Handbook of ¬<><∪∪ in Japanese

Tetsuhiko Koto
for Version 1.0.0 http://ne.cs.uec.ac.jp/~koto/notavacc/ koto@takopen.cs.uec.ac.jp

Overall Structure

¬<><∪∪ コンパイラ・コンパイラへの入力は、Java と同じプロセス (JLS 3.2) でトークンの列に変換される。ただし、終端記号は次のようになる。

終端記号	説明
IDENTIFIER	`$`で始まる
予約語	`$`で始まらない
セパレータ・オペレータ	以下の EBNF で使われる文字列
	および `||`, `&&`, `--`, `!!`, `**`, `++`, `??`
CHAR	Java の文字リテラル
STRING	Java の文字列リテラル

このドキュメントは、¬<><∪∪ への入力の文法を、次の記法を用いた EBNF で示す。

メタ表現	説明
Italic or *italic*	非終端記号
ITALIC	終端記号 (シンボル)
`Underlined`	終端記号 (即値)
expr1 | *expr2*	選択
$expr^*$	Kleene star
$expr_{opt}$	省略可能

トークンの列は、次の文法に従っていなければならない。ゴール記号（開始記号）は、*Root* である。

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

ParserDeclaration ::= `$protected`$_{opt}$ `$parser` *JavaName* `;`
JavaName ::= *IDENTIFIER* (`.` *IDENTIFIER*)*
ConstructorScope ::= `$protected` `$constructor` `;`
definition ::= *SubtokenDefinition*
| *TokenDefinition*
| *AliasDefinition*
| *TypeDefinition*

¬<><∪∪ は Java のソースファイルを 1 つ出力し、それは多数のネストしたタイプを含むトップレベル・クラスを 1 つ定義する。トップレベル・クラスの名前は、*ParserDeclaration* によって決められる。*ParserDeclaration* に `$protected` が指定されていれば、出力されるクラスはデフォルトスコープ (パッケージスコープ) になり、そうでなければ public になる。*ConstructorScope* は、出力されるトップレベル・クラスのコンストラクタを、デフォルトスコープ (パッケージスコープ) にする。*definition* は、出力されるトップレベル・クラスの詳細を定義する。

Lexical Analyzer

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

出力されるトップレベル・クラスは、字句解析のためのインターフェイスと、デフォルトの実装を含む。デフォルトの実装は、入力された文字列の先頭から、**終端記号**のどれかが最長一致で**マッチする**文字列を**トークン** (**終端記号のインスタンス**とも呼ぶ) として切り出すことを繰り返す。

1 つの *TokenDefinition* は、1 つの終端記号を定義する。終端記号は、*tokenExpression* が**マッチする**文字の列にマッチする。ただし、*tokenExpression* が省略されていた場合、その *TokenDefinition* によって定義される終端記号は、どの列にもマッチしない (ユーザ定義の字句解析器のためだけに使われる)。`$abstract` でない *TypeDefinition*/*AliasDefinition* の *expression* 中の *STRING* も終端記号を定義する。この終端記号は、文字列リテラルが表す文字列にマッチする。

もし、終端記号が `$white` として定義されていれば、そのインスタンスを**ホワイト・トークン**と呼ぶ。ホワイト・トークンは、構文解析に際して無視されるので、ホワイトスペースやコメントを扱うのに有用である。

SubtokenDefinition は、*tokenExpression* に対して名前を与える。この名前は、*tokenExpression* の中で使うことができる。*tokenExpression*s は次の表で与えられる。

P.	*tokenExpression*	**マッチ**する文字列
6	*expr1* `|` *expr2*	どちらかがマッチする文字列
5	*expr1* `&` *expr2*	両方がマッチする文字列
	expr1 `-` *expr2*	前者がマッチし、後者がしない文字列
4	*expr1 expr2 ...*	マッチした文字列をつなげた文字列
3	`!` *expr*	マッチしない文字列
2	*expr* `*`	0 回以上の繰り返し
	expr `+`	1 回以上の繰り返し
	expr `?`	0 か 1 回の繰り返し
1	`[` *expr* `]`	0 か 1 回の繰り返し
	`(` *expr* `)`	*expr* がマッチする文字列
	CHAR	*CHAR* が表す文字
	CHAR `..` *CHAR*	その範囲にある文字
	STRING	*STRING* が表す文字列
	IDENTIFIER	（サブ）終端記号がマッチする文字列

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*

出力されるトップレベル・クラスは、字句解析器が出力するトークンの列を構文解析する構文解析器を含む。文法は、拡張された EBNF で表現される。1 つの *TypeDefinition* か 1 つの *AliasDefinition* が、1 つのプロダクションを定義する。*IDENTIFIER* は、定義される非終端記号の名前を表し、*expression* が、プロダクションの右辺をあらわす。*TypeDefinition* は、*expression* の中で、*InlineExpression* としても現れる。

expression は次のように定義される。

P.	*expression*	**マッチ**するトークン列
5	*expr1* `|` *expr2*	いずれかがマッチする列
4	*expr1 expr2 ...*	マッチする列をつなげたもの
3	*expr* `*`	0 回以上の繰り返し
	expr `+`	1 回以上の繰り返し
	expr `?`	0 か 1 回の繰り返し
	expr `/` *TypeName*	*expr* がマッチする列 (ラベルの型を制御するのに使われる)
2	*IDENTIFIER* `:` *expr*	*expr* がマッチする列 (labeled expression)
	`$label` `:` *expr*	*expr* がマッチする列 (labeled expression)
1	`[` *expr* `]`	0 か 1 回の繰り返し
	`(` *expr* `)`	*expr* がマッチする列
	`$embed` `(` *expr* `)`	*expr* がマッチする列 (エイリアスをその式で置き換える)
	IDENTIFIER	終端・非終端記号がマッチする列
	STRING	その終端記号
	InlineExpression	その非終端記号がマッチする文字列

構文解析器は、最初に**具象構文木** (CST; concrete syntax tree) を構築する（Example 参照）。トークン (終端記号のインスタンス) が構文木の葉になり、**非終端記号のインスタンス**が構文木のノードを作る。ノードは、**ラベル付けされた**子を持つ。*label*`:` *expression* の形式の **labeled expression** は、*expression* に含まれる終端・非終端記号のインスタンスが、*label* によってラベル付けされることを表す。1 つのインスタンスが複数のラベルによってラベル付けされても構わないし、1 つのラベルが

複数のインスタンスをラベル付けしても良い。*expression* の中に、同じラベルが字句的に複数回存在しても良い。

その後、構文解析器は、具象構文木からノードを取り除くことで、**抽象構文木** (AST; abstract syntax tree) を構築する。*AliasDefinition* で定義される非終端記号（**エイリアス**）のインスタンスであるノードが、全て取り除かれる。取り除かれたノードの子は、取り除かれたノードの親の子になる。取り除かれたノードに `$label` によってラベル付けされた子が存在した場合、`$label` を取り除かれたノードをラベル付けしているラベルで置き換える。取り除かれたノードのどの子も `$label` によってラベル付けされていない場合、全ての子が取り除かれたノードをラベル付けしているラベルによってラベル付けされる。

抽象構文木は Java のオブジェクトによる木構造で表される。トークンは、入れ子のインターフェイス `Token` のインスタンスで表される。葉以外のノードは、そのノードが具象構文木においてインスタンスであった非終端記号と同じ名前を持つネストしたインターフェイス のインスタンスで表される。そのインターフェイスは、ラベルと同名で、引数の無いメソッドを持つ。メソッドの戻り値は、そのラベルによってラベル付けされている子（高々 1 つの子しかラベル付けしない場合）、あるいは子のリスト（複数の子をラベル付けする可能性がある場合）である。戻り値の静的型は、前者の場合ラベル付けされる可能性のある子に共通の型で、**公平に最も限定的な**ものになる。後者の場合、最も限定的な共通の型の配列か `java.util.List` になる。

supertypes は、ネストしたインターフェイスの継承関係を設定する。*TypeName* は、*TypeDefinition* によって定義された非終端記号の名前で無ければならない。もし、*supertypes* が無ければ、そのネストしたインターフェイスは、`Node` のサブインターフェイスになる。`Token` もまた、`Node` のサブタイプである。

もし、非終端記号が `$parsable` と定義されていれば、出力されるトップレベル・クラスは、その非終端記号をゴール記号（開始記号）とする文法を構文解析する、様々な引数を伴った public なメソッドを持つ。非終端記号が `$protected-parsable` と定義されていれば、同様だが protected なメソッドが出力される。

もし、非終端記号が `$protected` か `$private` と定義されていれば、その非終端記号と同名のネストしたインターフェイスは、protected か private になる。そうでない場合、そのネストしたインターフェイスは public になる。

もし非終端記号が `$abstract` と定義されていれば、その非終端記号はネストしたインターフェイスを生成するが、`$abstract` でない *TypeDefinition*/*AliasDefinition* の *expression* 中に現れることはできない。

Example

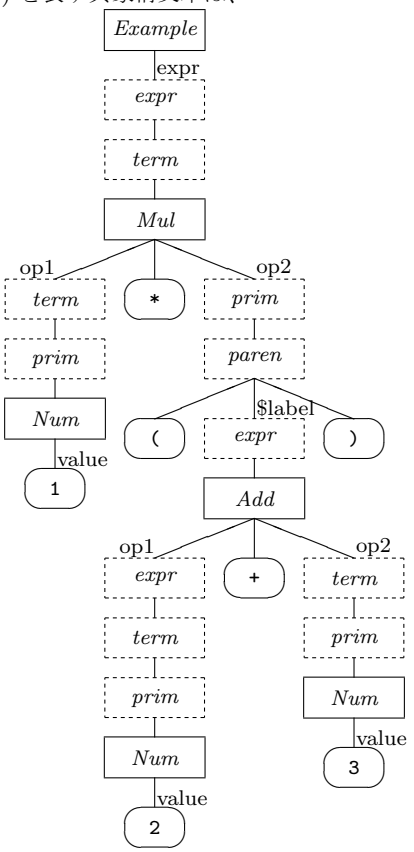
```
$parser parser.Parser;
$protected $constructor;

$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 }
```

このソースから次のような出力が得られる。

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

詳細は、出力されたプログラムの javadoc コメントへ記述されている。

$1 * (2 + 3)$ を表す具象構文木は、



点線で囲まれたノードを取り除いて作られた抽象構文木は、