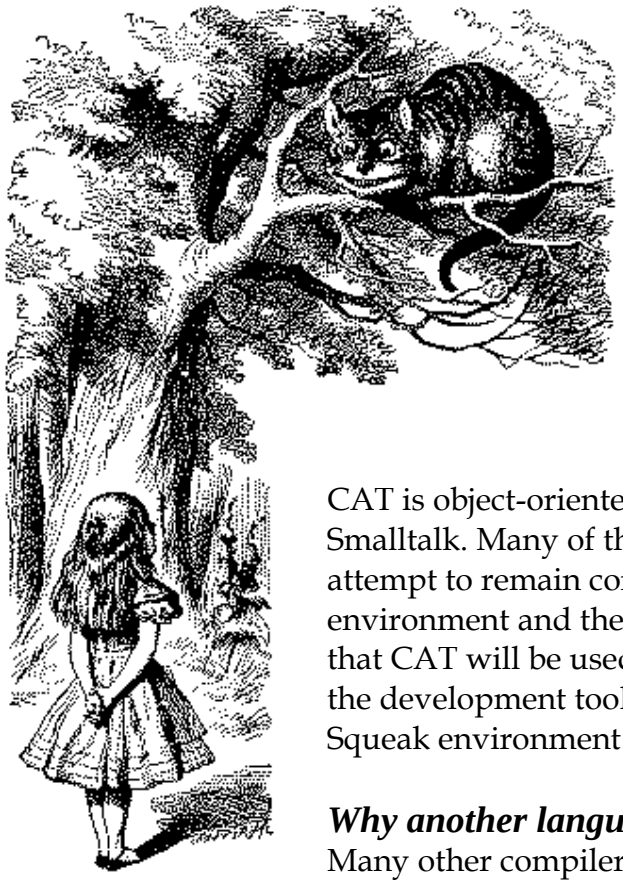
The CAT *Language: An Overview*

by
Jamie Douglass



Introduction

The Context-free Attributed Translation (CAT) language is intended to make it easier to develop translators for computer programming languages. CAT combines Backus-Naur Form (BNF), Regular Expressions and an Attribute Grammar (coded as Smalltalk methods) to allow users to specify both the syntax and semantics of their programming language(s). Another interpretation of the language name CAT is Computer Any-programming-language Translator.

CAT is object-oriented, and was developed in Squeak Smalltalk. Many of the design decisions are made in an attempt to remain consistent with the Squeak environment and the Smalltalk language. This is in hopes that CAT will be used by this community and to leverage the development tools that already exist within the Squeak environment.

Why another language?

Many other compiler-compilers have been developed such as LEX, FLEX, YACC, Bison and ANTLR, so why develop another one? First of all, CAT provides this language translation capability to the Squeak environment and its community of users. The second reason is the addition of features along with their integration and implementation which enhance CAT's ease of use over other language translation tools. The remainder of this paper is devoted to describing these features and illustrating how they enhance the user's experience in language development.

Goals and Objectives

Simple and Intuitive

By using the familiar notations of Backus-Naur Form (BNF), Regular Expressions and Attribute Grammars, CAT is easy for those knowledgeable in language design and development to learn and use. Here is a typical rule to specify an identifier.

indentifier ::= Letter {Letter | Digit}.

The same notation is available to specify production rules for both scanners and parsers.

CAT grammars are treated as a programming language where production rules are processed from left to right. The parsing technique used is a Top-Down approach except when left recursion is detected. Only in this case is a Bottom-Up approach used. The reason for this is to make it easier for users to trace the execution of their grammar using the Smalltalk symbolic debugger.

Left Recursion and Ambiguous Expressions

Users can express rules using left recursion and ambiguous expressions. Here is such a rule.

Expression ::= Expression ["+" | "-"] Expression | Number.

This would expand the left Expression first resulting in a parse tree for left associative addition and subtraction. The rule below would be right associative expanding the right Expression first.

Expression ::= Expression ["+" | "-"] Expression! | Number.

In addition to the exclamation point, **!**, users can disambiguate grammars using slash mark, **/**, following a non-terminal. This prevents recursion of the rule, forcing other references to be expanded.

Abstract Syntax Tree (AST) and Precedence

It is easy to create an AST in CAT. The up arrow **^** on a non-terminal creates an AST node. The tokens associate left or right governed by which side of the non-terminal the arrow appears. A single digit or capital letter between the arrow and the non-terminal indicates the precedence level of the node. Non-terminals in single quotes are omitted from the parse tree. Rules with an equal sign, **=**, do not generate a parse tree node. Like-wise rules with single colon and equal sign, **:=**, only generate a parse tree node if they have more that one child node. Example at the end of this section illustrates the use of these features.

CAT Overview

Object-oriented Attribute Grammar Representation in Smalltalk

All grammar elements and parse tree nodes are implemented by a single Smalltalk class. Users extend the parse tree node classes with the classes for the various language constructs they are designing. These classes implement the attributes in the grammar as Smalltalk methods. A clause at the end of a production rule associates the class name with that particular rule. When the rule is reduced, an instance of this class is created for the parse tree. The following rule associates the CATRule class with the syntaxRule production.

syntaxRule ::= ruleName '::=' definition [';' treeNodeClass]'.'; CATRule.

Attributes may also be invoked inline during the parser execution. This supports labeling elements apart of the parse, actions, semantic expressions and predicates. The list of attributes follow an @ separated by commas. Each attribute name may have a leading colon which means it consumes the element before it in the parse tree as input. It may also have a trailing colon which means it produces an output which is added to parse tree at the current position. Both leading and trailing colons create an attribute pipe. A question mark, **?**, at the end tests the attribute as a predicate.

Portable

Because CAT is implemented within Squeak, it is highly portable and available on all the systems where Squeak runs. The Squeak environment is open-source and bit-compatible across over 40 platforms including Windows, Mac and Linix.

Performance

At this stage of development, ease of use, understandability and correctness are more important than performance. Current performance still needs to insure that the users experience is not adversely affected. It is hoped, however, that future development of CAT will support high-performance execution.

Summary of CAT Features

- Extended Backus-Naur Form (BNF)
  - Nonterminals rule name identifiers
  - Terminals in double quotes **".."** are included in parse tree.
  - Basic Sequences
    - Series of items
    - Grouping of items **()**
    - Alternatives **|**
- Regular expressions
  - Optional items in **[]**
  - Zero or more occurrences in **{}** or **[]***
  - One or more occurrences in **{}+** or **[]+**
- Set operations
  - Intersection a **&** b means a and b
  - Exclusion a ~ b means a but not b.
- Standard and user defined non-terminal names for
  - Character sets such as Character, Letter and Digit
  - Token processing routines
  - Names can be over ridden by user production rules.
- Automatic Generation of Parse Tree from production rules.
- Supports Left Recursion
- Left-Right disambiguation of parse tree.
- User disambiguation preferences
  - Non-terminal with exclamation point **!** expanded first.
  - Non-terminal with slash mark **/** expanded once.
  - Left expansion (same as default) **E! + E/**
  - Right expansion **E/ + E!**
- Supports Abstract Syntax Tree (AST) generation
  - No parse tree node production rules with **=**
  - More than one child parse tree node production rules with **:=**
  - Terminals in single quotes **'..'** not included in parse tree.
  - AST node creation
    - AST grows to Left **^""**
    - AST grows to Right **""^**
- Precedences for AST node terminals
  - Terminals have 36 precedence levels p= 0..9, A..Z (default is zero).
  - Left association with precedence with **^p""**
  - Right association with precedence with **""p^**
- Attribute grammar
  - Inline attributes in **@attribute name list**
  - Action and semantic expressions
  - Predicate attributes followed by question mark **?**
  - Attributes Consume/Produce parse stream items
    - Consume is **:attribute**
    - Produce is **attribute:**
    - Both Consume and Produce (pipe) is **:attribute:**.
- Attributes in Smalltalk classes extend parse tree and token nodes classes
  - Associated to rule by clause at end of rule **; MyClassName**.
- Unified specification of language name, syntax and token rules, and items to ignore
- Generates both scanners and parsers

Example

The Calculator example[1] illustrates how CAT can be used to implement a simple interpreter. The following is the CAT grammar specification of Calculator.

```
Calculator
Syntax:
        file ::= {statement ';'}+; Calculator.

Rules:
        statement := "if"^  statement 'then' statement ['else' statement]
                | "print"^  (string | statement)
                | identifier "="^ statement
                | expression.

        expression = expression! operator expression
                | '(' subexpression ')'
                | identifier
                | number
                | "-"4^  expression.
        `! on left expression added for emphasis, left disambiguation is default.`

        operator = ^2"+" | ^2"-" | ^3"*" | ^3"/".

        subexpression := expression.

Tokens:
        identifier ::= Letter {Letter | Digit | "_"}.
        number ::= {Digit}+.
        string ::= '"' {Character ~ '"' | '"' '"""' } '"'.

Ignore:
        WhiteSpace = Space | Tab | LineFeed | Cr.
```

Attributes

SyntaxNode subclass: #Calculator instanceVariableNames: 'values'

```
execute
        Transcript openLabel: 'Calculator'.
        children do: [:each |
                self execute: each]

execute: aNode
        aNode name isUnary ifTrue: [
                ^self perform: (aNode name, #:) asSymbol with: aNode value].
        ^self perform: aNode name with: aNode value

statement: aCollection and subexpression: aCollection
        ^self execute:  aCollection first
```

[1] A Two-Page Introduction to ANTLR by Stephen A. Edwards, Columbia University

Statement and Expression Attributes

```
if: aCollection
        | test |
        test := self left: aCollection.
        test ~= 0 ifTrue: [
                ^self middle: aCollection]
          ifFalse: [
                aCollection size> 2 ifTrue: [^self  right: aCollection]]

print: aCollection
        | line |
        line := self left: aCollection.
        Transcript show: line; cr.
        ^line

= aCollection
        | var value |
        var := aCollection first value.
        value := self right: aCollection.
        self valuesAt: var put: value.
        ^value

+ aCollection
        ^(self left: aCollection) + (self right: aCollection)

- aCollection
        aCollection size = 1 ifTrue: [
                ^(self left: aCollection) negated].
        ^(self left: aCollection) - (self right: aCollection)

* aCollection
        ^(self left: aCollection) * (self right: aCollection)

/ aCollection
        ^(self left: aCollection) / (self right: aCollection)

string: aString
        ^aString

number: aString
        ^ aString asNumber

identifier: aSymbol
        ^self valuesAt: aSymbol
```

Utility Attributes

```
values
        values isNil ifTrue: [
                values := Dictionary new].
        ^values

valuesAt: aSymbol
        ^self root values at: aSymbol ifAbsent: [
                Transcript show: 'Unrecognized: ', aSymbol]

valuesAt: aSymbol put: anObject
        ^self root values at: aSymbol put: anObject

 left: aCollection
        ^self execute: aCollection first

middle: aCollection
        ^self execute: (aCollection at: 2)

right: aCollection
        ^self execute: aCollection last
```

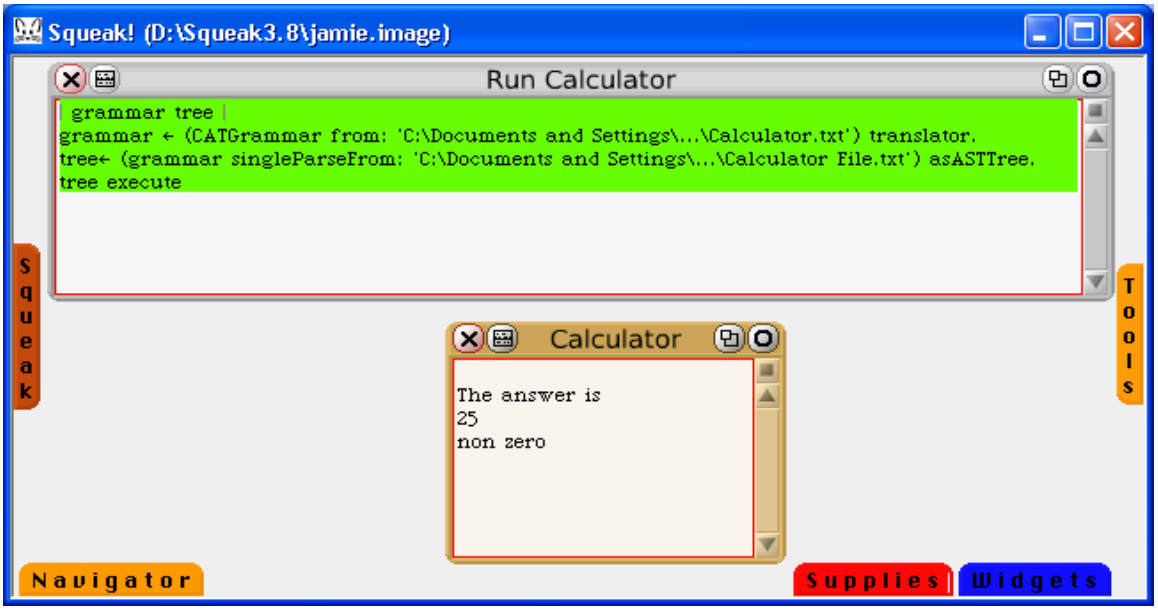
Input File

Run Calculator and Output Window

Here is the input file processed by the run of Calculator.

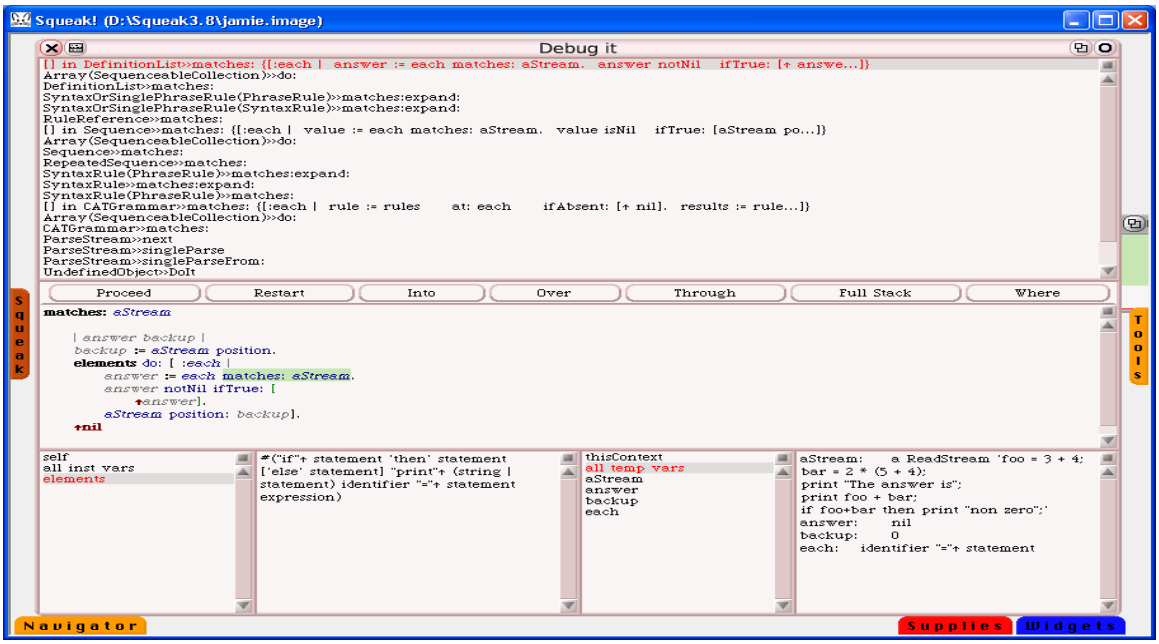
```
foo = 3 + 4;
bar = 2 * (5 + 4);
print "The answer is";
print foo + bar;
if foo+bar then print "non zero";
```

The code highlighted in green is executed in the Run Calculator window producing the output in the Calculator window below it.



Trace of Calculator

Here are two sample windows. This first one is during the execution of the Calculator parser. The parser is looking at alternatives for the first statement in the input file. The lower left hand window pane lists the candidates being considered. The lower right hand window pane shows that the assignment syntax is about to be matched.



The screen shot below is taken during the execution of the Calculator interpreter. The if statement within the input file is being processed. The sum of foo and bar is the test value of 25, and the then clause is about to be executed. The value of aCollection shows that the node named statement has only one child (**->**). The node + has two children, the identifiers foo and bar. The second item in aCollection is the chain of nodes: statement, print and the string 'non zero'.