oops reference manual

Version 1.1

Jan Kraneis, jkraneis@informatik.uni-osnabrueck.de
Bernd Kühl, Bernd.Kuehl@informatik.uni-osnabrueck.de
Axel-Tobias Schreiner, axel@informatik.uni-osnabrueck.de

1
Input format

oops input is free format. Whitespace must be supplied to separate words. Comments follow Java conventions and count as whitespace: A comment extends from `//` to the end of the input line or from `/*` to `*/`. Comments may not be nested.

Identifiers name rules and input categories (tokens). An identifier must start with a letter and consist of letters and digits; an underscore or a period can be used in place of a letter. There are no reserved words.

Literals are quoted with `"` and follow C conventions: The usual escape sequences such as `"\n"` and `"\t"` are recognized and converted to single characters.

2
Grammar specification

The following grammar is legal input to *oops* and describes it's own syntax. A grammar consists of one or more rules:

```
parser: { rule };
```

A rule has a name and a grammar expression and is terminated by a semicolon:

```
rule    : ID ":" alt ";";
```

Alternatives are separated with `|`:

```
alt     : seq [{ "|" seq }];
```

Each alternative consists of a sequence of one or more items:

```
seq     : { item };
```

An item can be an identifier naming a rule or a category of input symbols, a quoted literal, or a grammar expression enclosed in various parentheses for precedence or repetition.

```
item    : TOKEN | LIT | ID
        | "(" alt ")"
        | "{" alt "}"    // one or more
        | "[" alt "]";   // zero or one (optional)
```

Regular parentheses `()` are for precedence grouping.

Curly braces `{}` indicate zero or more and brackets [] indicate zero or one (optional) occurrences. As a special case, nesting braces and brackets `[{}]` or `[{}]` denotes zero or more occurrences.

The name of the first rule is termed the start symbol of the grammar. By convention we write tokens in upper case.

As described in the paper[2], *oops* verifies that a grammar is LL(1) but accepts shift/reduce conflicts as described in the tutorial chapter "Conflicts": Each rules must be computable on the basis of the start rule. Infinite recursions are not permitted, the lookaheads of alternatives must be different in pairs, and shift/reduce conflicts in the grammar are detected. A shift/reduce conflict is reported and treated as a shift. Other errors cause *oops* to abort following the grammar check.

3
Scanner interface

Provisional: we are currently discuss to simplify this interface. So the interface will change in the next release.

A generated parser operates on a sequence of terminal symbols, which are supplied by a scanner as a coroutine of the parser. A scanner must implement the Interface `oops.parser.Scanner`:

oops.parser.Scanner

```
package oops.parser;
import java.io.IOException;
import java.io.Reader;

/** describes what a scanner for an oops-generated parser must do. */
public interface Scanner {
  /** initialize, read one symbol ahead.
      @param parser is used to screen symbols.
    */
  void scan (Reader in, Parser parser) throws IOException;
  /** move on to next token.
      @return false if atEnd() becomes true.
    */
  boolean advance () throws IOException;
  /** @return true if positioned beyond tokens. */
  boolean atEnd ();
  /** @return single-element lookahead set, null for unidentifiable token. */
  Set tokenSet ();
  /** @return node corresponding to token. */
  Object node ();
}
```

`scan()` initializes the scanner and has to advance to the first symbol (literal or token) in the input. The scanner collects characters from `in` and produces symbols. In `parser` an object of the type `oops.parser.Parser` is provided.

`advance()` should detect the next symbol in the input. This method returns `false` at the end of the input, else `true`.

`atEnd()` returns `true` if the end of the input was reached by `scan()` or `advance()`, else `false`.

`tokenSet()` returns the detected symbol as `Set` object. `Set` objects are needed to identify the detected symbols. For each possible symbol the parser manages a `Set` object which must be returned by `tokenSet()` for the identification of detected symbols.

`node()` returns an arbitrary value which the scanner wishes to associate with the current symbol.

`oops.parser.Parser` provides methods to retrieve a `Set` from a `String`:

`getLitSet(String)` returns the `Set` representing a Literal quoted in the grammar; e.g., if the grammar contained

```
sum : product {[  "+" product ]} ;
```

then

```
parser.getLitSet("+")
```

returns the `Set` which the parser expects in the input for the Literal `"+"`.

`getPeer(String).getLookahead()` returns the `Set` representing a category of inputs named in the grammar; e.g., if the grammar contained

```
term : NUMBER ;
```

then

```
parser.getPeer("NUMBER").getLookahead()
```

returns the `Set` which the parser expects in the input for a `NUMBER`..

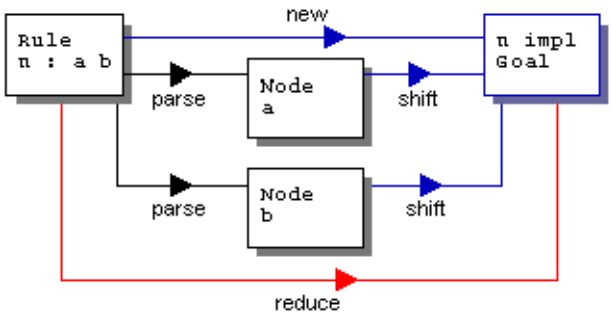
Scanner implementation

To implement a scanner we used *oolex*, *JLex* or a `java.io.StreamTokenizer`; see the chapter "References".

In the examples the package `scanners` contains a scanner implemented with *oolex*, *JLex* and `java.io.StreamTokenizer`.

4
Goal interface

A new `Goal` object is instantiated whenever a new rule is invoked while parsing a program.



The `Goal` object is sent certain messages as recognition proceeds:

oops.parser.Goal

```
package oops.parser;

/** describes what each nonterminal must be able to do during parsing. */
public interface Goal {
  /** presents result of reduction.
  @param sender just received reduce().
  @param value was created by sender.
  */
  void shift (Goal sender, Object value);
  /** presents result of scanning.
  @param sender just matched input.
  @param value was created by sender.
  */
  void shift (Token sender, Object value);
  /** presents result of scanning.
    @param sender just matched input.
    @param value was created by sender.
    */
  void shift (Lit sender, Object value);
  /** concludes rule recognition.
    @return generated (sub-)tree.
    */
  Object reduce ();
}
```

The method `shift(Lit sender, Object value)` is called if quoted text (a literal) is recognized in the rule. `sender` can be asked for the recognized text with `getBody()` and `value` is the associated value object.

`shift(Token sender, Object value)` is called if a token was recognized. `value` is the corresponding value object.

`reduce()` is called in the `Goal`, if the rule was completely recognized. `reduce()` must return an `Object` as the value of the rule.

The method `shift(Goal sender, Object value)` is called when an `ID` referencing a rule is on the right hand side of a rule and when the rule for the `ID` was completly recognized. In this case, `reduce()` was sent to the `ID` rule's `Goal` and `value` is the result returned by `reduce()`.

When a rule needs to be matched, the *oops* parser creates an object implementing the `Goal` interface. By default *oops* tries to create an instance from a class named like the rule. If no suitable class is found, a `GoalAdapter` is used. A `GoalAdapter` remembers the first none `null` value object received through a `shift()` message and returns it for `reduce()`.

See "GoalMaker, GoalMakerFactory and default factories" how to set a package name for the Goal classes or how to change the default behavior.

5
GoalMaker, GoalMakerFactory and default factories

GoalMakerFactory, GoalMaker

Before the parsing starts `oops.Compile` asks the parser to set an `oops.parser.GoalMaker` for every rule. The parser asks an object implementing `oops.parser.GoalMakerFactory` to produce the `GoalMaker`, which is stored in the `Rule`.

oops.parser.GoalMakerFactory

```
package oops.parser;

/** A GoalMakerFactory is called to produce a GoalMaker for a rule. */
public interface GoalMakerFactory {
  /** @return A GoalMaker instance */
  public GoalMaker goalMaker (String ruleName);
}
```

Whenever the parsing of a `Rule` starts, the rule asks the stored `GoalMaker` to produce a `Goal` for the `shift()` and `reduce()` messages:

oops.parser.GoalMaker

```
package oops.parser;

/** Every Rule knows one GoalMaker. When the parsing of a  Rule
  * starts, the Rule asks the GoalMaker for a new Goal.
  */
public interface GoalMaker {
  /** @return A Goal. */
  public Goal goal ();
}
```

`GoalMakerFactory` and `GoalMaker` permit arbitrary associations between rule names and `Goal` classes. With the option `-f` to `oops.Compile` a user can choose the desired factory; e.g. a factory produces `GoalMaker`s, which load `Goal`s from a network connection.

DefaultGoalMakerFactory, DebuggerGoalMakerFactory

Normally an *oops* user doesn't want to write his own `GoalMakerFactory`. Therefore the *oops* system provides two implementations: `oops.parser.DefaultGoalMakerFactory` and `oops.parser.DebuggerGoalMakerFactory`.

By default `oops.Compile` uses a `DefaultGoalMakerFactory`. This factory produces `GoalMaker`s which try to find the `Goal` class by searching the classpath for a class with a name matching the rule

name. If a suitable class is found, instances are created by calling the constructor without arguments. If no suitable class is found a `GoalAdapter` is used.

Factory classes should be configured with system properties, so that different factories may coexist. The property name should involve the factory class name.

For `DefaultGoalMakerFactory` the following properties may be configured to change the action of the produced `GoalMaker` instances:

`oops.parser.DefaultGoalMakerFactory.prefix`
: Sets a package name which is prefixed to a rule name when the `Goal` class name is searched for. By default the rule name is the class name — the rule name can contain periods.

`oops.parser.DefaultGoalMakerFactory.verbose`
: If `true`, generated `GoalMaker`s print a short message, when no suitable `Goal` class for a rule name can be found and a `GoalAdapter` will be used. The default is `true`.

A `DebuggerGoalMakerFactory` acts like a `DefaultGoalMakerFactory`, except that instances of `GoalDebugger` are used as default `Goal`s. Also similar two properties are supported:

`oops.parser.DebuggerGoalMakerFactory.prefix`
: Sets a package name which is prefixed to a rule name when the `Goal` class name is searched for. By default the rule name is the class name — the rule name can contain periods.

`oops.parser.DebuggerGoalMakerFactory.verbose`
: If `true`, generated `GoalMaker`s print a short message, when no suitable `Goal` class for a rule name can be found and a `GoalDebugger` will be used. The default is `false`.

With these two classes, a user of *oops* does not usually write his own `GoalMakerFactory` and `GoalMaker`s.

6
Tools

Parser generation — oops.boot.Oops

```
java oops.boot.Oops [ option] [ <] grammar.ebnf [ > grammar.ser]
```

Option:

`-v`	Prints the *oops* version.

`oops.boot.Oops` accepts a grammar from a file or from the standard input. The serialized output is written to standard output.

Parser execution — oops.Compile

```
java oops.Compile [ options] ... grammar.ser scanner [ <] source [ > tree.ser]
```

Options:

`-d`	An `oops.parser.DebuggerGoalMakerFactory` is the `GoalMakerFactory` and some information about the parser is dumped.
`-f factory`	Specifies a class name for the `GoalMakerFactory`. Default is `oops.parser.DefaultGoalMakerFactory`.

The user provides a file `grammar.ser` containing a serialized parser and a class name `scanner` for a class implementing the `oops.parser.Scanner` interface.

The command `oops.Compile` accepts input from a file source or from standard input. If the `Goal` of the start rule returns non-`null` and the returned object is serializable, `oops.Compile` serializes this object to standard output.

Parse tree dump — oops.tools.Dump

```
java oops.tools.Dump [ parser]
```

`oops.tools.Dump` reads from standard input or from a file a serialized parser generated by *oops* and dumps a text representation of the parse tree:

```
$ CLASSPATH=.:../.. java oops.tools.Dump aParser.ser
oops.parser.Parser
   oops.parser.Rule identifier : ( letter [{  ( letter | number ) }] ) .
      oops.parser.Seq
         oops.parser.Id letter
         oops.parser.Many
            oops.parser.Alt
               oops.parser.Id letter
               oops.parser.Id number
```

7
Running a parser from Java

The class method `main()` in `oops.Compile` is a normal method. To run a parser from Java just set the necessary system properties and call the `main()` method:

```
// set system properties
java.util.Properties p = System.getProperties();
p.put("oops.parser.DefaultGoalMakerFactory.prefix", "arith");
System.setProperties(p);

// run the parser
oops.Compile.main(new String[] {
  "pathToSerializedParser",  // the serialized parser
  "nameOfScannerClass",      // the scanner
  "pathToInputFile"          // optional an input file
});
```

8 References

[1] Homepage *oops* and tutorial examples: http://www.informatik.uni-osnabrueck.de/bernd/oops/

[2] Axel-Tobias Schreiner, Bernd Kühl, An object-oriented LL(1) parser generator, SIGPLAN Notices, ACM December 2000.

[3] Axel-Tobias Schreiner, An Object-Oriented Parser Generator, Rochester Institute of Technology, 19. October 2000, http://www.inf.uos.de/axel/talks/oops

[4] Documentation `java.io.StreamTokenizer`: http://java.sun.com/j2se/1.3/docs/api/java/io/StreamTokenizer.html

[5] Homepage *oolex*: http://www.informatik.uni-osnabrueck.de/bernd/oolex/

[6] Homepage *JLex*: http://www.cs.princeton.edu/~appel/modern/java/JLex/

[7] German lecture notes on compiler construction: http://www.vorlesungen.uos.de/informatik/compilerbau98/index.html

[8] Bernd Kühl, Axel-Tobias Schreiner, JLex: ein Scanner-Generator für Java, iX, 3/00, Heise Verlag

[9] The examples: http://www.inf.uos.de/bernd/oops/staff/tutorial_examples.zip