

GenDoc

Gen: A Java Package for Constructing and Manipulating Abstract Syntax Trees

Gen is a Java preprocessor that extends the Java language with special syntax to make the task of handling Abstract Syntax Trees (ASTs) easier. Two classes are used by Gen: the class `Tree` that captures an AST (with subclasses `VariableLeaf`, `IntegerLeaf`, `FloatLeaf`, `StringLeaf`, and `Node`), and the class `Trees` that captures a list of ASTs (see the [Appendix](#) for the detailed API). A `Tree` is a tree-like data structure, which is used for representing various tree-like data structures used in compiler construction.

For example, the Java expression

```
new Node("Binop",
    Trees.nil.append(new VariableLeaf("Plus"))
    .append(new VariableLeaf("x"))
    .append(new Node("Binop",
        Trees.nil.append(new VariableLeaf("Minus"))
        .append(new VariableLeaf("y"))
        .append(new VariableLeaf("z")))))
```

constructs the AST `Binop(Plus,x,Binop(Minus,y,z))`. To make the task of writing these tree constructions less tedious, Gen extends Java with the syntactic form `#< >`. For example, `#<Binop(Plus,x,Binop(Minus,y,z))>` is equivalent to the above Java expression. That is, the text within the brackets `#< >` is used by Gen to generate Java code, which creates the tree-like form (an instance of the class `Tree`) that represents this text. Values of the class `Tree` can be included into the form generated by the `#< >` brackets by "escaping" them with a backquote character (```). The operand of the escape operator (the backquote operator) is expected to be an expression of type `Tree` that provides the value to "fill in" the hole in the bracketed text at that point (actually, an escaped string/int/float is also lifted to a `Tree`). For example, in

```
Tree x = #<join(a,b,p)>;
Tree y = #<select(`x,q)>;
Tree z = #<project(`y,A)>;
```

`y` is set to `#<select(join(a,b,p),q)>` and `z` to `#<project(select(join(a,b,p),q),A)>`. There is also bracketed syntax, `#[]`, for constructing instances of `Trees`.

The bracketed syntax takes the following form:

bracketed ::=	"#<" expr ">"	a Tree construction (an AST)	
<ac:structured-macro ac:name="unmigrated-wiki-markup" ac:schema-version="1" ac:macro-id="9de36748-b721-48a5-bf4d-de17d848ccad"><ac:plain-text-body><![CDATA["#[" arg " , " ... " , " arg "]"	a Trees construction (a list of ASTs)]]></ac:plain-text-body></ac:structured-macro>
expr ::=	name	the representation of a variable name	
integer	the representation of an integer		
float	the representation of a float number		
string	the representation of a string		
"" name	escaping to the value of name		
"(" code ")"	escaping to the value of code		
name "(" arg " , " ... " , " arg ")"	the representation of a Node with zero or more children		
"" name "(" arg " , " ... " , " arg ")"	the representation of a Node with escaped name		
arg ::=	expr	the representation of an expression	
"..." name	escaping to a list of ASTs (Trees) bound to name		
"..."(" code ")"	escaping to a list of ASTs (Trees) returned by code		

where `code` is any Java code. The `#< `(code) >` embeds the value returned by the Java code `code` of type `Tree` to the term representation inside the brackets. For example, `#< { f(6, ... r, g("ab", (kⓧ)), y) } >` is equivalent to the following Java code:

```

new Node(f,
    Trees.nil.append(new IntegerLeaf(6))
        .append(r)
        .append(new Node("g",Trees.nil.append(new StringLeaf("ab"))
            .append(k(x))))
        .append(y))

```

If $f = "h"$, $y = \#<m(1, "a")>$, and $k \times$ returns the value $\#<8>$, then the above term is equivalent to $\#<h(6, g("ab", 8), m(1, "a"))>$. The three dots (...) construct is used to indicate a list of children in a Node. Since this list is an instance of the class `Trees`, the type of `name` in `...name` is also `Trees`. For example, in

```

Trees r = #[join(a,b,p),select(c,q)];
Tree z = #<project(...r)>;

```

`z` will be bound to $\#<project(join(a,b,p),select(c,q))>$. Finally, to iterate over `Trees`, use a for-loop:

```

for ( Tree v: #[a,b,c] )
    System.out.println(v);

```

Gen provides a case statement syntax with patterns. Patterns match the `Tree` representations with similar shape. Escape operators applied to variables inside these patterns represent variable patterns, which "bind" to corresponding subterms upon a successful match. This capability makes it particularly easy to write functions that perform source-to-source transformations. A function that simplifies arithmetic expressions can be expressed easily as:

```

Tree simplify ( Tree e ) {
    match e {
        case plus(`x,0): return x;
        case times(`x,1): return x;
        case times(`x,0): return #<0>;
        case _: return e;
    }
}

```

where the `_` pattern matches any value. For example, `simplify(#<times(z,1)>)` returns $\#<z>$, since `times(z,1)` matches the second case pattern. The syntax of the case statement is:

case_stmt ::=	"match" code "{" case ... case "}"	
case ::=	"case" pattern ":" code	
pattern ::=	name	exact match with a name
integer	exact match with an integer	
float	exact match with a float number	
string	exact match with a string	
" " name	match any Tree and bind name to the matched value	
name "(" arg ", " ... " ", arg ")"	match with a Node with a given name and match the Node children	
" " name "(" arg ", " ... " ", arg ")"	match the Node children and bind name to the Node name	
" _ "	match any Tree	
arg ::=	pattern	match with a pattern
"..." name	match with a list of ASTs (Trees) and bind name to the matched list	
"..."	match any arguments	

For example, the pattern ``f(...r)` matches any `Node`: when it is matched with `#<join(a,b,c)>`, it binds `f` to the string `"join"` and `r` to the list `#[a,b,c]`. Another example is the following function that adds the terms `#<8>` and `#<9>` as children to any `Node e`:

```
Tree add_arg ( Tree e ) {  
  match e {  
    case `f(...r): return #<f(8,9,...r)>;  
    case `x: return x;  
  }  
}
```

The special keyword `fail` is used in Gen to abort the current matching and skip to the next case. For example

```
match e {  
case `f(...r,join(`x,`y),...s):  
  if (x.equals(y))  
    return #<f(...r,`x,...s)>;  
  else fail;  
case `x: return x;  
}
```

will transform `join({{x,}}y)` to `x` if `x` is equal to `y` (it is not permitted to use a variable twice in a pattern, that is, `join({{x,}}x)` is an illegal pattern).

Appendix: The Tree API

The class `Tree` captures an AST, and the class `Trees` captures a list of ASTs. A `Tree` is a tree-like data structure, which is used for representing various tree-like data structures used in compiler construction.

```

abstract class Tree {
    public boolean is_node ();           // is this a Node?
    public boolean is_variable ();       // is this a VariableLeaf?
    public boolean is_long ();           // is this a LongLeaf?
    public boolean is_string ();         // is this a StringLeaf?
    public boolean is_double ();         // is this a DoubleLeaf?
    public String variableValue ();       // return the variable name
    public long longValue ();             // return the long value
    public String stringValue ();         // return the string value
    public double doubleValue ();         // return the double value
}
class VariableLeaf extends Tree {
    public VariableLeaf ( String s );
    public String value ();
}
class LongLeaf extends Tree {
    public LongLeaf ( long n );
    public long value ();
}
class DoubleLeaf extends Tree {
    public DoubleLeaf ( double n );
    public double value ();
}
class StringLeaf extends Tree {
    public StringLeaf ( String s );
    public String value ();
}
class Node extends Tree {
    public Node ( String n, Trees cs );   // construct a new Node with a given name n and children cs
    public String name ();               // Node label
    public Trees children ();             // Node children
}

```

where the Trees class represents a list of ASTs (a linked list):

```

class Trees implements Iterable<Tree> {
    public Tree head();                  // the head of the list
    public Trees tail();                  // the rest of the list (the list without the head)
    public Trees ( Tree e, Trees t );    // construct a new list with head e and tail t
    public final static Trees nil;       // the empty list (same as #[])
    public Trees cons ( Tree e );        // put the element e before the list
    public Trees append ( Tree e );      // append the element e at the end of list (does not change 'this')
    public Trees append ( Trees s );     // append the list s at the end of list (does not change 'this')
    public boolean is_empty ();           // is this an empty list?
    public int length ();                 // return the list length
    public Trees reverse ();              // return the reverse of the list (does not change 'this')
    public boolean member ( Tree e );    // is e a member of the list?
    public Tree nth ( int n );            // return the nth list element
}

```

To iterate over Trees, one can use Java iterators:

```

Trees ts = ...;
for( Tree e: ts )
    System.out.println(e);

```