Object–Oriented Tokens for the Parser Generator jay

Holger Haas & Martin Plümicke

University of Cooperative Education Stuttgart
Department of Information Technology
Florianstraße 15
D–72160 Horb
tel. +49-7451-521142
fax. +49-7451-521190
m.pluemicke@ba-horb.de

Abstract. jay is a parser generator for JAVA derived from ATTs' parser generator **yacc**, which was designed for the C programming language. Being based on **yacc**, jay is not well adapted for object-orientation, because this principle is not known in the C-world. To correct this flaw jay was redesigned to cope with objects.

1 The Parser Generator jay

The parser generator jay [1] is a tool which generates for a given grammar in BNF a respective parser in JAVA. The specification of jay is very similar to that of yacc [2]. This means that it is very easy to learn jay for a yacc user. But there are some significant differences, which arises from the fact that JAVA is object–oriented and the original yacc bases on C:
The generated JAVA program is parameterized by the JAVA interface `yyInput`:

```
boolean advance () throws java.io.IOException;
int token ();
Object value ();
```

The method `advance()` reads the next lexem and gives false if end-of-file is reached. The method `token()` gives the corresponding token. Finally, the method `value()` is used to get the corresponding object for the jay's value–stack. This interface must be implemented by the user, himself.
For the implementation of `advance()` and `token()` a scanner is needed. The simplest possibility is to implement a subclass of the JAVA's stringtokenizer. More comfortable is for example to use JLex [3]. JLex is a generator for a lexical analyzer. It bases on the well-known lex [2] but produces JAVA code and is written in JAVA. The design approach of JLex is object–oriented, which means that the produced tokens can be real objects (instances of classes, not only `ints`). In contrast jay accepts, as yacc, only int values as tokens.

2 The New Design

The original design of jay has the disadvantage, that attributes of Tokens, which are recognized by the scanner (e.g. the name of the identifier in the token identifier), must be handled by the user, himself. These attributes must be treated by the implementation of the interface `yyInput`. Moreover, the whole handling of the interface `yyInput` is inconvenient and can be automated.

Our approach changes the type of the tokens in jay from the base type `int` to a subtype of a new type `yyTokenclass`. This means that for each declaration `a subclass tok_name of yyTokenclass` is generated. The class `yyTokenclass` contains a field `value` of the type `Object` and a constructor to assign values to this field. This means that a typical statement in the JLex specification is `return tok_name(attribute)`, where `attribute` contains the token's attribute, as e.g. the name of the scanned identifier.

An overview of the generated class structure is presented in figure 1. The JLex specification `browserlexer` generates the class `browserlexer`, which contains the scanner method. The directive `%class` defines the name of the generated class. The other classes are generated by the new jay. The new directive `%jayscanner` produce a class, which represents the scanner as a subclass of the class generated by the JLex specification. The additional new directive `%jayparser` produces the

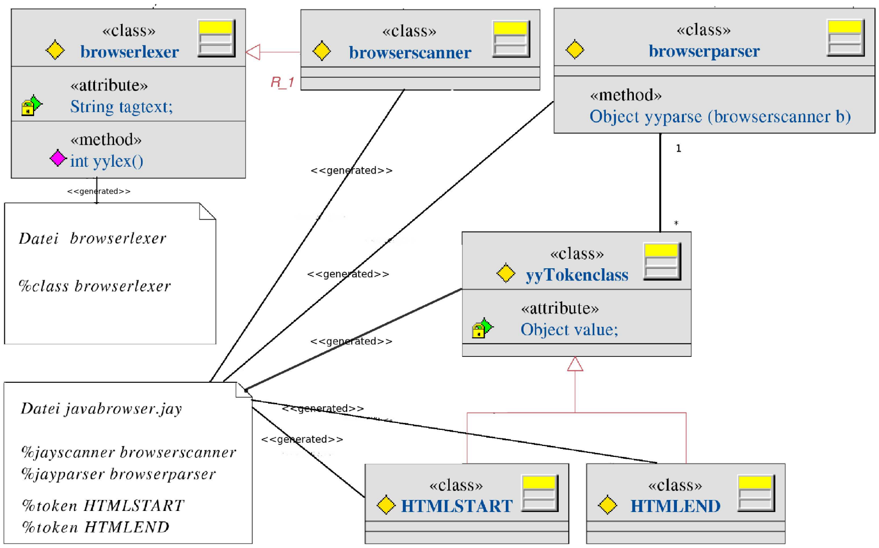

Fig. 1. The generated class structure of new jay and new JLex

parser class with the method yyparse(). Finally, for each declared token a class is generated as a subclass of the also generated class yyTokenclass.

3 The Implementation

With the new design, two goals were to be achieved:

1. The tokens should be handed over from the scanner to the parser as objects
2. The user should not longer need to implement the scanner-interface himself

To make this possible, the source code of jay and JLex had to be modified in several ways:

skeleton jay uses a file named "'skeleton"' as template for the interpreter code. It contains code fragments which determine among other things the token types, token handling and the scanner-interface simply by copying the corresponding parts to the generated parser code in the progress of the parser generation.
In the modified skeleton the scanner-interface does not longer contain the methods int token () and Object value (). With the use of object tokens, the whole parsing can be handled with the method yyTokenclass advance () alone. If end-of-file is reached, advance () returns NULL.
The superclass yyTokenclass, from which the token objects inherit their attributes, is also defined in the new skeleton.
The generated cellar automaton works internally still with integer values. The interpreter code had therefore to be adjusted to allow the utilisation of token objects.

jay The source code of jay had to be modified in a way that it does not longer create integer constants as tokens, but token objects which contain the corresponding integer constant as attribute. This attribute can then be processed by the cellar automaton as supplied before.

JLex In order that the user does not longer have to implement the scanner-interface himself, the source code of JLex had also to be adjusted. The modifications affect mainly the ability to process the new directives %jayparser and %jayscanner. If these directives are found, the new JLex implements the scanner interface. If not, JLex behaves as the original version.

4 Summary and Outlook

With the mentioned modifications jay and JLex can be used and understood more easy and, which is more important, fit into an object–orientated concept. However, the concept of encapsulation was not slightly neglected. For further development, improvements of encapsulation should be a challenge.

References

1. Kühl, B., Schreiner, A.T.: jay – Compiler bauen mit Yacc und Java. iX (1999) (in german).
2. von Levine, J.R., Mason, T., Brown, D.: Lex & yacc, UNIX Programming Tools. A Nutshell-Handbook. 2nd edn. O'Reilly Associates (1992)
3. Berk, E.: JLex: A lexical analyzer generator for Java(TM). 1.2 edn. (1997)