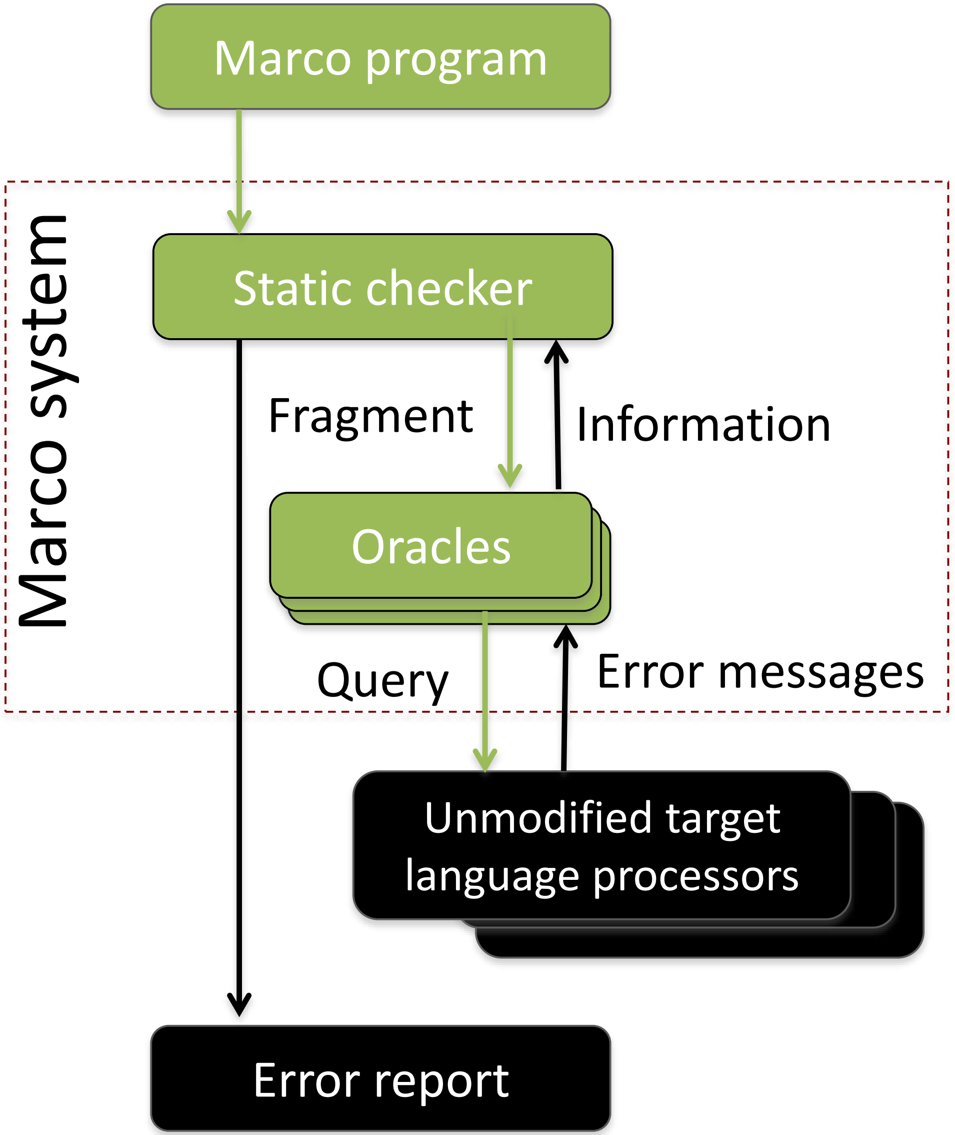
Fragment



Offloading analysis

```
`cpp(stmt) [ {
  int temp = _id0_;
  _id1_ = _id2_;
  _id3_ = temp;
}]
```

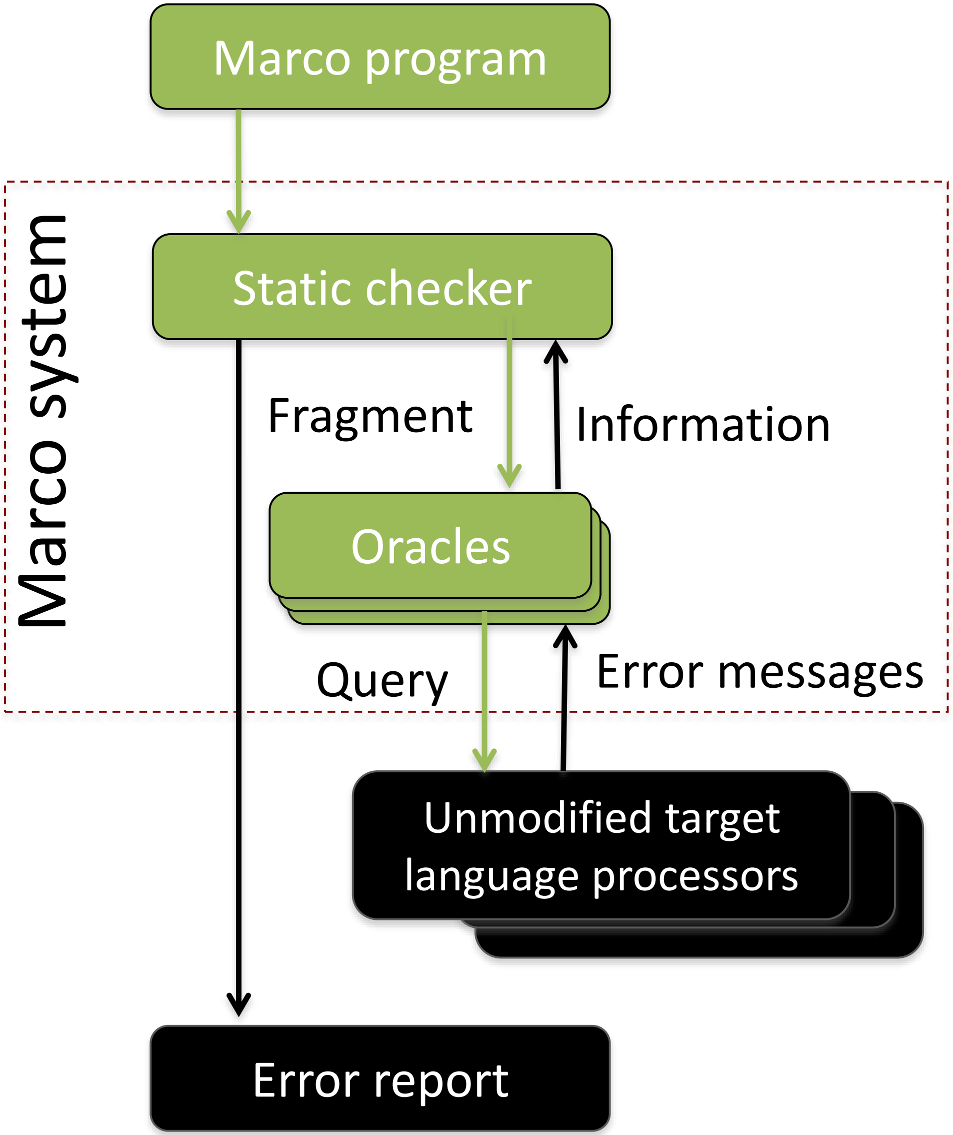
Fragment with concretized blanks



Offloading analysis

```
void _id4_() {
  if (1) {
    int temp == _id0_;
    _id1_ = _id2_;
    _id3_ = temp;
  }
   else ;
}
```

Query



Offloading analysis

☒ '_id4_' was not declared

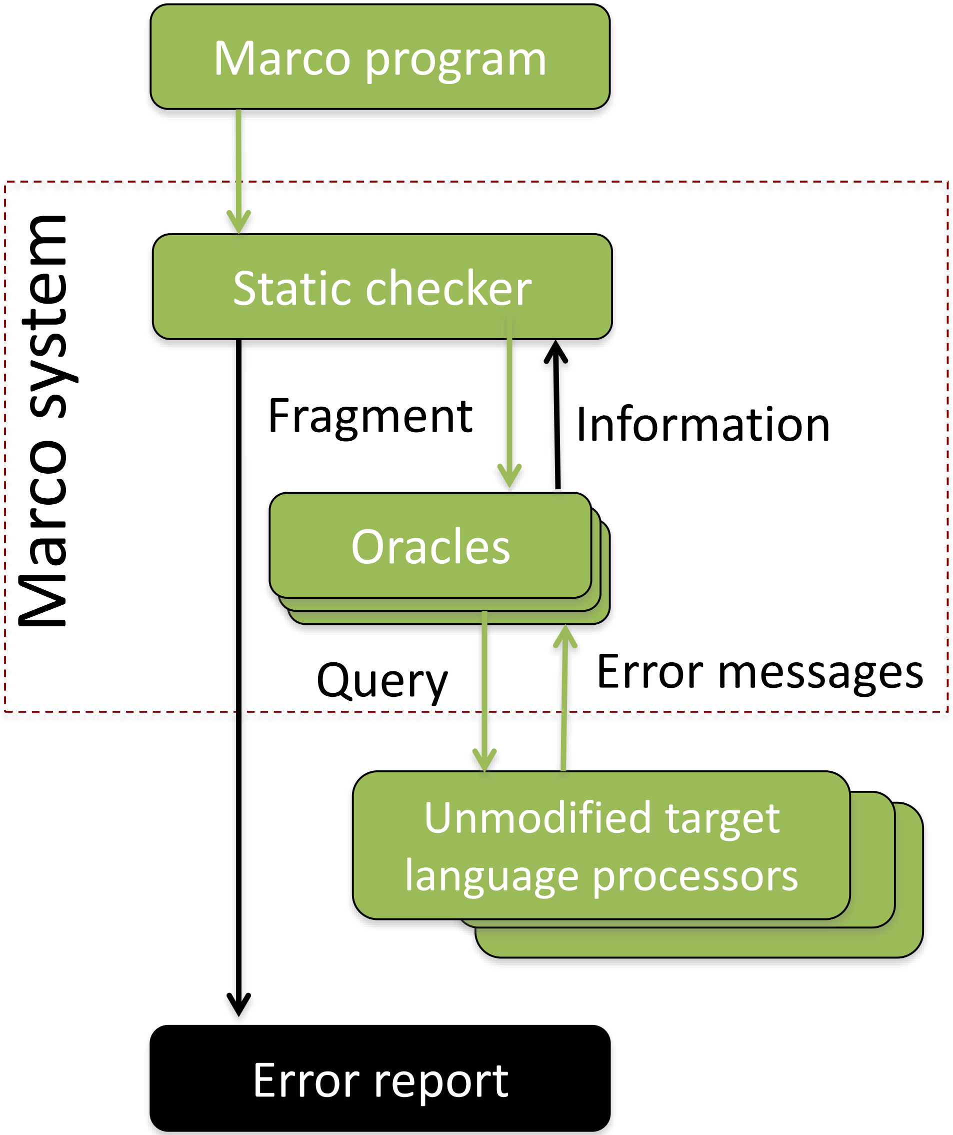
☒ '_id0_' was not declared

☒ '_id1_' was not declared

☒ '_id2_' was not declared

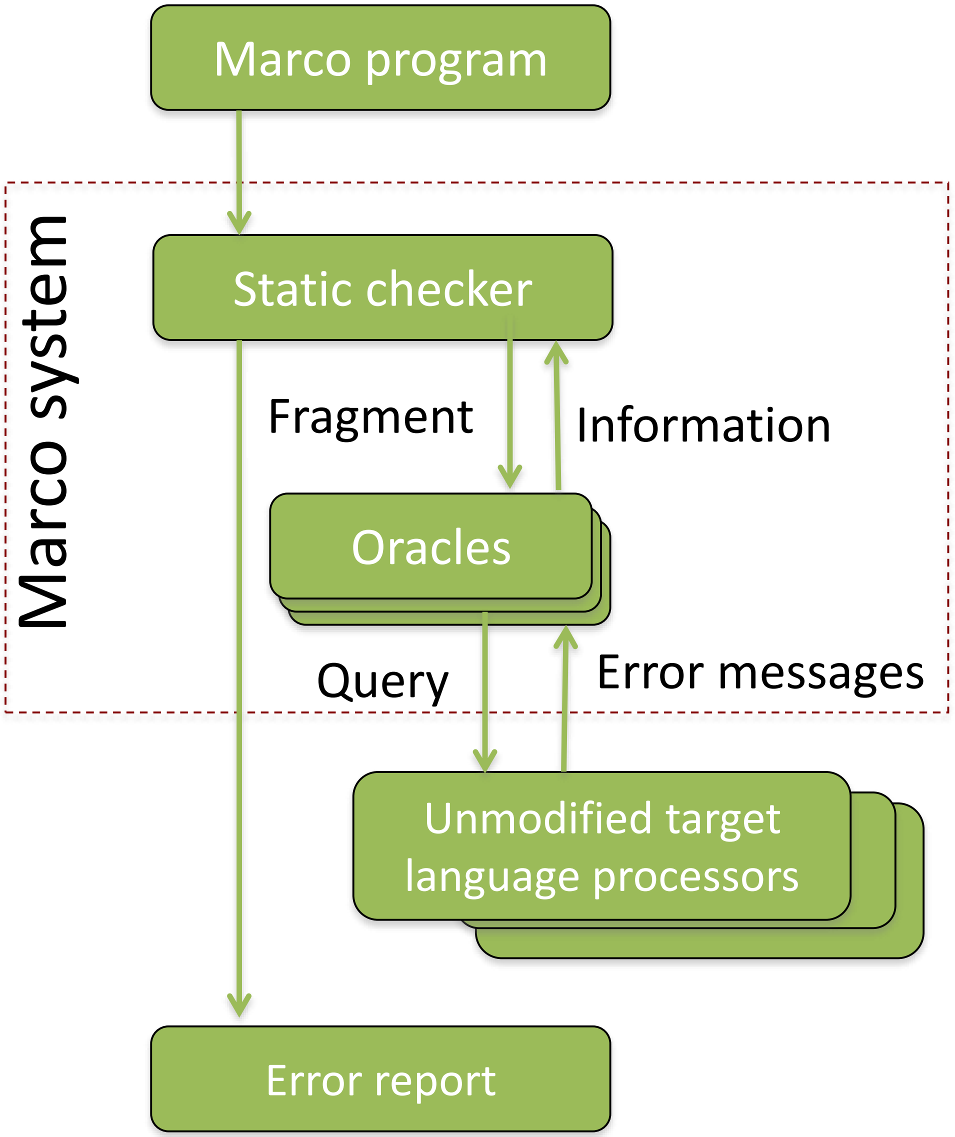
☒ '_id3_' was not declared

GCC Error messages



Offloading analysis

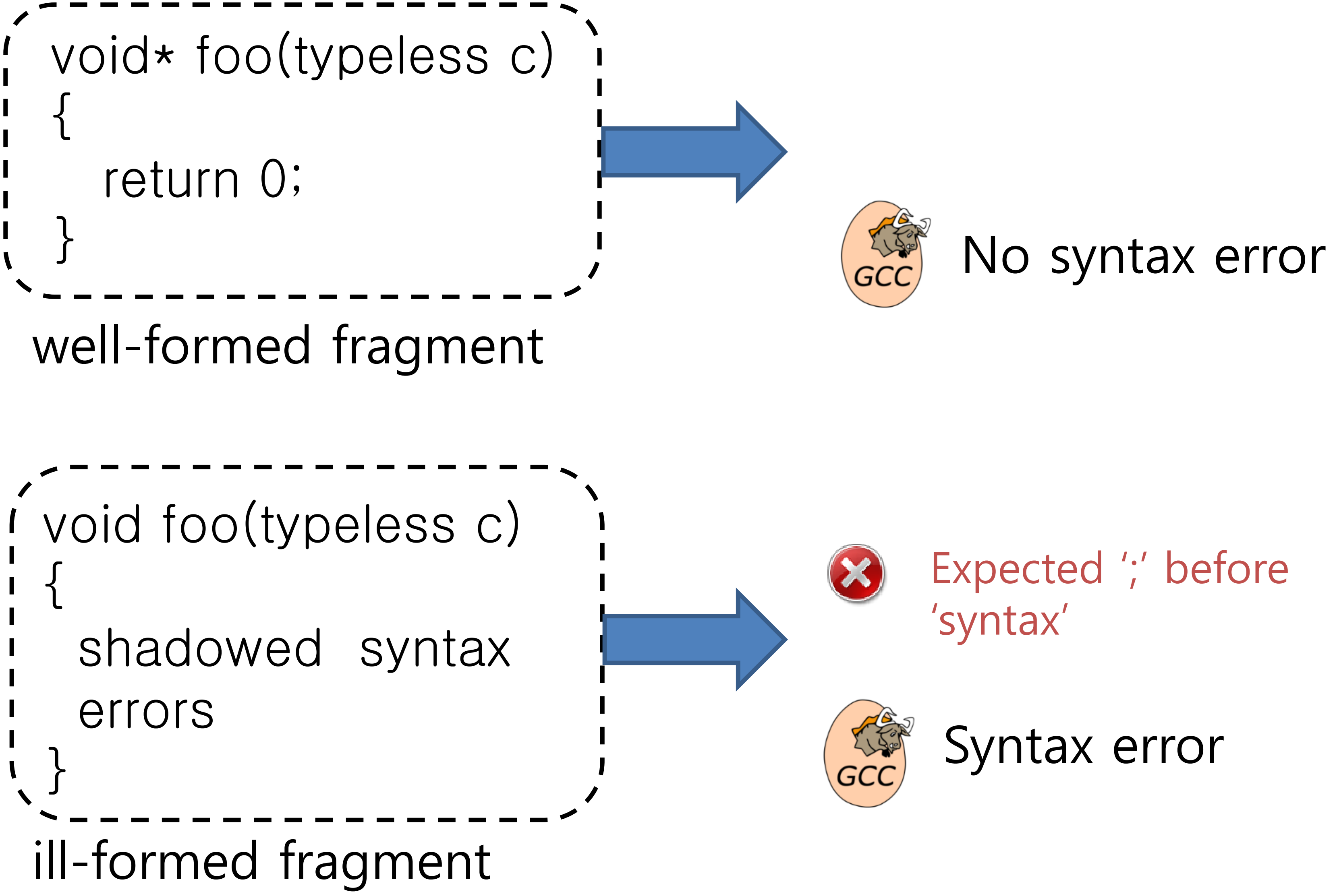
No syntax error



Outline

- Introduction
- Marco language and architecture
  - Expressing macros as Tokens
  - Offloading analysis using oracle queries
- Oracle analysis in practice
- Summary

Naïve oracle analysis in theory



```
void* foo(typeless c)
{
   return 0;
}
```

well-formed fragment

GCC No syntax error

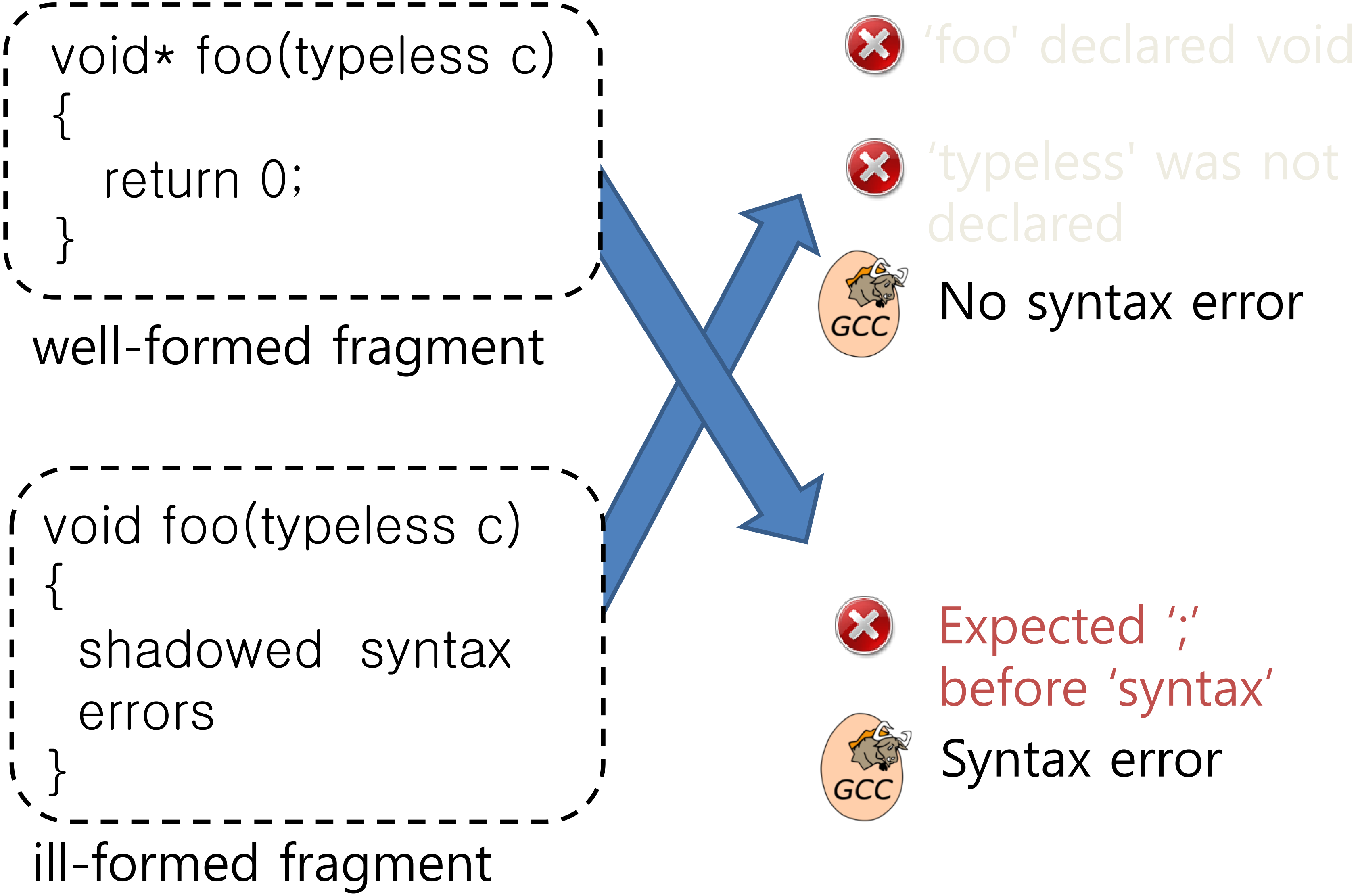
```
void foo(typeless c)
{
  shadowed  syntax
  errors
}
```

ill-formed fragment

✖ Expected ‘;’ before ‘syntax’

GCC Syntax error

Naïve oracle analysis in practice



```
void* foo(typeless c)
{
   return 0;
}
```

well-formed fragment

'foo' declared void

'typeless' was not declared

No syntax error

```
void foo(typeless c)
{
  shadowed  syntax
  errors
}
```

ill-formed fragment

Expected ';' before 'syntax'

Syntax error

Syntax errors for well-formed fragments

```
void* foo(typeless c)
{
   return 0;
}
```

well-formed fragment

☒ ‘typeless' was not declared

☒ expected ‘,’ or ‘;’ before ‘{’

Syntax errors

Syntax errors for correct fragments

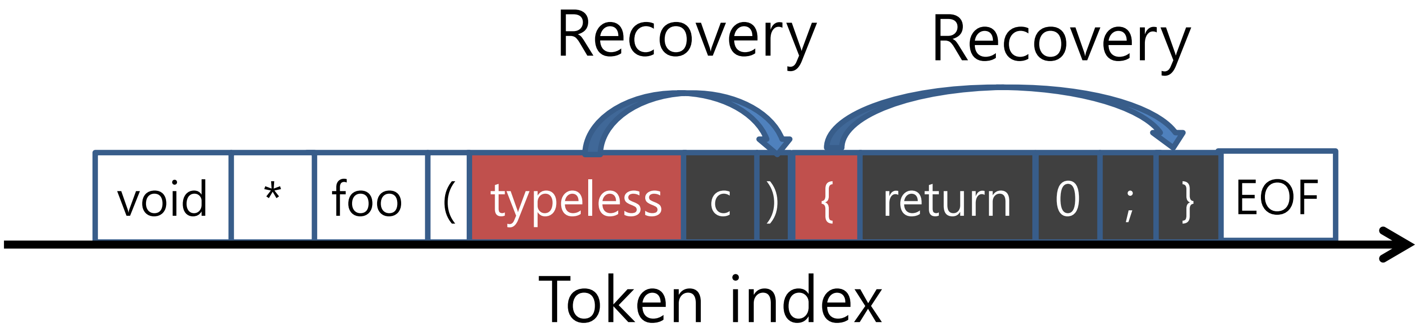
```
void* foo(typeless c)
{
   return 0;
}
```

well-formed fragment

✗ 'typeless' was not declared

✗ expected ',' or ';' before '{'

GCC Syntax errors



Recovery

Recovery

void | * | foo | (| typeless | c |) | { | return | 0 | ; | } | EOF

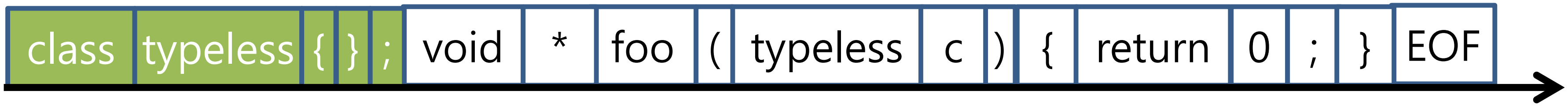
Token index

Our solution of speculating a context

```
class typeless {};
void* foo(typeless c)
{
    return 0;
}
```

well-formed fragment

No syntax errors



class	typeless	{	}	;	void	*	foo	(	typeless	c	)	{	return	0	;	}	EOF

Token index

No syntax errors for wrong fragments

```
void foo(typeless c)
{
  shadowed  syntax
  errors
}
```

ill-formed fragment

❌ ‘foo' declared void

❌ ‘typeless' was not declared

No syntax error

No syntax errors for wrong fragments

```
void foo(typeless c)
{
  shadowed  syntax
  errors
}
```

ill-formed fragment

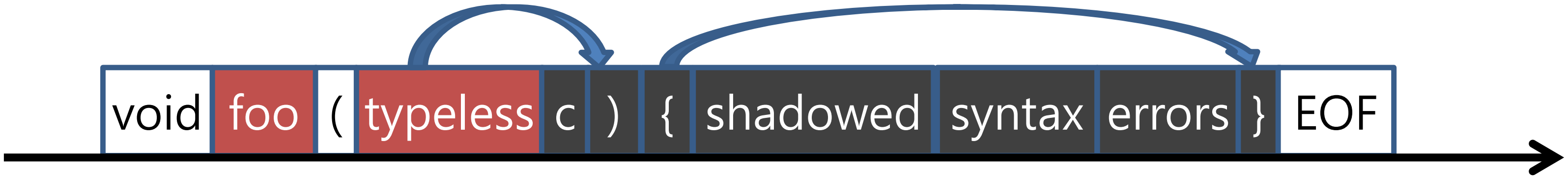
✖ 'foo' declared void

✖ 'typeless' was not declared

No syntax error

Recovery

Recovery



void | foo | (| typeless | c |) | { | shadowed | syntax | errors | } | EOF

Token index

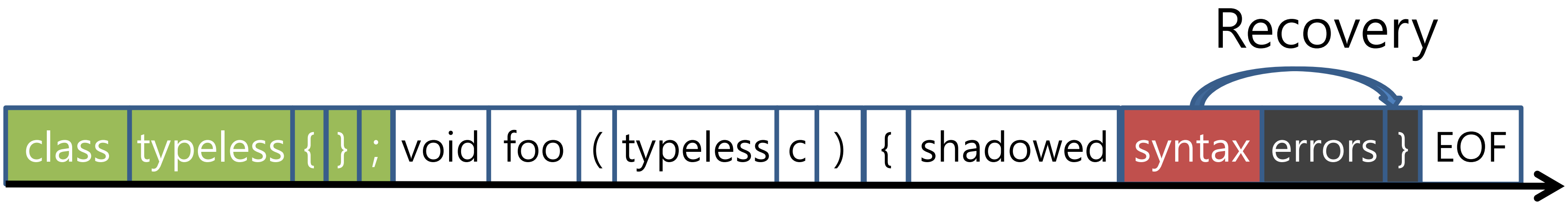
Our solution of speculating a context

```
class typeless {};
void foo(typeless c) {
{
  shadowed  syntax
  errors
}
```

Ill-formed fragment

Expected ‘;’ before ‘syntax’

Syntax error



Recovery

class	typeless	{	}	;	void	foo	(	typeless	c	)	{	shadowed	syntax	errors	}	EOF	

Token index

What this talk did not cover

- Issues in the paper
  - Speculation and backtracking
  - Classifying error messages
  - Ensuring hygienic macro expansion
  - Experimental evaluation

Summary

- Macros in programming languages
  - Simple, elegant core language
  - Abstraction and interoperability
  - Tradeoff between safety and encapsulation
- Our approach in **Marco**
  - Representing marcos as tokens
  - Offloading analyses to target-language processors
- Oracle analysis in practice
  - Context-sensitivity in C/C++
  - Speculations and backtracking

Thank you

Questions?

Backup slides

Speculations and backtracking

- Speculation
  - Guess entities for C++ identifiers
  - Type, variable, method, field, namespace
- Backtracking
  - Invalidate some speculations
  - Modest number of backtrackings in practice
- Empirical evaluation
  - 8 microbenchmarks and one realistic one
  - 10-20% backtrackings over 136 fragments

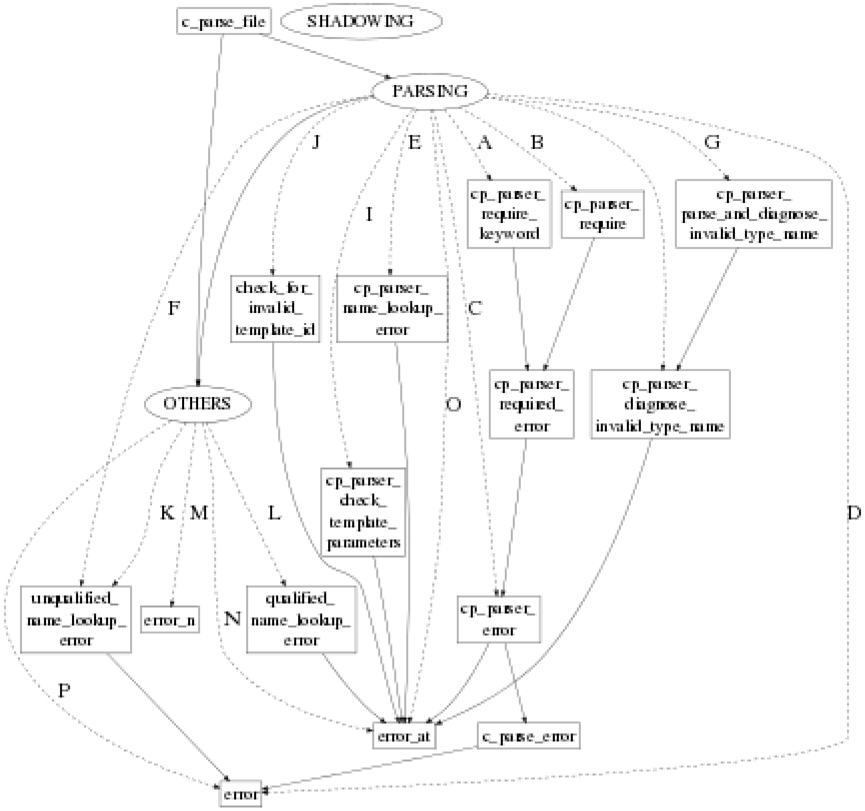
Backtracking in practice

- 8 micro benchmarks
  - 21 fragments
  - 20% queries backtrack
- “Aggregate” operator in IBM InfoSphere Streams
  - 115 fragments
  - 13% backtracking rate
- Modest rate of backtrackings

Classifying and handling error messages

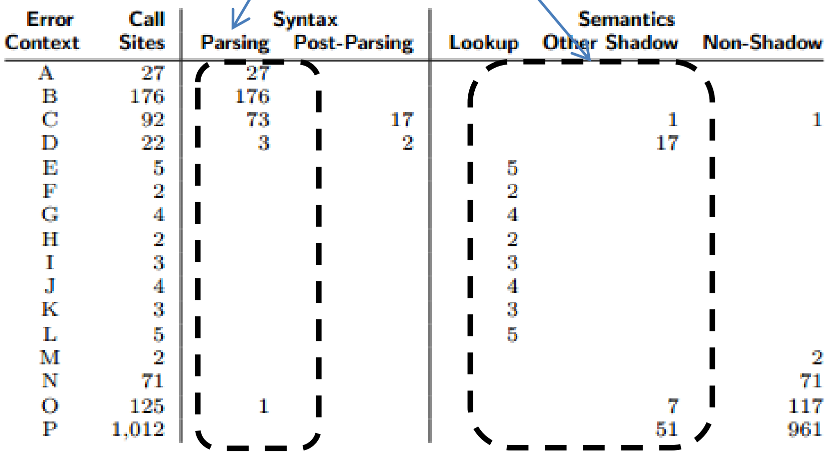
Classes	Example	Handling
Syntax	expected ‘;’ before ‘{‘	Forward to programmers
Lookup	‘typeless' was not declared	**Eliminate them by speculating a proper context.**
Shadowing	function ‘typeless‘ was duplicated	
Non-shadowing	‘foo' declared void	Ignore

Feasibility of classifying error messages


c_parse_file
SHADOWING
PARSING
J
E
A
B
G
cp_parser_require_keyword
cp_parser_require
cp_parser_parse_and_diagnose_invalid_type_name
I
F
check_for_invalid_template_id
cp_parser_name_lookup_error
C
OTHERS
O
cp_parser_required_error
cp_parser_diagnose_invalid_type_name
cp_parser_check_template_parameters
K
M
L
D
unqualified_name_lookup_error
error_n
N
qualified_name_lookup_error
cp_parser_error
P
error_at
c_parse_error
error

Abstracted call graph for printing error message in g++

Dozens of regular expressions cover 384 critical error messages



Error Context	Call Sites	Syntax		Semantics		
		Parsing	Post-Parsing	Lookup	Other Shadow	Non-Shadow
A	27	27				
B	176	176				
C	92	73	17		1	1
D	22	3	2		17	
E	5			5		
F	2			2		
G	4			4		
H	2			2		
I	3			3		
J	4			4		
K	3			3		
L	5			5		
M	2					2
N	71					71
O	125	1			7	117
P	1,012				51	961

Mapping from call sites to error classes

Cost of adding new target languages

	C++	SQL
Lexical analysis	a few lines in **rats!**	a few lines in **rats!**
SLOC for Oracle Plug-ins	1K SLOC in Java	392 SLOC in Java
SLOC in the target-language parser	110K+ SLOC in C	1K SLOC in Lamon DSL

Extension modules for new languages

Target-language processors reused

10-100 factor of benefit in offloading analysis to target-language processors!

Unhygienic macro expansion

```
code<cpp,stmt> swap(
  code<cpp,id> x,
  code<cpp,id> y) {
  return `cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }]; }
```

A macro function

```
code<cpp,stmt> fail() {
   return swap(
     `cpp[temp],
     `cpp[i]);
}
```

An unhygienic macro expansion

Unhygienic macro expansion

```
code<cpp,stmt> swap(
  code<cpp,id> x,
  code<cpp,id> y) {
  return `cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }]; }
```

A macro declaring
a local variable (temp)

```
{
    int temp = temp;
    temp = i;
    i = temp;
 }
```

Expanded code
containing accidental
name capture

Constraints for unhygienic expansion

```
code<cpp,stmt> swap(
  code<cpp,id> x,
  code<cpp,id> y) {
  return `cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }]; }
```

A macro generating captured name constraints

captured: x ≠ temp

```
code<cpp,stmt> fail() {
   return swap(
     `cpp[temp],
     `cpp[i]);
}
```

A macro generating free name constraints

free: x = temp

A conflict indicates that the macros are not hygienic

Captured name constraints

```
code<cpp,stmt> swap(
  code<cpp,id> x,
  code<cpp,id> y) {
  return `cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
  }]; }
```

captured: $x_1 \neq$ temp

How do we discover that **temp** is captured at the first blank?

A macro declaring a local variable (temp)

Oracle analysis for captured names

```
`cpp(stmt) [ {
    int temp = $x;
    $x = $y;
    $y = temp;
 }]; }
```

Fragment

```
void _id0_() {
  if (1) {
    int temp = temp;
    _id1_ = _id2_;
    _id3_= temp;
  } else ;
}
```

Oracle query

✖ '_id1_' was not declared

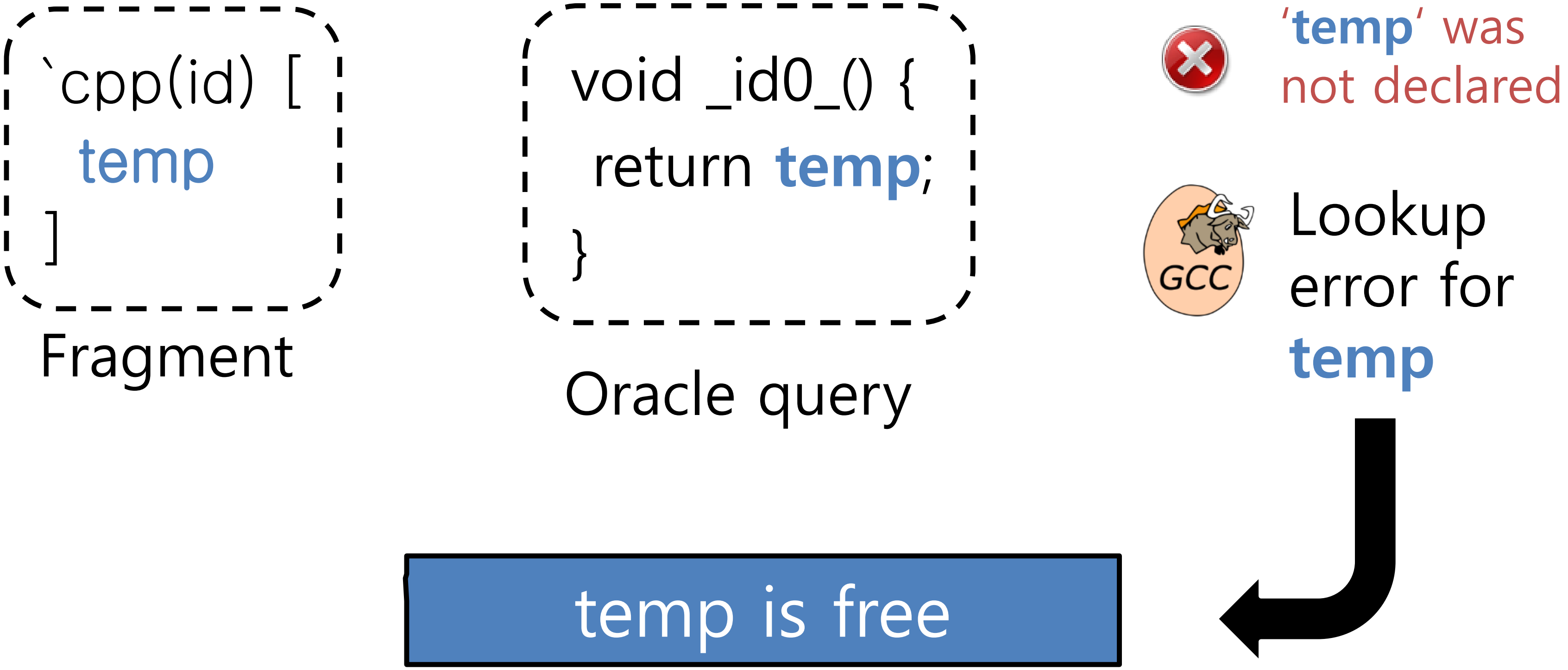
✖ '__d2_' was not declared

✖ '_id3_' was not declared

GCC No lookup error for **temp**

captured: x ≠ temp

Finding out free names



```
`cpp(id) [
  temp
]
```

Fragment

```
void _id0_() {
 return temp;
}
```

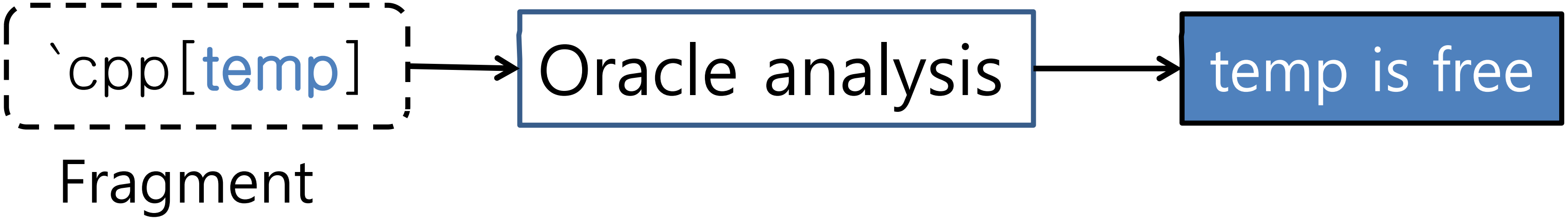
Oracle query

'temp' was not declared

Lookup error for temp

temp is free

Propagating free name constraints



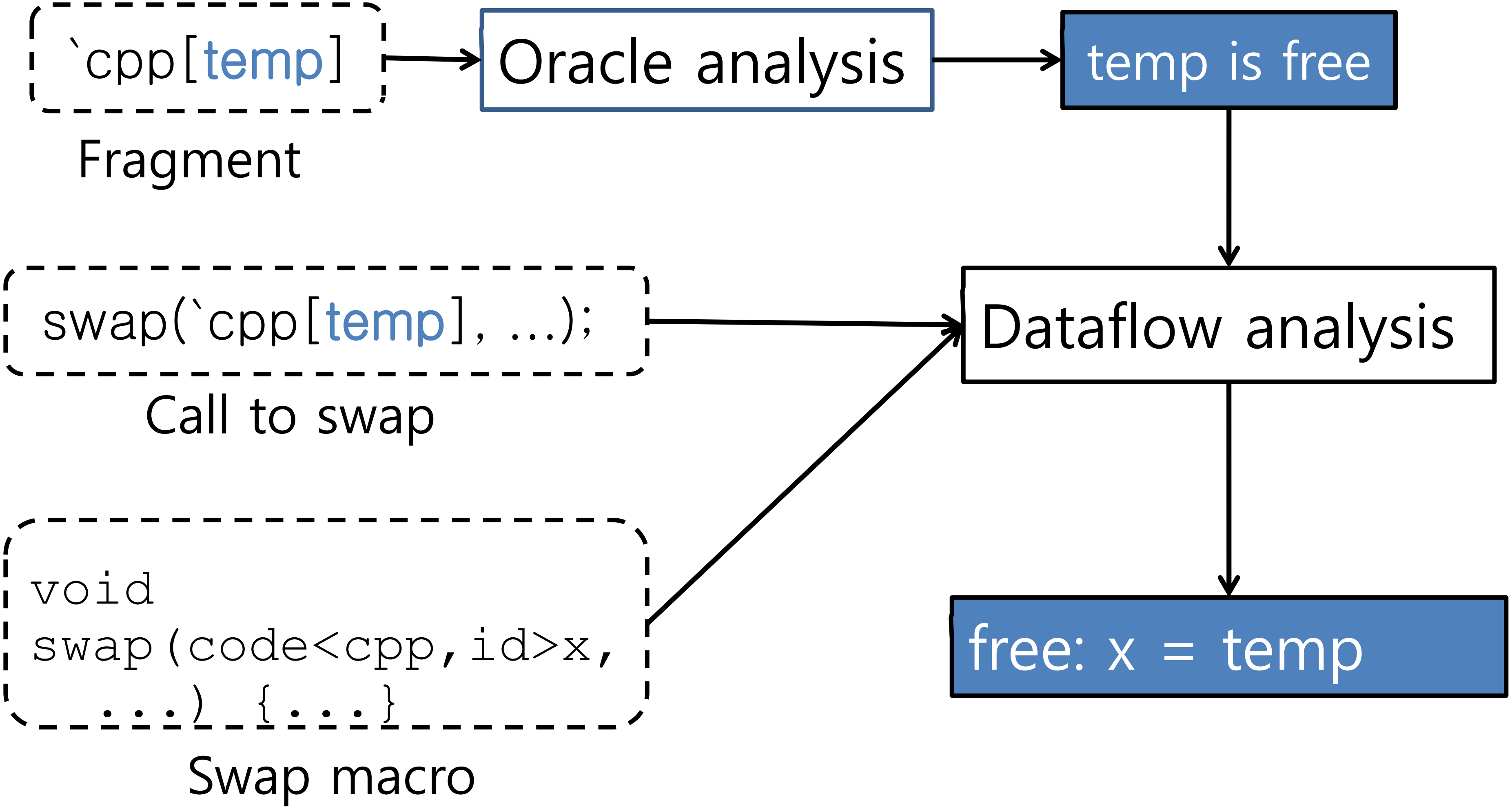
`cpp[temp]

Fragment

Oracle analysis

temp is free

Propagating free name constraints



`cpp[temp]

Fragment

Oracle analysis

temp is free

swap(`cpp[temp], ...);

Call to swap

Dataflow analysis

```
void
swap(code<cpp,id>x,
  ...) {...}
```

Swap macro

free: x = temp