Marco: Safe, Expressive Macros for any Language

Byeongcheol Lee[1], Robert Grimm[2], Martin Hirzel[3], Kathryn S. McKinley[4,1]

[1] University of Texas Austin
[2] New York University
[3] IBM Watson Research Center
[4] Microsoft Research

Abstract. Macros improve expressiveness, concision, abstraction, and language interoperability without changing the programming language itself. Using macros, developers generate code with concrete syntax instead of futzing with abstract syntax trees or, worse, strings. Macro systems include the C/C++ preprocessor, Scheme macros, and multi-stage programming. On the one hand, the C/C++ preprocessor is unsafe. It operates on individual tokens and is a well-known source of errors. On the other hand, Scheme macros and multi-stage programming ensure syntactic and semantic correctness, but they are deeply coupled to their target languages, compilers, and interpreters. This coupling limits extensibility, i.e., adding new syntax or target languages.

This paper presents the Marco macro system, which achieves expressiveness, safety, and language scalability by composing target-language compilers and interpreters through a common translation engine. (1) Marco is expressive. Its domain-specific macro language is strongly typed and includes loops, conditionals, functions, and *target-language fragments*, which may contain blanks that other fragments fill. (2) Marco is safe. A static checker verifies fragments with blanks and a dynamic interpreter verifies fragments after blanks are filled. (3) Marco scales across languages. A plug-in facility reuses off-the-shelf target-language compilers and interpreters. They resolve fragment-specific safety queries in the form of specially crafted programs. We demonstrate Marco with two target languages, C++ and SQL. We chose C++ for depth, due to its complexity, and SQL for breadth, due to its peculiar syntax and scoping rules. Our approach opens the door to safe and expressive macros for any language.

1 Introduction

Macros enhance programming languages without changing them. Programmers use macros to add missing language features, to improve concision and abstraction, and to interoperate between distinct languages and systems. Programmers use concrete syntax in macros instead of tediously futzing with abstract syntax trees or, worse, untyped and unchecked strings. For instance, Scheme relies heavily on macros to provide a fully featured language while at the same time keeping the core simple and elegant. The C preprocessor lets programmers derive variations from the same code base, e.g., through conditional compilation, and abstract over the local execution environment, e.g., through types and variables conditionally defined in system-wide header files. Web programmers are accustomed to writing meta-programs in PHP and similar languages to generate target code in HTML. And middle-tier web applications interface with back-end databases through generated SQL queries. As demonstrated by the last two examples, macros not only improve individual languages, but are also indispensable for building increasingly prevalent multilingual applications.

Macros are written in a *macro language*, generate code in a *target language*, and are often embedded in a *host language*, which may be the same as the target language. The macro translator always resolves macro invocations before target code is compiled or interpreted and, ideally, should ensure that target-language fragments are syntactically and semantically correct. Not all macro systems meet this ideal. Notably, C preprocessor macros operate on individual tokens and are not guaranteed to produce correct code. Consequently, open-source C programs contain numerous errors due to macros [7]. Researchers have proposed *safe* macro systems that generate syntactically and semantically correct code in their target languages. For example, MS^2 [27] and JSE [2] are preprocessing systems that raise the level of abstraction from token transformation to syntax transformation for C and Java, respectively. In multi-stage programming, a formal type system guarantees syntactic safety in dynamically generated code [15, 16, 18, 28].

Unfortunately, these safe macro systems are deeply tied to their target languages, which seriously limits their *scalability* across host and target languages. For instance, MS^2, JSE, JavaMint, MetaML, and Scheme macros all are tightly coupled to their respective target languages (C, Java, ML, and Scheme) and are deeply embedded into their compilers [2, 16, 18, 27, 28]. This tight coupling of host and target languages has two primary disadvantages, which negatively reinforce each other. First, a particular macro system is limited to a prescribed combination of host and target languages and cannot be shared across languages. Language designers thus need to reinvent the wheel and developers need to relearn macro programming for each host and target-language combination. Second, the macro translator is deeply embedded in the host compiler, seriously complicating its implementation. This coupling increases the temptation to either omit macros or fall back on a better encapsulated but unsafe macro processor, such as the C preprocessor. Given the general utility of macros and the diversity of current and emerging programming languages, expressive and safe macros that scale across host and target languages are clearly desirable.

This paper presents the Marco macro system, an expressive, safe, extensible, and language-scalable system that composes target-language compilers and interpreters with a language-independent macro translator. Its only requirement is that each target-language compiler or interpreter produce descriptive error messages that identify the cause and source location of errors. Programmers write macros in the domain-specific Marco language. Marco is largely a straight-forward procedural language. It is strongly typed and supports loops, conditionals, functions, and so on. Marco also contains target-language *fragments* as first-class values. As indicated by the name, fragments need not be complete target-language programs. Rather, they may contain individual expressions, statements, and declarations, as well as *blanks* that are filled in by other fragments. Since the Marco system is a first stab at language scalability for macros, Marco programs are currently stand-alone and not embedded in a host language. We leave host language integration as future work, e.g., one potential approach is to use the composition techniques from Jeannie for C and Java [12].

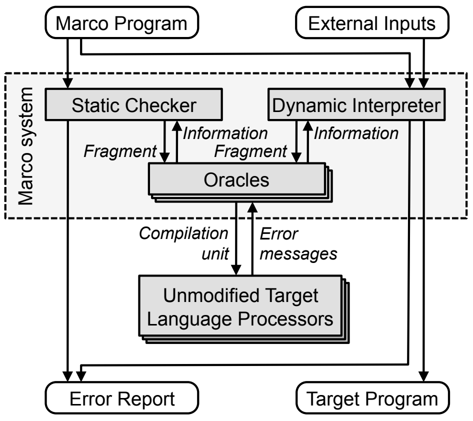

Fig. 1. The Marco architecture.

Figure 1 illustrates the Marco *static checker* taking a Marco program as input and verifying that target-language fragments are correct at macro definition time. Because Marco supports code generation based on external inputs, including additional target-language fragments, some unresolved syntax and/or name errors may survive static checking. Consequently, the Marco *dynamic interpreter* verifies fragments again at macro instantiation time, when Marco has filled in all blanks. This double checking is critical for the development of macro libraries, where one set of programmers writes the macros and another set of programmers instantiates them with application-specific inputs. Both the static checker and dynamic interpreter rely on common *oracles* to verify target-language code and report any errors. The oracles abstracts over the different target languages. They query their target-language processors, currently, gcc for C++ macros and SQLite for SQL macros, by submitting specially crafted small compilation units.

Marco achieves language scalability by composing off-the-shelf target-language compilers and translators with a common translation engine. To add a new target language, developers implement a simple lexical analysis module that recognizes target-language identifiers and the end of target-language fragments. They also need to implement a module with three oracle functions that (1) check for syntactic well-formedness, (2) determine a fragment's free names, and (3) test whether a fragment captures a name. Everything else is target-language independent. In particular, Marco includes a reusable dataflow analysis, which propagates free and captured identifiers and then flags accidental name capture [16].

We evaluate Marco based on three criteria: expressiveness, safety, and scalability. For expressiveness, Marco has loops, conditionals, and functions, making it Turing-complete and more expressive than C/C++ macros. For safety, Marco is strongly-typed and uses its dataflow analysis for fragments to ensure the naming discipline of target-language code, making it safer than Scheme and C/C++ macros. For language scalability, Marco relies on error messages from target-language implementations, making it the first safe macro system that is independent of the target language. We demonstrate Marco for two target languages. For depth, we chose C++, which is relatively complex due to its rich syntax, types, and many other features. For breadth, we chose SQL, which differs substantially from C++ and is of critical importance to many web applications.

In summary, this paper's contributions are: (1) The design of a safe and expressive macro language that is scalable over target languages. (2) A macro safety verification mechanism that uses unmodified target-language compilers and interpreters. (3) An implementation of a static checker, dynamic interpreter, and fragment analyzer for C++ and SQL.

2 The Marco Language

This section describes the Marco language, using examples, grammar rules, and type rules. The Marco language is a statically typed, procedural language. It supports macros using three constructs: code types, fragments, and blanks. We define and illustrate these constructs using the motivating example in Figure 2.

The *synch* macro in Figure 2 ensures that lock acquire and release operations are properly paired (modulo exceptions), which C++ does not ensure. Lines 1-2 contain the signature of the Marco function *synch*, which takes two parameters, a C++ identifier and a C++ statement, and returns a C++ statement. The **code** type is parameterized by the target language and a nonterminal. Line 4 uses the back-tick operator (`) to begin a *fragment*, which is a quoted piece of target-language code. Line 6 uses the dollar operator (**$**) to begin a *blank*, which is an escaped piece of Marco code embedded in a fragment. The evaluation rule for a fragment first evaluates embedded blanks, then splices their results into the fragment's target-language code:

```
1  code<cpp, stmt>                # code type
2  synch(code<cpp, id> mux,
3        code<cpp, stmt> body) {
4    return
5      `cpp(stmt)[{               # C++ fragment
6        acquireLock($mux);
7        $body                    # blank
8        releaseLock($mux);
9      }];
10 }
```

Fig. 2. Marco code to generate C++.

$$\frac{\forall i \in 1 \ldots n : \mathsf{Env} \vdash e_i \longrightarrow \beta_i \qquad \gamma = \alpha_0 \beta_1 \alpha_1 \ldots \beta_n \alpha_n}{\mathsf{Env} \vdash \;\grave{}\,\mathit{lang}(\mathit{nonT})\,[\alpha_0 \$ e_1 \alpha_1 \ldots \$ e_n \alpha_n] \longrightarrow \;\grave{}\,\mathit{lang}(\mathit{nonT})\,[\gamma]} \quad \text{(E-Fragment)}$$

Each α_i is a sequence of target-language tokens, each $\$e_i$ is a blank, and each β_i is the result of evaluating a blank to a sequence of target-language tokens. The result γ is the concatenation of all α_i and β_i.

Figure 3 presents the Marco grammar. The unique grammar rules are *fragment* and its helpers. A fragment, such as `cpp(stmt)[...$x...$y...], consists of a head and a sequence of fragment elements. The head specifies the target language, a nonterminal, and optionally a list of captured identifiers. We use the optional list of captured identifiers to check the naming discipline in the target

```
program      ::= functionDef+
functionDef  ::= type ID ‘(’ (formal (, formal)*)? ‘)’ ‘{’ stmt* ‘}’
formal       ::= type ID
stmt         ::= ‘{’ stmt* ‘}’                           # block
               | type ID ‘=’ expr ‘;’                    # variable declaration
               | ‘if’ ‘(’ expr ‘)’ stmt (‘else’ stmt)?   # conditional
               | ‘for’ ‘(’ ID ‘in’ expr ‘)’ stmt         # loop
               | ‘return’ expr ‘;’                       # function return
               | expr ‘;’                                # expression statement
expr         ::= fragment                                # fragment
               | ‘(’ expr ‘)’                            # parentheses
               | ID                                      # variable use
               | expr INFIX_OP expr                      # infix operation
               | ID ‘(’ (expr (, expr)*)? ‘)’            # function call
               | expr ‘.’ ID                             # record attribute
               | expr ‘[’ expr ‘]’                       # list subscript
               | ‘[’ (expr (, expr)*)? ‘]’               # list literal
               | ‘{’ ID ‘=’ expr (, ID ‘=’ expr)* ‘}’    # record literal
               | ‘true’ | ‘false’ | INT | STRING         # primitive literal
type         ::= ‘code’ ‘<’ language ‘,’ nonTerm ‘>’     # fragment type
               | ‘list’ ‘<’ type ‘>’                     # list type
               | ‘record’ ‘<’ formal (, formal)* ‘>’     # record type
               | ‘boolean’ | ‘int’ | ‘string’            # primitive type
fragment     ::= fragmentHead ‘[’ fragmentElem* ‘]’
fragmentHead ::= ‘`’ language ‘(’ nonTerm (‘,’ capture)? ‘)’
language     ::= ID
nonTerm      ::= ID
capture      ::= ‘capture’ ‘=’ ‘[’ ID (, ID)* ‘]’
fragmentElem ::= TARGET_TOKEN | blank
blank        ::= ‘$’ baseExpr
```

Fig. 3. Marco grammar.

code (see Section 6). There are two kinds of fragment elements: target-language tokens to emit, and blanks to fill in during evaluation.

Since the fragment elements are enclosed in square brackets, the Marco parser must count matching square brackets in the fragment itself to find the end, e.g., in `` `cpp(expr)[arr[idx]] ``. However, the Marco parser should ignore square brackets in target-language strings or comments, e.g., in `` `cpp(expr)[printf("[")] ``. Since different languages have different tokens for strings and comments, Marco must be configured with target-language specific lexers. To select lexers based on the syntactic context, we use a scannerless parser generator, *Rats!* [10]. Marco's lexers are very simple, since they only need to recognize a few key target-language tokens. The target-language specific lexers must also find identifiers. The lexer must recognize strings as well as certain numbers (such as "3.1e4" or "1llu"), and it needs a table of keywords, to avoid reporting spurious identifiers.

The static type system includes the primitive types **`boolean`**, **`int`**, and **`string`**; the type **`list`** parameterized by element type **`record`** parameterized by attribute names and types; and the type **`code`** parameterized by target language and non-terminals. The latter is key to *target language scalability*. For example, Figure 4 shows a Marco macro generating SQL. Compared to Figure 2, the Marco syntax and semantics remain the same, but the target language and therefore the code type parameters differ. The Marco engine code may therefore perform the same dataflow analysis on both to ensure that identifiers from multiple macros generate consistent bindings by selecting and invoking the appropriate target-language oracle that determine free and captured identifiers. In other words, the strongly typed quote and unquote mechanism lets Marco maximize the target-language independent engine while identifying which oracle to utilize for which target language, both of which are novel to Marco.

The type rule for a Marco fragment first checks the types for each of the embedded blanks, which must result in code belonging to the same target language ($lang$). It then uses the language $lang$, the nonterminal $nonT$ of the fragment, the nonterminals $nonT_i$ of each of the blanks, and the contents of the fragment as inputs to a syntax oracle. As far as the Marco type system is concerned, the syntax oracle is a black-box that can either succeed or fail. If the oracle succeeds, the type of the fragment is **code**<$lang$, $nonT$>.

```
code<sql,query>
genTitleQueryInSQL(code<sql,expr> pred){
  return `sql(query)[select title
      from moz_bookmarks where $pred
    ];
}
```

Fig. 4. Marco code to generate SQL.

$$\frac{\begin{array}{c}\forall i \in 1 \ldots n : \Gamma \vdash e_i : \texttt{code<}lang\texttt{,}\ nonT_i\texttt{>} \\ syntaxOracle(lang)(nonT, [nonT_1, \ldots, nonT_n], \alpha_0 \texttt{\$} 1 \alpha_1 \ldots \texttt{\$} n \alpha_n)\end{array}}{\Gamma \vdash \texttt{`}lang\texttt{(}nonT\texttt{)}\ \texttt{[}\alpha_0 \texttt{\$} e_1 \alpha_1 \ldots \texttt{\$} e_n \alpha_n \texttt{]}\ :\ \texttt{code<}lang\texttt{,}\ nonT\texttt{>}} \quad \text{(T-FRAGMENT)}$$

3 The Marco Analysis Framework

The Marco system provides a static checker and a dynamic interpreter. The static checker verifies correctness at macro development time. The dynamic interpreter generates target-language code and verifies correctness at runtime. The two components share target-specific oracles, which check for syntactic well-formedness and naming discipline in target-language fragments.

Our central design goal is target-language independence. Marco is extensible, since adding target-language specific *oracles* as plug-ins requires no changes to the core engine. Each oracle communicates with a black-box target-language *processor* (compiler or interpreter) to analyze fragments. The system generates compilation units as inputs to the oracle target-language processors and parses error messages from their outputs. The only target-language specific parts are the target-language specific lexers (see Section 2) and the oracles. A key advantage of Marco over other safe macro systems is that it does not require new or even modified target-language processors.

To check target-language syntax, the system first parses the Marco program and tokenizes target-language fragments. It then ensures that the target-language fragments are consistent with their declared **`code`** type parameters. For example, consider the C++ fragment `` `cpp(expr)[x = 1;] ``. The Marco type checker applies rule T-Fragment from Section 2, which triggers a call to *syntaxOracle*(`cpp`)(`expr`, [], '`x = 1;`'). The C++ syntax oracle then generates the following compilation unit for the unmodified C++ compiler (i.e., the GNU C++ compiler, gcc): `int query_expr(){return (x = 1;);}` For this input, gcc reports an error complaining about the spurious semicolon after `x = 1`. Based on this error message, the oracle deduces that the fragment was syntactically ill-formed for non-terminal `expr`. Since the oracle failed, Marco type-checking fails as well, and Marco reports an error. This example ignores idiosyncrasies of blanks and C++, which Sections 4 and 5 explain in detail.

Each plug-in provides three oracles: *syntax*, *free-names*, and *captured-name*. While all plug-ins implement the same Java interface, they interact with the target-language processor in different ways. Notably, since Java already integrates database access through JDBC, the SQL plug-in uses this API. In contrast, the C++ plug-in interacts with gcc through the file system. Either way, all target-language interactions share two characteristics. First, target-language processors receive programs as sequences of characters: strings for JDBC and files for gcc. In other words, we lower the tokenized fragments to produce character strings. Second, the processor outputs are strings that indicate syntactic or semantic errors. The concrete error reporting mechanism depends on the target language, e.g., Java exceptions for JDBC and standard error output for gcc.

For checking the naming discipline, consider a fragment f_1 that fills a blank in fragment f_2. Fragment f_1 is `` `sql(expr)[birthYear >= 1991] ``, and fragment f_2 is `` `sql(query)[select name from Patrons where $pred] ``. The free-names oracle discovers that f_1 contains the free identifier `birthYear`. Marco then uses its dataflow analysis to discover that f_1 flows into the blank of f_2. Finally, Marco uses its captured-name oracle to check whether identifier `birthYear` is captured at blank `$pred`. Programmers use annotations to tell Marco when a capture is intentional; otherwise, Marco reports an accidental-capture error. The next sections describe how the oracles abstract errors from target-language processors into information for Marco's static checker and dynamic interpreter.

4 Checking Syntactic Well-Formedness

This section describes how Marco checks target-language syntax. The *syntax oracle* is the interface between the target-language agnostic Marco system and the black-box target-language processors. The signature of the syntax oracle, as embodied in type rule T-Fragment from Section 2, is:

$$\mathit{syntaxOracle} : \mathit{lang} \to (\mathit{nonT}, \mathbf{list}{<}\mathit{nonT}{>}, \alpha_0\mathbf{\$}1\alpha_1 \ldots \mathbf{\$}n\alpha_n) \to \mathbf{list}{<}\mathit{error}{>}$$

Marco type	Place-holder fragment	Completion fragment
code<*sql*, *expr*>	0	**select** $*fresh1* **from** $*fresh2* **where** ($*orig*)
code<*sql*, *query*>	**select** *	$*orig*
code<*sql*, *qlist*>	/**empty**/	$*orig*
code<*cpp*, *expr*>	0	**int** $*fresh*() { **return** ($*orig*); }
code<*cpp*, *stmt*>	;	**void** $*fresh*() { **if**(1) $*orig* **else**; }
code<*cpp*, *id*>	$*fresh*	**int** $*fresh*() { **return** ($*orig*); }
code<*cpp*, *type_sp*>	**int**	$*orig* $*fresh*;
code<*cpp*, *type_id*>	**int**	**int** $*fresh*() { **return** *sizeof*($*orig*); }
code<*cpp*, *fdef*>	**void** $*fresh*(){}	$*orig*
code<*cpp*, *mdecl*>	**int** $*fresh*;	**class** $*fresh* { $*orig* };
code<*cpp*, *decl*>	**int** $*fresh*;	$*orig*
code<*cpp*, *cunit*>	/**empty**/	$*orig*

Table 1. Helper fragments for syntax oracles. Place-holder fragments fill in blanks. Completion fragments turn a fragment into a self-contained compilation unit. Marco fills in **$***fresh* blanks with fresh identifiers, and **$***orig* blanks with the original fragment.

For example, consider the following invocation of the syntax oracle:

syntaxOracle (*sql*) (*query*, [*expr*], '**select** *a* **from** *B* **where** **$**1')

In this example, the target language is SQL, the nonterminal of the fragment is *query*, and there is one blank, whose nonterminal is *expr*. The fragment contents have the form α_0**$**1$\alpha_1$, where α_0 is the token sequence before the blank, **$**1 marks the location of the blank, and α_1 is the token sequence after the blank. In the example, α_0 is '**select** *a* **from** *B* **where**' and α_1 is empty. The remainder of this section describes the syntax oracle algorithm for producing compilation units, interpreting the results, and iterating when necessary.

4.1 Syntax Oracle Algorithm

The syntax oracle algorithm has four steps. The key problem is to fill in each *blank* in the target-language fragment.

Step 1: Fill in blanks. The syntax oracle starts by filling in each blank with a place-holder fragment. The middle column of Table 1 shows the place-holder fragments for each of the code types in Marco's SQL and C++ plug-ins. Each such place-holder fragment is syntactically valid for a given nonterminal. In the example above, the nonterminal for blank 1 is *expr*, so the syntax oracle fills in blank 1 with the place-holder fragment for SQL expressions, which is 0. The result is the fragment **select** *a* **from** *B* **where** 0. The intuition why filling in blanks works is that target languages have (more or less) context-free grammars, and the syntax oracle can check syntactic validity even when there are semantic errors. For instance, in the example, the place-holder fragment is of type integer and the blank expects type boolean, but this semantic mismatch is irrelevant to syntactic well-formedness.

Step 2: Complete the fragment. Next, the syntax oracle completes the fragment to obtain a self-contained compilation unit. In this example, the fragment is already a full query, and needs no additional code. The right column of Table 1 shows the completion fragments for each of the code types in Marco's SQL and C++ plug-ins. Besides turning a fragment into a compilation unit, Step 2 may also generate boiler-plate syntax. For SQL, this step adds code to begin and then abort a transaction, which prevents side-effects when sending the SQL query to a live database during analysis.

Step 3: Run the target-language processor. At this point, the syntax oracle sends the completed fragment to the target-language processor, and collects error messages, if any. For SQL, Marco makes a JDBC call and catches any exceptions. For C++, Marco generates a file with the fragment, compiles it with gcc, and reads any error messages from `stderr`.

Step 4: Determine oracle results. Finally, the syntax oracle translates errors from the target-language processor into oracle results. It must distinguish syntax errors from other errors. It only fails the syntactic well-formedness test if there are syntax errors. In C++, other errors may mask syntax errors, so the oracle may iterate to determine if the fragment also has a syntax error, as Section 5 explains. If the syntax oracle fails, the oracle maps error message line-numbers back to the original Marco code, and reports the error to the Marco user.

4.2 Syntax Oracle Example

Consider the example fragment `` `sql(expr)[type = ] ``, which is missing its right operand. Type rule T-Fragment invokes the syntax oracle as follows: *syntaxOracle*(`sql`)(`expr`, [], '`type =`'). The oracle goes through its four steps:

1. Fill in blanks. This step is a no-op, since there are no blanks.
2. Complete the fragment. The oracle consults Table 1 to find the completion fragment for **`code`**`<sql, expr>`, yielding **`select`** `x` **`from`** `T` **`where`** `(type =)`.
3. Run the target-language processor. The oracle uses JDBC to send the completed fragment to SQLite, and then catches the resulting SQLException, which contains the error message "Syntax error near '='.".
4. Determine result. Since the error from the target-language processor was a syntax error, the oracle reports this error back to the user.

Now, assume the user fixes the fragment, writing `` `sql(expr)[type = 1] ``, and then runs Marco again.

1. Fill in blanks. This step is still a no-op, since there are no blanks.
2. Complete the fragment yields **`select`** `x` **`from`** `T` **`where`** `(type = 1)`.
3. Run the target-language processor. If there is no table `T` with an attribute `type` in the database, the error message is "No attribute '`type`' in table '`T`'.".
4. Determine result. Since the error is not a syntax error, the oracle succeeds and indicates that the fragment is syntactically well-formed.

5 Context-Sensitive Syntax

Most programming languages, including SQL, Java, ML, and Scheme, have context-free syntax. In this case, the syntax oracle from Section 4 works directly as described: it checks the syntactic well-formedness of a fragment in isolation based on code type, which specifies the language and nonterminal. However, our goal was to handle any language, including languages with context-sensitive syntax such as C++. Prior work on safe macro systems ignores this issue. This section extends our syntax oracle to deal with context sensitivity correctly.

5.1 Context-Sensitive Syntax Examples

As a first example, consider the following C++ fragment:

```
`cpp(mdecl)[void* method(typeless o) { return 0; }]
```

The nonterminal `mdecl` stands for a member declaration. The fragment is syntactically well-formed for this nonterminal, since a method is a special case of a member. However, after using the completion fragment for `mdecl` from Table 1, gcc reports the following errors:

```
error: expected ';' at end of member declaration
error: expected ')' before 'o'
```

These are syntax errors, even though the root cause is a semantic problem: identifier `typeless` has not been declared as a type. When gcc cannot parse `typeless` as a declaration specifier, it speculates that `method` is a variable name. But the downstream tokens make no sense for a variable declaration. This case shows how a semantic root problem, the missing declaration context for `typeless`, *induces* a syntax error. To resolve cases like this, our C++ syntax oracle enumerates all identifiers in the input fragment, and speculates one by one that they are a type name. Speculation results in additional gcc queries with context declarations. The next query for this example is:

```
class typeless { };                          // speculative context
class id1 {                                  // completion fragment
  void* method(typeless o) { return 0; } // input fragment
};
```

Since gcc reports no syntax errors for this query, the syntax oracle succeeds, concluding correctly that the input fragment was syntactically well-formed. In general, our syntax oracle chooses an identifier that allows gcc to parse furthest past the original reported error. That way, it avoids having to try an exponential number of speculation steps.

As a second example, consider another C++ fragment (due to [21]):

```
`cpp(stmt)[ A(*x)[4] = y; ]
```

The C++ compiler can parse this in two ways: either as a variable declaration or as an expression statement. If `A` is a type, then the code declares variable `x` as a pointer to an array of 4 elements of type `A`, and initializes `x` to `y`. On the other hand, if `A` is an identifier for a function, then the code calls the function with parameter `*x`, accesses element `[4]` of the result, and assigns `y` to it. The ambiguity between declarations and statements is so prevalent in C++ that the

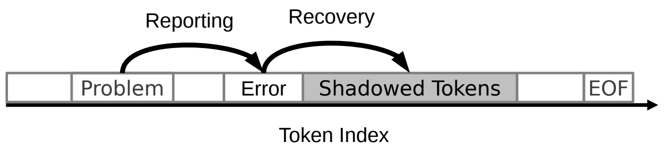

Fig. 5. Error reporting and recovery

language specification has a disambiguation rule of preferring declarations over expression statements [24, p. 802].

If part of a program is not well-formed, language processors report the error and try to recover, so that they can report more than one error per invocation. Figure 5 depicts how language processors scan through lexical tokens, detect a syntactic or semantic problem, generate an error message, skip several tokens, and continue analysis. For instance, parsers generated by ANTLR report syntax errors and then recover by either inserting one token or skipping downstream tokens. The hand-written parsers in gcc report both syntax and semantic errors, and may skip all or some downstream tokens.

If the skipped tokens contain a syntax error, then the error recovery for the first error *shadows* the syntax error. Therefore, the absence of syntax errors does not imply syntactic well-formedness. The example fragment `A(*x)[4] = y;` triggers the following error messages:

```
error: 'x' was not declared in this scope
error: 'A' was not declared in this scope
error: 'y' was not declared in this scope
```

These semantic errors may shadow downstream syntax errors, so our oracle speculates that an identifier in a semantic error may be either a type or variable name. In the example, making `A` a type name and `x` and `y` variable names eliminates the missing declaration errors, as shown in the following oracle query:

```
class A {}; class {} x; class {} y;  // speculative context
void query() {                       // completion fragment
  A(*x)[4] = y;                      // input fragment
}k
```

This fragment still yields a semantic error ("cannot convert '`<anonymous class>`' to '`A (*)[4]`'"), but the error cannot shadow syntax errors. Hence, the oracle concludes that the fragment is syntactically well-formed.

5.2 Error Classification

The syntax oracle is concerned with syntax errors, and in an ideal world, it would not have to deal with semantic errors. However, as the examples above demonstrate, semantic root problems affect syntax errors in two cases: a semantic problem can *induce* a syntax error, or a semantic problem can *shadow* a syntax error. In the induced-error case, it appears as if the fragment is syntactically ill-formed, but it is actually well-formed. In the shadowed-error case, it appears as if the fragment is syntactically well-formed, but it is actually ill-formed.

We need not handle the case of a syntax error inducing or shadowing another error (the first error suffices to conclude that the fragment is syntactically ill-formed). Neither do we need to handle the case of a semantic problem inducing or shadowing another semantic error (as long as it does not induce or shadow a syntax error, it does not affect the syntax oracle). Consequently, the syntax oracle must recognize only the following errors:

- Syntax errors
- Semantic errors that may shadow other errors

We systematically investigated all C++ errors potentially generated by gcc to validate that our syntax oracle handles these cases correctly. Our investigation found that there are two kinds of syntax errors, which we call parsing errors and post-parsing errors. Parsing errors are easy to recognize (they all begin with the word "expected" and include a token or nonterminal symbol), and there are only a few post-parsing errors.

To collect all shadowing error messages, we identified the seven error recovery routines in gcc that update the parser state to skip tokens until a synchronization token. For example, one such routine is `skip_until_sync_token()`. Next, we enumerated all call-sites to the recovery routines. Finally, we looked at the code leading up to the call-site. That code commonly looked similar to the following:

```
bool ok = perform_semantic_check();
if (ok)
  error("A");
else
  error("B");
if (!ok) {
  error("C");
  skip_until_sync_token();
}
```

Error `C` may shadow syntax errors, because the compiler reports `C` before it skips tokens. Similarly, error `B` may shadow syntax errors, because the compiler reports `B` if *ok* is false, in which case it then skips tokens. Error `A`, on the other hand, does not shadow tokens. In most cases, we only had to look at a single routine to understand error shadowing, though in a few cases, multiple routines were involved. We found that shadowing errors are most commonly lookup errors, though there were also a few non-lookup errors that shadow other errors.

5.3 Iterative Syntax Oracles in Marco

In summary, context sensitivity prevents a Marco oracle from concluding syntactic well-formedness based solely on the absence of syntax errors. In particlur for C++, a semantic problem can either induce or shadow a syntax error. Therefore, Marco's syntax oracle for C++ speculatively resolves syntax errors and shadowing semantic errors by issuing repeated queries to gcc, each with a different speculative context. In other words, the oracle speculates declarations for identifiers as variables or types. If Step 4 from Section 4.1 detects lookup errors, the algorithm iterates back to Step 2 and resolves them by generating different declarations for identifiers.

6 Checking Naming Discipline

This section describes how Marco uses dataflow analysis to ensure that code generation does not cause accidental name capture in the target language. Some macro systems, notably Scheme, prevent capture by automatically renaming variables, but require deep target-language specific knowledge and is therefore not an option for Marco, which is target-language agnostic. Accidental name capture is a typical bug when using the C preprocessor, as illustrated in Fig. 6.

```
1 #define swap(v,w) {int temp=v; v=w; w=temp;}
2 int temp = thermometer();
3 if (temp<lo_temp) swap(temp, lo_temp)
```

Fig. 6. Example for accidental name capture bug with C preprocessor [6, 7].

Line 1 in Fig. 6 declares a macro `swap` with a local variable declaration `temp` (short for "temporary"). Line 2 declares a different variable `temp` (short for "temperature") that is not nested in the macro. Line 3 passes name `temp` as an actual parameter to the formal `v` of `swap`. This use of `v` captures the name `temp`. Since the code intends to use `temp` to refer to "temperature", this problem is called an accidental name capture.

More generally, accidental name capture happens when a first fragment f_1 contains a free name x; a second fragment f_2 unintentionally captures name x at blank b; and f_1 flows into b. Marco detects capture as follows. The *freeNamesOracle* discovers all free names in fragment f_1. Marco's forward dataflow analysis propagates free names to blanks. The *capturedNameOracle* discovers whether f_2 captures name x at blank b. In dataflow terminology, the analysis state is a map from meta-language variables to free target-language names. The Gen-set of a fragment is the set of free names in the fragment. The Kill-set of a fragment at a blank is the set of names that it captures at that blank.

Marco uses static dataflow analysis at compilation time and dynamic dataflow analysis in the interpreter at code generation time. The oracles are target-language specific and use the off-the-shelf target-language processor as a black-box to generate error messages that reveal information about free and captured names. The dataflow analysis itself is target-language independent.

6.1 Free-Names Oracle

The signature of the free-names oracle is:

$$\mathit{freeNamesOracle} : \mathit{lang} \rightarrow (\mathit{nonT}, \texttt{list}{<}\mathit{nonT}{>}, \alpha_0 \texttt{\$}1\alpha_1 \ldots \texttt{\$}n\alpha_n) \rightarrow \texttt{list}{<}\mathit{ID}{>}$$

For example, given fragment `` `cpp(expr)[100 * (1.0 / (foo))] ``, Marco invokes the free-names oracle as follows:

$$\mathit{freeNamesOracle}(\mathtt{cpp})(\mathtt{expr}, [], \text{‘}\mathtt{100 * (1.0 / (foo))}\text{’})$$

A target-language name is free if it is not bound inside the fragment. In the example, *foo* is free, and thus, the oracle call should return the list [*foo*]. To obtain this result, the free-names oracle first consults Table 1 to generate a completion fragment to send to gcc:

```
int query_expr() { return (100 * (1.0 / (foo))); }
```

For this query, gcc returns the error message "name '*foo*' was not declared in this scope". The free-names oracle looks for error messages that indicate a name is undefined. In the example, the message specifies the name *foo*. The oracle speculates that *foo* is free. To validate this hypothesis, it runs one more experiment. It prepends a declaration of the name *foo* to the translation unit, and sends it to gcc again. In the example, the test is:

```
int foo;
int query_expr() { return (100 * (1.0 / (foo))); }
```

This modification resolves the declaration error and confirms the hypothesis is correct. Hence, the oracle adds the name *foo* to the list of free names. It repeats this process until it does not observe any more declaration errors. The insight Marco exploits is that a name in a fragment is free, as long as it could be bound by a declaration from an enclosing scope.

6.2 Captured-Name Oracle

The captured-name oracle checks whether a target-language name is captured by a blank in a fragment and thus if it is safe to fill in that blank with a fragment in which the name is free. The signature of the captured-name oracle is:

$$\begin{aligned}\mathit{capturedNameOracle} \;:\;& \mathit{lang}\\ & \to (\,\mathit{nonT}, \mathbf{list}{<}\mathit{nonT}{>},\\ & \quad\; \alpha_0\mathbf{\$}1\alpha_1 \ldots \mathbf{\$}n\alpha_n, \mathbf{int}, \mathit{ID})\\ & \to \mathbf{boolean}\end{aligned}$$

We check the blank number **int** and the free name *ID* for capture. Consider swapping two integers: `` `cpp(stmt)[{int temp=$v; $v=$w; $w=temp;}]``. The following oracle call checks whether blank 1 captures name *temp*:

$$\begin{aligned}\mathit{capturedNameOracle}\;(\,&\mathtt{cpp}\,)\\ (\,&\mathtt{stmt}, [\mathtt{expr}, \mathtt{expr}, \mathtt{expr}, \mathtt{expr}],\\ &\text{‘}\mathtt{\{int\ temp=\$1;\ \$2=\$3;\ \$4=temp;\}}\text{’}, 1, \mathtt{temp}\,)\end{aligned}$$

Since blank 1 in the fragment does in fact capture the target-language name *temp*, the oracle returns **true**. Consider the fragment `` `sql(query)[select name from Patrons where $pred]``. The blank in this fragment captures any names referring to column names in the *Patrons* table in the database. Note that SQL's scoping rules implement semantics similar to a **`with`**-statement, presenting a different naming discipline challenge than C++ scoping rules. Our *capturedNameOracle* algorithm handles both target languages with the same approach.

Assume that *capturedNameOracle* was invoked to check whether free name x is captured at blank number i. Like the other oracles, the captured-name oracle fills in all blanks j with $i \neq j$ using the place-holders corresponding to their non-terminals from Table 1. However, for blank i, our analysis hypothesizes that x is captured at the blank. To find counter-evidence, it places x in the blank, wrapping it as necessary in some boiler-plate code for syntactic well-formedness. If the target-language processor reports an "x is unknown" error message, then the oracle concludes that x is not captured at blank i, and returns **`false`**. Otherwise, it returns **`true`**.

6.3 Intentional Capture

The dataflow analysis propagates free target-language names through variables and blanks. It reports an error if a blank accidentally captures a free name. To determine whether a capture is accidental or intentional, the analysis uses the optional **`capture`** annotation in the *fragmentHead* clause of the Marco grammar (see Fig. 3). If a name is listed in the **`capture`** annotation, the capture is intentional, otherwise the capture is accidental and Marco reports an error.

```
1 code<cpp,stmt> boundIf(code<cpp,expr> cond, code<cpp,stmt> body) {
2   return `cpp(stmt, capture=[it])[{
3       int it = $cond;          #blank 1
4       if (it) { $body }        #blank 2
5     }];
6 }
```

Fig. 7. Example for intentional name capture when using Marco to generate C++ code.

For an example of intentional capture, assume that the author of a macro wants to bind name `it` to a particular value at a blank. In other words, the intention is that if the blank gets filled in with a fragment containing name `it`, then `it` gets captured. Fig. 7 shows a Marco function `boundIf` that illustrates this case. The function implements an if-statement that binds the value of the condition to `it`, in order to use it in the body. The annotation **`capture`**`=[it]` in Line 2 indicates that the capture is intentional. Now, assume that blank 2 gets filled with fragment `printf("%d", it);`. Since the analysis should only report accidental capture and not intentional capture, it will not report an error for `it` even though it appears free in the fragment and gets captured in blank 2.

6.4 Dataflow Analysis

The interesting statements for the dataflow analysis are statements with fragments and blanks. Fig. 8 shows the transfer-function for such statements. Given a Marco statement and an input analysis state *inState*, the transfer function computes an output analysis state *outState*. The analysis state only changes for

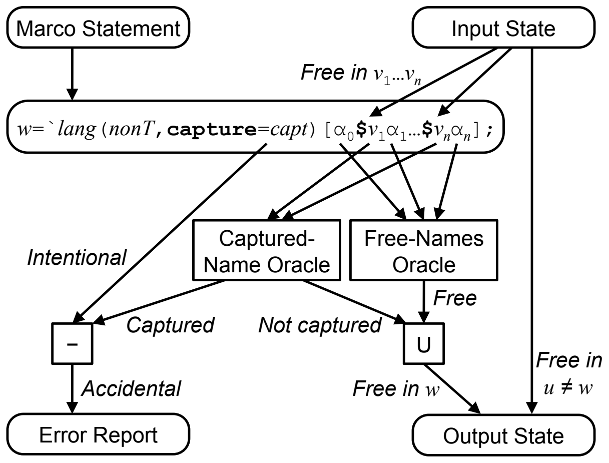

Fig. 8. Transfer functions for the naming-discipline analysis.

the Marco location w assigned by the statement. The captured-name oracle from Section 6.2 checks whether free names from $\mathit{inState}(v_1)$ thru $\mathit{inState}(v_n)$ are captured by the fragment. If they are captured, and the capture is not intentional, the analysis reports an error. If they are not captured, then they are still free in w. In addition, the free-names oracle checks for free names in the constant portions α_0 thru α_n of the fragment. Those free names are also free in w. The resulting output state $\mathit{outState}(w)$ uses the free names for w as discovered by the oracles. For all other locations $u \neq w$, the transfer function forwards the free names from the input state $\mathit{outState}(u) = \mathit{inState}(u)$.

One pragmatic issue is how to report high-quality error messages in the case of accidental name captures. The analysis remembers which errors it has reported so far and avoids duplicates. Furthermore, the analysis tracks the originating fragment for each free name to more accurately report the source of accidental name captures. When the analysis detects an accidental capture, it reports both the line number of the origin and the line number of the capture.

The static dataflow analysis checks naming discipline in the static checker, and can hence find many bugs early. It reports accidental name capture errors to macro authors at compile time. However, a Marco program may also receive fragments from external input parameters at runtime. These fragments may contain free names, and thus, the Marco dynamic interpreter also checks for accidental captures at code-generation time. The dynamic analysis uses the same transfer function as the static analysis, invoking the same oracle queries.

7 Experimental Evaluation

This section experimentally validates the key characteristics of Marco: expressiveness, safety, and scalability. To evaluate expressiveness, we implemented microbenchmarks from prior work and a code-generation template for a high-performance stream processing operator. To evaluate safety, we execute Marco on each microbenchmark and on the streaming operator. To evaluate scalability,

Marco Program	Fragment	Code Type	Size	Backtrack	Queries	Decls
paint	*Painting1*	**code**<*cpp*, *stmt*>	17	5	17	7
dynamic_bind	*dynamic_bind1*	**code**<*cpp*, *stmt*>	13	3	14	8
exceptions	*throw1*	**code**<*cpp*, *stmt*>	23	2	12	7
	throw2	**code**<*cpp*, *stmt*>	28	2	16	7
	catch1	**code**<*cpp*, *expr*>	1	1	3	3
	catch2	**code**<*cpp*, *stmt*>	51	1	8	4
	unwind_protect1	**code**<*cpp*, *expr*>	1	1	3	3
	unwind_protect2	**code**<*cpp*, *stmt*>	44	2	12	6
myenum	*myenum1*	**code**<*cpp*, *decl*>	5	0	1	1
	myenum2	**code**<*cpp*, *stmt*>	9	1	5	4
	myenum3	**code**<*cpp*, *decl*>	15	0	1	1
	myenum4	**code**<*cpp*, *stmt*>	14	2	8	6
	myenum5	**code**<*cpp*, *decl*>	18	0	2	2
discriminant	*discriminant1*	**code**<*cpp*, *expr*>	9	0	1	1
complain	*complain1*	**code**<*cpp*, *stmt*>	4	0	2	2
	main1	**code**<*cpp*, *expr*>	1	1	3	3
	main2	**code**<*cpp*, *stmt*>	13	0	2	2
swap	*swap1*	**code**<*cpp*, *id*>	1	1	3	3
	swap2	**code**<*cpp*, *id*>	1	1	3	3
	swap3	**code**<*cpp*, *stmt*>	28	5	31	12
SQLSyntax	*good1*	**code**<*sql*, *expr*>	3	0	1	0
	good2	**code**<*sql*, *stmt*>	6	0	1	0

Table 2. Oracle analysis results for the fragments in the micro-benchmarks.

we report statistics on the implementation effort for supporting different target languages.

7.1 Methodology

Tools and environments. We use Marco r278 running on the Sun HotSpot Client JVM 1.6.0_21-ea. For the unmodified target-language processors, we downloaded and built gcc 4.6.1 as well as SQLiteJDBC v056 based on SQLite 3.4.14.2. We conducted all experiments on a Core 2 Duo 1.40 GHz with 4 GB main memory. The machine runs Ubuntu 11.10 on the Linux 3.0.0-12 kernel.

Marco programs. We wrote 8 Marco microbenchmark programs with 22 macro functions derived from related work [6, 27] and the *Aggregate* operator derived from IBM InfoSphere Streams [14]. The *Aggregate* operator generates C++ declarations, statements, and expressions that exercise classes, namespaces, and templates. Table 2 presents the microbenchmarks. The first four programs implement C++ macros from the MS^2 paper by Weise and Crew [27]. These macros add new abstractions such as resource management (*paint*), dynamic binding (*dynamic_bind*), rich exception handling (*exception_handling*), and multiple declarations (*myenum*). The next three programs implement C++ versions of the examples from "Macros That Work" by Clinger and Rees [6]. These macros illustrate naming issues in macro expansions. The final program generates SQL queries of extracting the titles of the bookmarks in a web browser's database.

Data collection methodology. To collect statistical results from analyzing fragments, we turned on Marco's -pstat command-line option. To count source lines

Code type	Count	Size	Backtrack	Queries	Decls
code<*cpp*, *id*>	5	1.00	0.80	3.00	3.00
code<*cpp*, *type_spec*>	8	6.88	0.00	3.75	2.75
code<*cpp*, *type_id*>	1	1.00	0.00	2.00	2.00
code<*cpp*, *expr*>	12	4.50	0.08	2.67	2.58
code<*cpp*, *stmt*>	40	13.20	1.58	10.13	6.63
code<*cpp*, *fdef*>	11	31.00	4.09	21.73	9.36
code<*cpp*, *mdecl*>	22	12.36	0.05	3.23	3.00
code<*cpp*, *decl*>	13	11.38	0.00	4.08	3.00
code<*cpp*, *cunit*>	3	7.00	0.00	2.00	2.00

Table 3. Oracle analysis results for the fragments in the *Aggregate* operator.

of code, we ran the sloccount utility. For the number of error handling rules, we manually examined source files in the Marco system.

7.2 Expressiveness and Safety

In Table 2, Column "Fragment" names the macros using logical function names. Column "Code Type" shows the types of the macros, which indicate the target language and nonterminal. Column "Size" counts the number of target-language tokens and blanks. The remaining columns present the results from running the oracle analysis. The oracle analysis synthesizes query programs before it concludes that the input fragment is syntactically correct. Column "Backtrack" counts how often the syntax oracle needed to backtrack before it finished. Column "Queries" counts the number of compilation units sent to the target-language processor. Column "Decls" shows the number of declarations that the oracles needed to synthesize to provide evidence for syntactic well-formedness.

For fragments containing 1–52 tokens or blanks, our oracle analyzer concludes syntactic well-formedness after evaluating 1–13 query fragments. The number of queries is proportional to the number of synthesized declarations rather than the size of input fragments. This result is not surprising, because the number of C++ parsing errors for syntactically well-formed fragments should be proportional to the number of undefined symbols. About 10-20% of query programs backtrack speculations during the oracle analysis.

This research was originally motivated by language interoperability and concision for IBM's InfoSphere data-stream management system (DSMS) [14]. In an DSMS, an application is represented as a directed graph of data streams and operators. Each operator continuously consumes data from one or more input streams, performs its computation on the data, and then presents the results to one or more output streams. An Aggregate operator uses sum, average, maximum, etc. over a sliding window and must be customized for the particular aggregate and data types. To implement these operator variants, commercial DSMSs use "code generation templates," i.e., macros that generate custom code for a specific operator variant. IBM's InfoSphere Streams DSMS includes a template for the *Aggregate* operator and we re-implement this *Aggregate* operator in Marco.

Table 3 presents statistics from running the oracle analyzer over the 115 fragments in *Aggregate*. The first column classifies fragments by their code type and the second lists the number of fragments for each type. The remaining columns average the number of tokens and blanks ("Size"), the number of backtracks during oracle query analysis ("Backtracks"), the number of generated C++ compilation units for queries ("Queries"), and the number of helper declarations to disambiguate the C++ syntax ("Declarations").

The *Aggregate* operator exercises more C++ specific code types than the microbenchmarks. For instance, the 22 fragments of type **code**<*cpp*, *mdecl*>, where *mdecl* is the member-declarations nonterminal, generate C++ fields, methods, and constructors. No other macro system generates members of a C++ class and checks syntactic correctness of the generated code. Due to ambiguity in the C++ grammar, our oracle analyzer backtracked speculations 72 times over the 114 fragments. Most backtracking arises form C++ fragments that contain unknown variable declarations or expression statements. Even when a variable is bound to a type in C++, the gcc parser uses backtracking. So it comes as no surprise that our oracle also backtracks.

7.3 Scalability

To add target languages in a traditional safe macro system, the developer must modify the target-language processor, which is usually large and complex. To make matters worse, the modified target-language processor is effectively a branch version, and keeping it up-to-date with the main branch requires additional engineering effort. Adding a target language to Marco requires that the developer write a small plug-in consisting of a simple lexer and three oracles. The oracles wrap unmodified target-language processors. We argue that the effort is smaller in the Marco approach.

C++ Plug-in Like all target-language specific Marco plug-ins, our C++ plug-in consists of the lexical analyzer and three oracles. For the lexical analyzer, we define the TARGET_TOKEN terminal in Figure 3 with a few lines of regular expressions for *identifier* (1), *literal* (5), *keyword* (74), and *preprocessing-op-or-punc* (72) [24]. Most regular expressions are trivial and only *identifier* and *literal* (6) require meta-level operators. Our three oracles consist of 1K+ non-blank source lines of Java. About half of the source lines describe declarations in oracle queries, and the other half handle error messages. The error handlers contain 52 regular expressions to classify gcc error messages. In contrast, the gcc source files for the C++ front-end (cc1plus) contain 100K+ non-blank C source lines. The hand-written parser in parser.c has 14K+ non-blank source lines, and it relies on the semantic analysis in 96K+ non-blank source lines to disambiguate parsing decisions. Our C++ plug-in is much smaller and reuses, unmodified, the sophisticated code base that has been maintained for decades.

Figure 9 presents an abstracted, static call graph for gcc's 1,400+ error messages. The cc1plus module consists of 53 C source files, 5,400+ procedures, and 67,000+ call sites. Out of 5,400+ procedures, 2,500+ procedures are reachable from the parser (c_parse_file). We use these 2,500+ procedures to over-approximate the syntactic and semantic error messages. We exclude the preprocessing library (libcpp) and backend library (libbackend) by treating them as

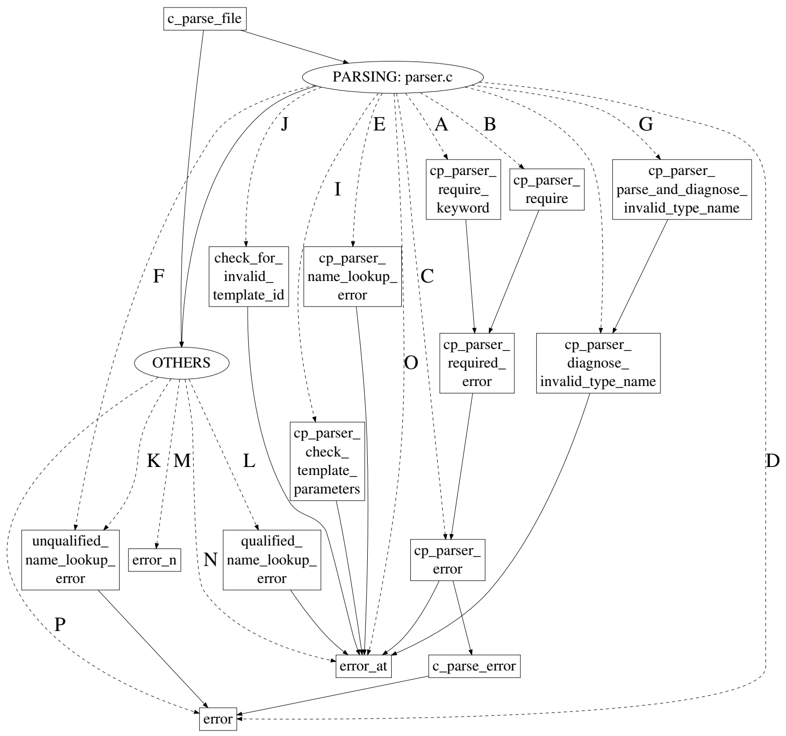
c_parse_file
PARSING: parser.c
J
E
A
B
G
I
cp_parser_ require_ keyword
cp_parser_ require
cp_parser_ parse_and_diagnose_ invalid_type_name
F
check_for_ invalid_ template_id
cp_parser_ name_lookup_ error
C
cp_parser_ required_ error
cp_parser_ diagnose_ invalid_type_name
OTHERS
O
cp_parser_ check_ template_ 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

Fig. 9. Abstract Call Graph for error reporting routines.

terminal functions in the call graph. Our analysis assumes that all error messages must go through the three final error reporting functions: error, error_n, and error_at. We exclude the permerror function because it reports "permissive" errors that never shadow any downstream error messages. We labeled each node with a function name or a group name for functions. The node with the PARSING label represents a set of functions for C++ nonterminals in the top-town parser. The remaining nodes are summarized as the oval node labeled with OTHERS. We label some important edges with capital letters from A to P because their call sites tentatively characterize the kinds of error messages.

Table 7.3 maps the labeled call edges to our classifications of error messages. A and B are parsing syntax errors because they expect specific tokens including *keywords*, *punctuation*, and *operators* in C++. C contains 90 syntax errors and 2 semantic errors. E-L are semantic lookup identifying undeclared or unsatisfiable identifiers. D and M-P are mostly semantic errors. We identified 104 semantic errors that may shadow downstream error messages.

Our oracles recognize 384 critical error messages: 280 parsing error messages, 28 lookup error messages, and 76 other shadowing semantic error messages. A large fraction of these error messages are recognized by a few dozen regular expressions. For instance, all parsing error messages begin with `expected` and end with either a terminal or nonterminal symbol. The Lookup error messages begin with `undeclared`.

Error	Call	Syntax		Semantics		
Context	Sites	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

Table 4. Mapping from calling contexts to error classes.

SQL Oracle Our SQL plug-in consists of the lexical analyzer and the set of three SQL oracle analyzers containing 400 source lines of code. The lexical analyzer for SQL is simpler than the one for C++. SQLite has about 1K source lines in the `parser.y` file written in LALR(1) specification. In this case, they have comparable source lines. SQLite has been maintained, adapted, and tested widely for over a decade. It is better to use a proven parser than to reinvent a new one. Marco uses not just the SQLite parser, but also other components of SQLite for checking naming discipline.

8 Related Work

Unlike previous work, Marco is target-language agnostic and provides safety guarantees by using unmodified existing language processors. In the literature, safe macro systems are deeply coupled with their target language and its implementation (Section 8.1), whereas language-agnostic macro systems fail to provide safety guarantees (Section 8.2). While there is previous work that uses error messages from unmodified language processors, this approach has not been applied to macro systems (Section 8.3). Our oracles take advantage of error reporting, handle error recovery, and extend the-state-of-art in error repairing (Section 8.4).

8.1 Language-Specific Safe Macro Systems

Some macro systems check safety of generated code by deeply coupling meta-languages and their target languages. Like Marco, these systems check safety in macros, but unlike Marco, they are target-language specific.

Syntax. Not all macro systems guarantee syntactic well-formedness of the generated code, and those that do usually do so by implementing a grammar for the target language. For instance, MS^2 implements a C grammar [27], `metafront` implements grammars for Java and HTML [3], and Ur implements grammars for HTML and SQL [5]. These macro systems approach the level of extensible compiler toolkits such as ASF+SDF [26], Polyglot [19], or *Rats!* [10]. While the approach of implementing a grammar works well for language extension and small domain-specific languages, it is problematic for large, existing languages. Besides the sizable development effort, another issue is compatibility. For example, HTML syntax is deceptively simple, but in practice, HTML processors have so many corner-cases that checkers resort to random testing [1]. Marco side-steps this issue by leveraging unmodified target language processors for any language.

Scope. In the functional-language community, a wide-spread technique for ensuring that macros respect scope rules is *hygiene.* Kohlbecker et al. introduced hygienic expansion [16]. Clinger and Rees presented an improved algorithm for renaming identifiers to guarantee hygiene [6]. Kim et al. formally characterize accidental and intentional capture, but do not implement intentional capture [15]. All these systems depend on the syntax and scope rules in their specific target language. The Marco system recognizes scopes with black-box target-language compilers. Marco does not automatically rename identifiers, but uses data-flow analysis to find and report errors when identifiers are accidentally captured.

Semantics. Some macro systems check whether or not expanded fragments will pass type checking in their target languages. Multi-stage extensions generate safe code in, for instance, ML [18] and Java [28]. C++ concepts add contracts to templates [9]. MorphJ verifies some contracts statically so that expanded code will not have name-resolution conflicts [13]. Quail checks types between SQL queries and the database system [25]. Target-language agnostic type checking is an open problem that has not been addressed by any of these systems, and that we do not yet address in Marco either.

8.2 Language-Agnostic Unsafe Macro Systems

Language-agnostic macros are quite common because most programming languages have string data types that represent both well-formed and ill-formed programs in any target language. A JSP web program uses strings to generate SQL queries and HTML/JavaScript pages to talk to back-end database systems and front-end web browsers, respectively. The C preprocessor does not incorporate much information about its target language, because it takes tokenized streams as input and output. Unfortunately, these language-agnostic macro systems provide no safety checks. Ernst et al. present an empirical study that finds that programmers often break safety rules when using the C preprocessor [7].

Compared to these systems, Marco adds safety checks while remaining expressive and language-agnostic. Marco relies on high-quality error messages from its target language compilers, while these unsafe system assume nothing. We believe that our assumption aligns well with compiler writers who want to give good explanations for compilation failures. The type of a Marco fragment constrains both its target-language and its non-terminal, allowing syntactic well-formedness checks for each fragment in isolation. Furthermore, Marco is the first language-agnostic system to check and provide an intentional naming discipline.

8.3 Using Messages from Black-Box Compilers

A few systems consume error messages from compilers and interpreters for a variety of reasons. SEMINAL takes error messages from the OCaml and g++ compilers and suggests changes for ill-formed programs [17]. Autoconf generates C/C++ programs, and sends them to C/C++ compilers. It checks error messages to determine whether or not some header files and some preprocessor symbols are available in the build host environment. The HelpMeOut system mines IDE logs to discover common fixes, and then proposes them to programmers based on which error messages are displayed [11]. Similar to Marco, each of these systems runs unmodified language compilers, and then inspects their error messages for clues. Unlike Marco, none of these systems is a macro system. To our knowledge, Marco is the first system that mines error messages from black-box compilers for safe code generation.

8.4 Error Reporting, Recovery, and Repair

Matured language processors devote significant effort to producing knowledgeable error messages, recovering from problematic fragments, and even repairing them. To explain failures well, some language processors choose top-down parsers [23] while bottom-up parsers analyze the stack of erroneous states and produce error messages [22]. To report as many error messages as possible, both hand-written and generated parsers skips several tokens from the point of the error to a synchronization token [8, 20].

Our oracles take advantage of knowledgeable error reports, deal with error recovery techniques, and advance the-start-of-art in error repairing. Since Marco reuses black-box compilers, it does not matter whether their parsers are top-down or bottom-up, hand-written or auto-generated. While the error recovery techniques in literature focus on syntactic failures, production-level C++ compilers skip tokens even due to a semantic error (e.g, redefined functions) to avoid spurious error messages. Our oracles handle both syntactic and semantic errors of shadowing downstream tokens and their error messages. Our speculative analysis shares the strategy with global error repair [4] in that both choose a repair candidate that advances the token index of the first error report, but our oracles produce semantic repair candidates. Our speculative repairs complement syntactic error repairs in literature.

9 Conclusion

As multilingual applications are becoming prevalent, so increases the need for macros that are expressive, safe, and language scalable at the same time. This paper has presented the first such macro processor called Marco. Our work is based on two key ideas. First, a plug-in facility provides target-language specific oracles implemented with off-the-shelf compilers and interpreters. In particular, we have identified three simple oracles, syntax, free-names, and captured-name.

They are sufficient for ensuring syntactic correctness and naming discipline of macros. Oracles are discharged by submitting specially crafted programs to the target-language processor and then analyzing the resulting error messages, if any. Second, a statically typed quote/unquote facility lets us maximize the target-language independent translation engine. Notably, the type of a fragment specifies its target language and its nonterminal, which the engine uses to invoke the appropriate language-specific oracles. Our evaluation of the Marco prototype supporting C++ and SQL demonstrates the viability of this approach. In future work, we will explore further target languages and safety guarantees.

References

1. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in dynamic web applications. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2008.
2. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2001.
3. C. Brabrand, M. I. Schwartzbach, and M. Vanggaard. The `metafront` system: Extensible parsing and transformation. *Electronic Notes in Theoretical Computer Science*, 82(3), Dec. 2003.
4. M. G. Burke and G. A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9:164–197, 1987.
5. A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.
6. W. Clinger and J. Rees. Macros that work. In *ACM Symposium on Principles of Programming Languages (POPL)*, 1991.
7. M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering (TSE)*, 28(12), Dec. 2002.
8. Free Software Foundation, Inc. GNU compiler collection (GCC) internals. http://gcc.gnu.org/onlinedocs/gccint/.
9. D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.
10. R. Grimm. Better extensibility through modular syntax. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.
11. B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do? Suggesting solutions to error messages. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2010.
12. M. Hirzel and R. Grimm. Jeannie: Granting Java native interface developers their wishes. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2007.
13. S. S. Huang and Y. Smaragdakis. Expressive and safe static reflection with MorphJ. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2008.

14. IBM Corporation. InfoSphere Streams: Stream processing system. http://www-01.ibm.com/software/data/infosphere/streams/.
15. I.-S. Kim, K. Yi, and C. Calcagno. A polymorphic modal type system for LISP-like multi-staged languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2006.
16. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *ACM Conference on LISP and Functional Programming (LFP)*, 1986.
17. B. Lerner, D. Grossman, and C. Chambers. SEMINAL: Searching for ML type-error messages. In *ML Workshop*, 2006.
18. E. Moggi, W. Taha, Z.-E.-A. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, 1999.
19. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *International Conference on Compiler Construction (CC)*, 2003.
20. T. Parr. *The Definitive ANTLR Reference*. The Pragmatic Programmers, May 2007.
21. J. Roskind. Parsing C, the last word. The comp.compilers newgroup, Jan. 1992. http://groups.google.com/group/comp.compilers/msg/c0797b5b668605b4.
22. S. Sippu and E. Soisalon-Soininen. A syntax-error-handling technique and its experimental analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5:656–679, October 1983.
23. B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.
24. B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000.
25. Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep typechecking and refactoring. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2008.
26. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4), July 2002.
27. D. Weise and R. Crew. Programmable syntax macros. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1993.
28. E. Westbrook, M. Ricken, J. Inoue, Y. Yao, T. Abdelatif, and W. Taha. Mint: Java multi-stage programming using weak separability. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010.