SMACC: BEHIND THE REFACTORINGS

Jason Lecerf & Thierry Goubier

May 17, 2017
# TABLE OF CONTENTS

1. Introduction

2. Overview of SmaCC
   - General workflow
   - From a user point of view

3. The rewrite engine
   - The engine
   - Anatomy of the rewrites
   - And RB

4. Final words
The Smalltalk Compiler Compiler is a parser generator for Smalltalk

Originally developed by John Brant & Don Roberts

Used in Moose, Synectique, CEA, RefactoryWorkers
Architecture to generate the front-end of compilers
- for DSL parsing
- for program analysis
- for program migration
- for refactoring and transformation of source code
- for compiler front-end implementation

SmaCC is written in Smalltalk and generate parsers in Smalltalk

It has the infrastructure for generating parsers in other programming languages
EXAMPLE OF USE

What you want to do:
- Find pattern in code written in a (niche) language
- Refactor these patterns into something new

What you will need:
- The grammar for your language (if it does not already exists in SmaCC)
- Your patterns and related transformations
- Your program
TABLE OF CONTENTS

1 Introduction

2 Overview of SmaCC
   - General workflow
   - From a user point of view

3 The rewrite engine
   - The engine
   - Anatomy of the rewrites
   - And RB

4 Final words
Automated generation of LR parsers

- **Input:** the specification of the grammar
- **Output:** parser for the grammar
  - can create arbitrary code run at parse time
  - can create an AST\(^1\)
Figure: SmaCC overall pipeline
In a slightly modified BNF\textsuperscript{2} form

\begin{verbatim}
CondExp
    :   ArithExp
        | BitwiseExp
        | RelationalExp
        | BoolExp
        | TernaryExp
    | <lpar> CondExp <rpar>
    | "defined(" Id <rpar>
    | Id
    | Number

#elseif NB_BITS < 32
#else
#endif
\end{verbatim}
The generation:

- Produces a DFA lexer (the scanner)
- Produces either LR(1) or LALR(1) parsers
- Can be augmented to support GLR parsing
- Generate methods for parse table transitions
  - not exactly the parse tables themselves (optimizations were done)
  - no tables, just methods for the lexer

**LR** Standard parser for context-free grammars

**LALR** Merge states resulting in a smaller memory footprint

**GLR** Try all the possible transitions for a state
The generated parser is a Smalltalk package containing:

- a Scanner class
- a Parser class
- the AST node classes
- a generic AST visitor

Running the parser on an input program:

- produce AST nodes instances
- execute arbitrary code given to the grammar
Expression

: Expression 'left' "+" Expression 'right'
  {left + right}
| Expression 'left' "-" Expression 'right'
  {left - right}
| Expression 'left' "*" Expression 'right'
  {left * right}
| Expression 'left' "/" Expression 'right'
  {left / right}
| Expression 'left' "^" Expression 'right'
  {left raisedTo: right}
| "(" Expression 'expression' ")" {expression}
| Number 'number' {number}
;

Number

: <number> 'numberToken'
  {numberToken value asNumber}
;
Expression

: Expression 'left' "+" 'op' Expression 'right'
  {{Expression}}
  | Expression 'left' "-" 'op' Expression 'right'
  {{Expression}}
  | Expression 'left' "*" 'op' Expression 'right'
  {{Expression}}
  | Expression 'left' "/" 'op' Expression 'right'
  {{Expression}}
  | Expression 'left' "^" 'op' Expression 'right'
  {{Expression}}
  | "(" Expression 'expression' ")" {{}}
  | Number

Number

: <number> {{Number}}

;
1 Introduction

2 Overview of SmaCC
   - General workflow
   - From a user point of view

3 The rewrite engine
   - The engine
   - Anatomy of the rewrites
   - And RB

4 Final words
EXTENDED SMACC PIPELINE

**Figure:** Extended SmaCC pipeline
PRECONDITIONS

- Parser for the language
- Enable GLR parsing in the grammar
- Declare a pattern token in the grammar
  - usually in between backquotes since they are used in barely any language
INPUT TO THE REWRITE ENGINE

Pattern

Metavariables

Transformation

Parser: MyExpressionParser

>>> 'a' + 'b' <<<

->

>>> 'a' 'b' + <<<
Input program:

\[(3 + 4) + (4 + 3)\]

Rewritten program using the rewrite engine:

\[3 4 + 4 3 + +\]
Figure: SmaCC rewriting process
Metavariables can match any nodes (unless specified otherwise)
  - can be modified to match list of nodes or specific types of nodes

Use the GLR parser to parse the pattern

Try all the possible starting symbols (entry points) of the grammar
  - ex: Methods, expressions, method call
  - not only the top entry point (often ”Program”)

Get all the possible ASTs for the pattern
1. Parse *program* using the GLR parser
   - Produces the program AST

2. Parse *pattern* using the GLR parser
   - If there are conflicts: we get a forest of trees
   - If there are pattern nodes (metavariables): we get a forest of trees if valid for the grammar
   - Otherwise: we get a single tree
if currentToken = patternToken then
    for all symbol in \{tokens OR non-terminal nodes\} do
        actionsToProcess ← all possible LR actions for symbol
        for all LR action in actionsToProcess do
            Check if action was not already performed
            if symbol = Token OR (symbol = Node AND action = reduction) then
                Add a token interpretation to the current token
                Try to perform current LR action
            else if symbol = Node AND action = shift then
                stateStack add new ambiguous state
            end if
        end for
    end for
end if

Remove current pattern token state
MATCHING OF THE PATTERN

Based on ambiguity handling by GLR

- Reuses the parser for your grammar
- Based on the parse tables of said parser

- **When parsing a pattern token:**
  - Try all the valid action-token combinations (transitions) for the current state
  - When conflicts arise (i.e. more than one transition is possible), fork the parser
ANATOMY OF THE REWRITE ENGINE

Figure: SmaCC rewriting process
UNIFICATION ALGORITHM

**Require:** $\text{patternForest, programTree}$

for all $\text{programNode in programTree}$ do

> Depth first traversal

    for all $\text{patternTree in patternForest}$ do

        for all $\text{patternNode in patternTree}$ do

            if $\text{patternNodeclass} = \text{programNodeclass}$ then

                Tries to match $\text{patternNode}$ subnodes with $\text{programNode}$ subnodes

            else

                continue

            end if

        end for

    end for

end for
Program

\[ x + 2 \times y \]

Pattern matching

`a` `op{nodeName: #Op}` `b`

---

**Figure:** SmaCC rewriting process
ArgumentListPar
: "(" (Expression 'argument'
(""," Expression 'argument')* )? ")" 

Node equality Class are identical and every subnodes, subtokens match

Token equality Both values are identical (same string)
- Here: the left and right parenthesis tokens

Node collection equality Every individual node matches
- Here: the list of $n - 1$ commas

Token collection equality Every individual token matches
- Here: the list of $n$ Expression arguments
Program

\[ x + 2 \times y \]

MetavARIABLE binding

\['a` `op\{nodeClassName: #Op\}` `b` \rightarrow `a` `b` `op`\]

Figure: SmaCC binding & rewriting process
ANATOMY OF THE REWRITE ENGINE

Figure: SmaCC rewriting process
When nodes match, they are stored in the context.

Even if it seems intuitive, SmaCC does not perform AST rewriting.

- Is rewriting a part of a tree really that simple?
- And what if I rewrite in another programming language?

When rewriting, only transform the source of the nodes to the source of the transform.

- i.e.: the source of the transform is not parsed.
WHAT ABOUT RB?

RB and SmaCC share the same creators

But Smalltalk is a very simple language (to parse)
- It is simple to specify a pattern tree directly
- Use a bit the parser to complete the pattern tree
- Rule: a pattern is valid Smalltalk code
- But may match a slightly different tree (message)

Subtree matching algorithm is exactly the same as in SmaCC
- For example, see
  - RBPatternMethodNode>> #match:inContext:
1 Introduction

2 Overview of SmaCC
   - General workflow
   - From a user point of view

3 The rewrite engine
   - The engine
   - Anatomy of the rewrites
   - And RB

4 Final words
SMACC APPROACH

- Put some (most) of the complexity in the parser
- Use reflexivity on the grammar
  - The grammar specify all correct phrases (all valid sequences of tokens)
  - The AST directives specify all possibles nodes and trees of nodes (complete type specification)
  - The parser contains all that information in the state tables
  - Query it!
- Match and rewrite (on a large scale... over a million lines of code)
CONCLUSION

- SmaCC is a parser generator extended with pattern matching and rewriting capabilities
- Uses the parser as a way to reflect on the grammar and build pattern trees
- Tree traversal for matching is depth first (ASTs are not very deep)
- Generalization of the Refactoring Browser in the case of an "arbitrary" grammar
- Used extensively by John Brant, Thierry Goubier and others...
Tutorial and documentation book for SmaCC
Thank you!