Handbook of ¬<><∪∪ in English

Tetsuhiko Koto
koto@takopen.cs.uec.ac.jp

Alpha Release for Version 0.71

Overall Structure

The input for the ¬<><∪∪ compiler compiler is converted into a sequence of tokens by the process same as Java, but the tokens of ¬<><∪∪ are

A Token	Description
IDENTIFIER	not starting with `$`
Keyword	starting with `$`
CHAR	a Java character literal
STRING	a Java string literal
etc.	

The grammar of the input for the ¬<><∪∪ is drawn by EBNF. The notation is

Meta-notation	Description
Italic or *italic*	nonterminals
ITALIC	terminals (symbol)
`Typewriter`	terminals (itself)
expr1 | *expr2*	alternative
$expr^*$	zero or more
$expr^+$	one or more
$expr_{opt}$	optional

The sequence of tokens should be structured by the following EBNF. The goal symbol is *Root*.

Root ::= *PackageDeclaration*$_{opt}$
ConstructorScope$_{opt}$
definition*

PackageDeclaration ::= `$package` *JavaName* `;`

JavaName ::= *IDENTIFIER* (`.` *IDENTIFIER*)*

ConstructorScope ::= `$protected $constructor ;`

definition ::= *SubtokenDefinition*
| *TokenDefinition*
| *AliasDefinition*
| *TypeDefinition*

The *PackageDeclaration* gives the package the generated Java class belongs to. (The name of the generated class is given by the name of the source file.) The *ConstructorScope* makes the scope of the constructor of the generated class default. The *definition*s gives the specification of the generated class.

Lexical Analyzer

SubtokenDefinition ::=
`$subtoken` *IDENTIFIER* `=` *tokenExpression* `;`

TokenDefinition ::=
`$white`$_{opt}$ `$token` *IDENTIFIER*
(`=` *tokenExpression*)$_{opt}$ `;`

The generated class contains a interface and a default implementaion for lexical analysis. A *TokenDefinition* defines a **terminal**. A terminal **matches** the character sequences described by the *tokenExpression*, or matches no sequence if *tokenExpression* is omitted (used only for user-defined lexical analyzers). The default implementation repeats cutting out, from the character sequence inputed into the implementaion, the longest mached string as a **token**, that is an instance of a terminal. A *STRING* in *expression*s also defines a terminal. It matches the represented string.

If a terminal is `$white`, the instance of it is a **white token**. White tokens are ignorable for parsing and useful for white spaces and comments.

A *SubtokenDefinition* gives a name the *tokenExpression* to be used in *tokenExpression*s.

*tokenExpression*s are defined as

Rank	*tokenExpression*	Matched Strings
6	*expr1* `|` *expr2*	alternative one matches
5	*expr1* `&` *expr2*	both matches
	expr1 `-` *expr2*	former matches, latter not
4	*expr1 expr2 ...*	connection of matches
3	`!` *expr*	not matches
2	*expr* `*`	zero or more repeats
	expr `+`	one or more repeats
	expr `?`	one or zero repeats
1	`[` *expr* `]`	one or zero repeats
	`(` *expr* `)`	the expression matches
	CHAR	the character
	CHAR `..` *CHAR*	a character between
	STRING	the string
	IDENTIFIER	(sub)terminal maches

Syntax Analyzer

AliasDefinition ::= *IDENTIFIER* `=` *expression* `;`

TypeDefinition ::= *modifiers IDENTIFIER supertypes*$_{opt}$
`{` *expression* `}`

modifiers ::= (`$protected` | `$private`)$_{opt}$ `$abstract`$_{opt}$
| `$parsable`
| `$protected-parsable $protected`$_{opt}$

supertypes ::= `->` *TypeName* (`&` *TypeName*)*

TypeName ::= *IDENTIFIER*

InlineExpression ::= *TypeDefinition*

The generated class contains one or more syntax analyzers, which parses the sequence of the tokens given by a lexical analyzer. The grammar is described by an extended EBNF. A *TypeDefinition* or an *AliasDefinition* describes a rule. The *IDENTIFIER* describes the name of the defined **nonterminal**, and the *expression* gives the right-hand side of the rule.

*TypeDefinition*s also appear in *expression*s as *InlineExpression*s for convenience.

The *expression* is defined as

R.	*expression*	Matched Tokens
5	*expr1* `|` *expr2*	alternative one matches
4	*expr1 expr2 ...*	connection of matches, sandwiching zero or more white tokens
3	*expr* `*`	zero or more repeats, sandwiching zero or more white tokens
	expr `+`	one or more repeats, sandwiching zero or more white tokens
	expr `?`	one or zero repeats
	expr `/` *TypeName*	the expression matches (used to control types of labels)
2	*IDENTIFIER* `:` *expr*	the expression matches (labeled expression)
	`$label :` *expr*	the expression matches (labeled expression)
1	`[` *expr* `]`	one or zero repeats
	`(` *expr* `)`	the expression matches
	`$embed (` *expr* `)`	the expression matches (replaces aliases by its *expression*)
	IDENTIFIER	terminal or nonterminal matches
	STRING	the terminal
	InlineExpression	the nonterminal matches

The syntax analyzer builds a **concrete syntax tree** (**CST**) first. A token (an instance of a terminal) is a leaf

of the tree and an instance of a nonterminal is a node of the tree. The nodes has **labeled** children. A **labeled expression** formed as *label*:*expression* means the instances of the terminals and noterminals in the *expression* are labeled by the *label*. We also say the children **have** the *label*. A node can have multiple labels and a label can be had by multiple nodes.

And then, the analyzer builds an **abstract syntax tree** (**AST**) by removing some nodes from the CST. The node that is an instance of the nonterminal defined by *AliasDefinition* is removed. The children of the removed node become the children of the parent node of the removed node (see Example). If there is the child of the removed node having the special label `$label`, the analyzer makes these children release the `$label` and instead have the labels the removed node has. If no children have the `$label`, all the children get the labels the removed node has.

The analyzer outputs the objects of Java representing the AST. A token is an instance of the nested interface `Token`. A node is an instance of the nested interface whose name is the same as the nonterminal an instance of which in the CST the node was. The nested interface has the method whose name is the same as a label, that has no arguments, and that returns the children labeled by the label. If the label labels at most one child, the static type of the result is the most specific common type of the nodes the label may label. If the label may label more than one children, the result is a `java.util.List`, or will be a `java.util.List` parameterized by the most specific common type for Java 1.5.

The *supertypes* specifies the supertype(s) of the nested interface. A *TypeName* should be a name of the nonterminal defined by *TypeDefinition*. If no *supertypes* are specified, the nested interface is a subtype of the nested type `Node`. `Token` is also a subtype of `Node`.

If a nonterminal is `$parsable`, the generated class has public methods with various arguments to parse the grammer whose goal symbol is the nonterminal. If a nonterminal is `$protected-parsable`, the generated class has the methods but they are protected.

If a nonterminal is `$protected` or `$private`, the nested interface whose name is the same as the nontermina is protected or private. Otherwise, it is public.

If a nonterminal is `$abstract`, the nonterminal generates the nested interface but should not appear in syntax trees.

Example

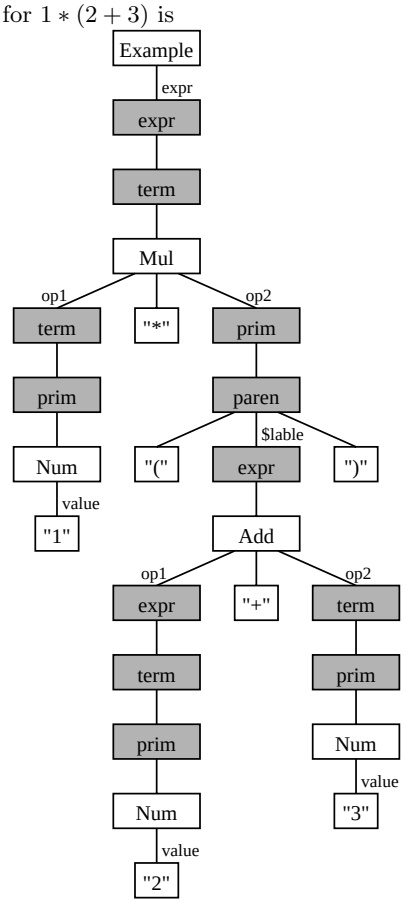
```
$package parser;
$protected $constructor;

$token INTEGER = '0'..'9'+;
$white $token WHITE_SPACES = ( ' ' | '\t' )+ ;

$parsable Example { expr:expr }

$abstract Expr { }
expr = term | Add | Sub ;
Add -> Expr { op1:expr "+" op2:term }
Sub -> Expr { op1:expr "-" op2:term }
term = prim
     | Mul -> Expr { op1:term "*" op2:prim }
     | Div -> Expr { op1:term "/" op2:prim } ;
prim = "(" $label:expr ")" | $label:Num ;
Num -> Expr { value:INTEGER }
```

The output is

```
package parser;
public class Parser {
    Parser() { ... }
    public interface LexicalAnalyzer { ... }
    protected LexicalAnalyzer
            createLexicalAnalyzer(...) { ... }
    public interface Node {
        List getChildNodes(); ...
    }
    public interface Token extends Node {
        String getImage();
        int getLine(); int getColumn();
        ...
    }
    public Example parseExample(File file) { ... }
    public Example parseExample(LexicalAnalyzer la) {
        ...
    }
    ...
    public interface Example extends Node {
        Expr expr();
    }
    public interface Expr extends Node { }
    public interface Add extends Expr {
        Expr op1(); Expr op2();
    }
    ...
}
```

(The details are described in the javadoc comment of the generated class.)

The CST for $1 * (2 + 3)$ is



The AST generated by removing the grayed boxes is