Handbook of ¬<><∪∪ in English

Tetsuhiko Koto

for Version 0.72

`http://ne.cs.uec.ac.jp/~koto/notavacc/`

`koto@takopen.cs.uec.ac.jp`

Overall Structure

A input for the ¬<><∪∪ compiler compiler is converted into a sequence of tokens by the process same as Java (JLS 3.2), but the noterminal symbols of ¬<><∪∪ are defined by the following table.

Noterminal	Description
IDENTIFIER	not starting with `$`
keywords	starting with `$`
separators/operators	the strings used in the following EBNF
CHAR	Java character literal
STRING	Java string literal

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

Meta-notation	Description
Italic or *italic*	nonterminals
ITALIC	terminals (symbol)
`Underlined`	terminals (literal)
expr1 | *expr2*	alternative
expr*	Kleene star
expr$_{opt}$	optional

The sequence of tokens should be structured by the following grammar. 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*

¬<><∪∪ outputs a single Java source file and it defines a top level class that contains many nested types. The name of the top level class is given by the name of the ¬<><∪∪ source file, and the package of the class is given by the *PackageDeclaration*. The *ConstructorScope* makes the scope of the constructor of the generated top level class the default scope (not the protected scope). The *definition*s gives the details of the generated top level class.

Lexical Analyzer

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

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

The generated top level class contains a interface and a default implementaion for lexical analysis. A *TokenDefinition* defines a **terminal**. A terminal **matches** the character sequences that **matches** the *tokenExpression*, or matches no sequence if *tokenExpression* is omited (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**, which is an instance of a terminal. A *STRING* in *expression*s of non-`$abstract` syntax definitions 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 is defined by the following table.

Priority	*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 match
2	*expr* `*`	zero or more repeats
	expr `+`	one or more repeats
	expr `?`	one or zero repeats
1	`[` *expr* `]`	one or zero repeats
	`(` *expr* `)`	*expr* 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 top level class contains zero 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 production. The *IDENTIFIER* describes the name of the defined **nonterminal**, and the *expression* gives the right-hand side of the production.

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

The *expression* is defined by the following table.

P.	*expression*	Matched Token Sequence
5	*expr1* `|` *expr2*	alternative one matches
4	*expr1* *expr2* ...	connection of matches
3	*expr* `*`	zero or more repeats
	expr `+`	one or more repeats
	expr `?`	one or zero repeats
	expr `/` *TypeName*	*expr* matches (used to control types of labels)
2	*IDENTIFIER* `:` *expr*	*expr* matches (labeled expression)
	`$label` `:` *expr*	*expr* matches (labeled expression)
1	`[` *expr* `]`	one or zero repeats
	`(` *expr* `)`	*expr* matches
	`$embed` `(` *expr* `)`	*expr* 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 (See Example). 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*. An instance can be labeled by multiple labels and a label can label multiple instances. The same label can also appear twice or more in an *expression* lexically.

And then, the analyzer builds an **abstract syntax tree** (**AST**) by removing some nodes from the CST. All the node that is an instance of the nonterminal defined by *AliasDefinition* (**Alias**) is removed. The children of a removed node become the children of the parent node of the removed node. If there is the child that is labeled by `$label`, the analyzer replaces the `$label` with the labels that label the removed node. If no children labeled by `$label`, all the children become labeled by the labels the removed node has.

The analyzer outputs the objects representing the AST. A token is represented by an instance of the nested interface `Token`. A node is represented by 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 children 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 interface `Node`. `Token` is also a subtype of `Node`.

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

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

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

Example

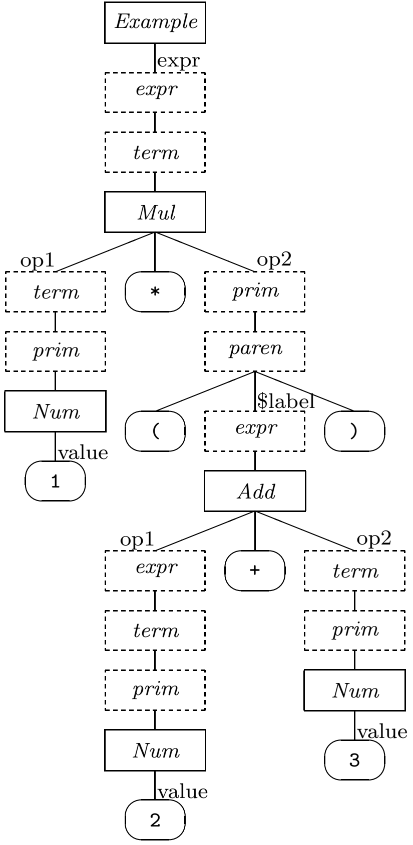
```
$package parser;
$protected $constructor;

$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 from the above source is the following.

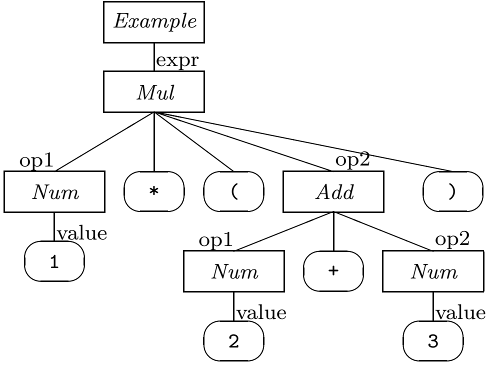
```
package parser;
public class Parser {
    Parser() { ... }
    public interface LexicalAnalyzer { ... }
    protected LexicalAnalyzer
                createLexicalAnalyzer(...) { ... }
    public interface Node {
        List getChildNodes(); ...
    }
    public interface Token extends Node {
        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 file.

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



The AST generated by removing the broken line boxes is


Example
expr
Mul
op1
op2
Num
*
(
Add
)
value
1
op1
op2
Num
+
Num
value
2
value
3