

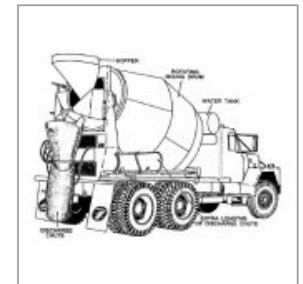


# Mistaeks I Hav Made

"Good judgement is the result of experience ... Experience is the result of bad judgement." — Fred Brooks

## Concrete to Abstract Syntax Transformations with SableCC

I'm known to grumble about Java technologies, especially those used for enterprise development but I have to grudgingly admit that there are some pretty cool Java things out there that don't suffer from "enterprise quality". One example is [SableCC](#), my top choice of parser generator. I often use SableCC to define the syntax of configuration files and other data formats that must be read and/or edited by people. It makes for formats that are more human-friendly than XML but are still easy to parse and process.



### Why SableCC?

SableCC has an elegant design that cleanly separates the grammar definition from the programming language used to implement the parser, meaning you don't have to embed Java code into the grammar. Instead, SableCC creates classes to represent the syntax tree and abstract [visitor](#) classes that you extend to perform transformations, semantic analysis and other processing by walking the tree.

The main drawback of the current release of SableCC, 2.18.3, is that the generated classes represent the *concrete* syntax tree, not the *abstract* syntax tree that you really want to work with. Defining the abstract syntax tree and writing transformations from to concrete to abstract tree is quite an effort, and then you have to write your own visitor classes to process that tree.

The latest beta version of SableCC has a feature that addresses this problem: you can define an abstract syntax in the grammar and specify how the concrete syntax relates to the abstract syntax. The generated parser translates from the concrete syntax to the abstract syntax as it parses and returns you the abstract syntax tree. The generated visitor classes all work on the abstract syntax tree so your software is shielded from the complexities of the concrete syntax.

Currently this feature has virtually no documentation, so here's a short tutorial. I assume you already have some experience of using SableCC. If not, give it a try; I think it's really good.

### Concrete vs. Abstract Syntax

What's so great about concrete to abstract syntax transformations? Let's use the following grammar of simple arithmetic expressions as an example.

#### Tokens

```
add = '+';
sub = '-';
mul = '*';
div = '/';
left_paren = '(';
right_paren = ')';
number = [0-9]+;
whitespace = (' ')+;
```

#### Ignored Tokens

```
whitespace;
```

#### Productions

```
expr = {add} [left]:expr add [right]:factor
      | {sub} [left]:expr sub [right]:factor
      | {factor} factor;

factor= {mul} [left]:factor mul [right]:value
        | {div} [left]:factor div [right]:value
        | {value} value;

value = {number} number
        | {parens} left_paren expr right_paren;
```

Using this grammar to parse the expression "(1 + 2) \* 3" produces the following concrete syntax tree.

```
AFactorExpr
|
+- factor : AMulFactor
  |
  +- left : AValueFactor
    |
    +- value : AParenValue
      |
      +- "("
      |
      +- expr : AAddExpr
```

```

|
|
|   +- left : AFactorExpr
|   |
|   |   +- factor : AValueFactor
|   |   |
|   |   |   +- value : ANumberValue "1"
|   |
|   |   +- "+"
|   |
|   |   +- right : AValueFactor
|   |   |
|   |   |   +- value : ANumberValue "2"
|   |
|   +- ")"
|
+- "*"
|
+- right : AValueFactor
|
+- value : ANumberValue "3"

```

As you can see, the concrete syntax tree is pretty complex. This complication affects code that analyses this tree and couples that code to the operator precedence rules defined by the grammar. One result of this coupling is that the analysis code will frequently break when we change the language.

Ideally, our code would process the tree below, which represents the expressions themselves rather than the particular notation we use to represent expressions.

```

AMulExpr
|
+- left : AAddExpr
|   |
|   |   +- left : ANumberExpr "1"
|   |   |
|   |   |   +- right : ANumberExpr "2"
|   |
|   +- right : ANumberExpr "3"
+- right : ANumberExpr "3"

```

This is the abstract syntax tree of the concrete syntax tree shown above. The abstract syntax of our expression language can be also be represented as a grammar:

```

expr = {add} [left]:expr [right]:expr
      | {sub} [left]:expr [right]:expr
      | {mul} [left]:expr [right]:expr

```

```
| {div} [left]:expr [right]:expr  
| {number} number;
```

If only we could tell the parser how to translate the concrete syntax to this abstract tree...

With SableCC 3 we can! A SableCC 3 grammar specifies both the concrete and abstract syntaxes with the same notation and annotates the concrete syntax to define how it is translated to the abstract syntax.

## CST to AST Transformations

The abstract syntax tree is defined in a new section called "Abstract Syntax Tree" that goes after the productions in the grammar file. Once we've defined our abstract syntax tree, we then annotate the concrete syntax tree with translations to the abstract syntax tree.

The translation for each concrete production is specified in two parts: the concrete production is annotated to specify which abstract production it maps to and each alternative of the production is annotated to specify how to translate it to an alternative of the abstract production.

Starting with the concrete `expr` production, we want to translate that into the abstract `expr` production we have defined in our "Abstract Syntax Tree" section. To do so, we use the following annotation written between curly braces:

```
expr {-> expr} = ...
```

Note that the concrete and abstract grammars have separate namespaces: a production in the "Productions" section can have the same name as one in the "Abstract Syntax Tree" section.

We then need to specify how the alternatives of the concrete `expr` production map to those of the abstract `expr` production. We must map the concrete `add` and `sub` productions to new nodes of the appropriate abstract type. Here's what these transformations look like:

```
expr {-> expr} =  
  {add} [left]:expr add [right]:factor  {-> New expr.add(left.expr, right.expr)}  
  | {sub} [left]:expr sub [right]:factor  {-> New expr.sub(left.expr, right.expr)}  
  ...
```

Again, the transformations are specified in curly braces and preceded by an arrow. New nodes are created with the **New** keyword followed by the name of the node of the abstract syntax tree to create and its children, if any, as arguments. The identifier of an abstract syntax tree node is the name of the production suffixed with the name of the alternative. E.g. "expr.add" identifies the add alternative of the `expr` abstract production, "expr.sub" identifies the sub alternative.

The child nodes passed as arguments have already been transformed to abstract syntax by the time they are passed to the new node. You identify a transformed node by the name of the child in the concrete grammar suffixed by the type that it has been transformed to. E.g. "left.expr" identifies

the left subexpression that has been transformed to an abstract expr production.

The last alternative of the concrete expr production is `expr.factor`. This exists in the concrete syntax to define operator precedence and doesn't exist in our abstract syntax. We want to replace it with the transformation of its only subtree, also named `factor`.

```
expr {-> expr} =
  {add} [left]:expr add [right]:factor  {-> New expr.add(left.expr, right.expr)}
  | {sub} [left]:expr sub [right]:factor {-> New expr.sub(left.expr, right.expr)}
  | {factor} factor                      {-> factor.expr};

factor {-> expr} =
  {mul} [left]:factor mul [right]:value {-> New expr.mul(left.expr, right.expr)}
  | {div} [left]:factor div [right]:value {-> New expr.div(left.expr, right.expr)}
  | {value} value                        {-> value.expr};

value {-> expr} =
  {number} number                        {-> New expr.number(number)}
  | {parens} left_paren expr right_paren {-> expr.expr};
```

The transformations of the `factor` and `value` productions follow the same pattern.

## Transforming Lists

Another common difficulty with SableCC concrete syntax trees is that arbitrarily sized lists often have an unwieldy representation in the generated Java code. As an illustration, let's extend the expression language to support functions:

```
value = {number} number
  | {parens} left_paren expr right_paren
  | {function} [name]:identifier left_paren [args]:arg_list right_paren;

arg_list =
  {single} [arg]:expr
  | {multiple} [arg]:expr comma [rest]:arg_list;
```

The concrete definition of argument lists is pretty awkward. Our analyser has to deal with the "single" and "multiple" alternatives and follow the "rest" link in the multiple alternative to collect all the arguments. It would be much easier to analyse if the argument expressions were stored in a collection.

Again, SableCC 3 come to the rescue. We can transform the concrete `arg_list` into a list of `exprs`. In the generated code, this list is stored in a standard Java List.

Firstly, we'll need to add a function alternative to our abstract syntax:

```

expr = {add} [left]:expr [right]:expr
      | {sub} [left]:expr [right]:expr
      | {mul} [left]:expr [right]:expr
      | {div} [left]:expr [right]:expr
      | {number} number
      | {function} [name]:identifier [args]:expr*;

```

The "expr\*" type indicates a list of zero or more expr clauses.

Now we need to annotate the concrete function syntax with the transformation to the abstract syntax:

```

value {-> expr} =
  {number} number                                {-> New expr.number(number) }
  | {parens} left_paren expr right_paren          {-> expr.expr}
  | {constant} [name]:identifier                  {-> New expr.constant(name) }
  | {function} [name]:identifier left_paren [args]:arg_list right_paren {-> New expr.function(name, [args.expr]) };

```

Note the square brackets around the "[args.expr]" argument. They indicate that the "args" clause is translated into a list of exprs, not a single expr.

Finally we need to define how the different alternatives of the args\_list are collected into a single list.

```

arg_list {-> expr*} =
  {single} [arg]:expr                            {-> [arg.expr]}
  | {multiple} [arg]:expr comma [rest]:arg_list {-> [arg.expr, rest.expr]};

```

The annotation of the arg\_list production "{-> expr\*}" indicates that it is translated to a list of expr elements. The annotations of the "single" and "multiple" alternatives specify how to build that. The single transformation builds a list with one element, the subexpression named arg. The multiple transformation prepends the subexpression named arg to the list created by transforming the rest of the arg\_list.

## Conclusion

That covers the basics of concrete-to-abstract syntax transformations in SableCC 3. I hope this overview sheds some light on a powerful but underdocumented feature. Concrete-to-abstract syntax transformations let SableCC shine in an iterative development process. By working on the abstract syntax, analysis code is shielded from changes to concrete syntax. You can add to or change the grammar during the project without breaking a lot of existing code. I've found it easy to design a language test-first, repeatedly writing unit tests to specify syntax and semantics and then changing the grammar and interpreter to make the tests pass. This has let me play with language design and quickly experiment with new language features.

All the source code used in the examples above and IntelliJ project files are available in a [Zip archive](#).

Update 28/09/2005: SableCC 3.0 is now a stable release, so you can happily use it in production systems.

Copyright © 2005 Nat Pryce. Posted 2005-09-02. [Share it.](#)

0 Comments    **Mistaeks I Hav Made**

 Login ▾

 Recommend   9     Share

Sort by Best ▾




Start the discussion...

Be the first to comment.

ALSO ON MISTAEKS I HAV MADE


Property-Based TDD at XP Day London 2012

1 comment • 3 years ago

 **Paul Holser** — Very nice! I have been working on a library for supplying JUnit theories with lots of random values for their parameters, junit-quickcheck (<http://github.com/pholser/juni....>) I'd be interested in your feedback. It's got a ways


Affordance Open-Space at XP Day

2 comments • 2 years ago

 **jonericsn** — Surprised to find myself paying more attention to the "Hinderance" section of this article. Curious. I guess it is evolution at work; our species survived by focusing on threats... which is probably why the evening news doesn't have a lot of


Message Obsession

2 comments • 2 months ago

 **nat\_pryce** — I'm using Smalltalk's OO terminology. In this example, 'moveNorth()' is a message sent to the robot object, giving it a goal to achieve. On receiving the message, the object looks up and executes its method for achieving that goal. The distinction

State vs Interaction Based Testing

1 comment • 5 years ago

 **Curtis Cooley** — I just ran across this and think it's great. I think mocks are being accused of "bad OO" when it's really how they are used to microtest that is the issue. It's easy to blame the framework and not take responsibility for how one uses the