oops tutorial

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 Introduction

About oops?

oops is an object-oriented parser generator implemented for Java version 1.1 or later. *oops* was developed in a course on compiler construction at the University of Osnabrück in Germany to show the parts of a parser generator and the parts of a compiler. Since then *oops* has been developed further as part of Bernd Kühl's Ph.D. thesis.

About this tutorial

The main part of the tutorial contains step by step instructions to build an interpreter and a tree builder for arithmetic expressions. Additionally, there is a short example to indicate how *oops* deals with the dangling else problem.

The "oops reference manual" is the definitive description of *oops*; where necessary, the tutorial references the manual for details. There is a separate paper on the design of *oops*, published in SIGPLAN Notices (12/2000).

In the examples we use a UNIX-system. Users of other platforms have to adapt the `CLASSPATH` of the examples.

Versions

Because *oops* is still being developed, there will be changes in the tutorial and in *oops* itself. Whenever there are changes, they will be announced on the *oops* homepage.

2
Arithmetic expressions

Consider a typical arithmetic expression with the usual operations, precedence and parentheses:

/tutorial_examples/recog/input.zb

```
2+3*(4+5);
```

This type of expression is described by the following grammar, expressed in an extended version of Backus-Naur-Form:

/tutorial_examples/recog/arith.ebnf

```
// arithmetic expressions

expression      : [{ [  sum ]  ";" }];
sum             : product [{  "+" product | "-" product }];
product         : term [{  "*" term | "/" term }];
term            : NUMBER | "(" sum ")";
```

Look at the reference manual sections “Input format” and "Grammar specification"for conventions for a suitable grammar: parentheses are used for grouping, curly braces indicate one or more occurrences, square brackets indicate zero or one occurrence. As a consequence, nested braces and brackets together indicate zero or more occurrences.

Grammar check

A file `arith.ebnf` containing this grammar is legal input for *oops*:

```
$ # working directory and classpath
$ pwd; export CLASSPATH=../../jars/oops.jar
/tutorial_examples/recog

$ # check grammar
$ java oops.boot.Oops arith.ebnf >/dev/null
This is oops version 1.0
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using oops generated by oops on Mon Jan 08 15:28:38 GMT+01:00 2001
```

If the grammar is not suitable for recursive descent parsing (i.e. not LL(1)), *oops* would produce error messages. Therefore, *oops* can be used to check if a grammar is LL(1).

Even if a grammar is not (quite) LL(1), it could be used with *oops*. See also the chapter "Conflicts"

Scanner

In the execution example above, standard output has been suppressed. If the grammar is LL(1), *oops* writes a serialized parser to standard output. Once the parser is executed, it obtains input symbols such as numbers, parentheses, and operators from a scanner and checks, if the sequence of symbols agrees with the grammar.

Therefore, before a parser can be executed, a scanner must be written which conforms to an interface prescribed by the *oops* system. See the reference manual section “Scanner interface” for the interface.

In the trivial case of the arithmetic expressions introduced above, a `StreamTokenizer` can be adapted to conform to this interface:

/tutorial_examples/recog/Input.java

```
package recog;

import oops.parser.Scanner;
import oops.parser.Parser;
import oops.parser.Set;
import oops.parser.Token;
import java.io.*;

/** lexical analyzer for arithmetic example. */
public class Input implements Scanner {
  protected StreamTokenizer st;
  protected Parser parser;
  protected Set NUMBER, NL;

  /** called once to initialize scanner. */
  public void scan (Reader r, Parser parser) throws IOException {
    st = new StreamTokenizer(new BufferedReader(r));
    st.ordinaryChar('/'); st.ordinaryChar('-'); // would start comment or number
    st.commentChar('#');                // comments from # to end-of-line

    this.parser = parser;
    NUMBER = ((Token)parser.getPeer("NUMBER")).getLookahead();
    NL = parser.getLitSet("\n");

    advance();                          // one token lookahead
  }

  /** called to detect end of input. */
  public boolean atEnd () { return st.ttype == st.TT_EOF; }

  protected Set tokenSet;               // null or lookahead identifying token
  protected Object node;                // node representing token

  /** called to move forward in input; returns false at end of input. */
  public boolean advance () throws IOException {
    if (atEnd()) return false;
    node = null;
    switch (st.nextToken()) {
      case st.TT_EOF:    tokenSet = Set.getEOFSet(); return false;
      case st.TT_EOL:    tokenSet = NL; break;
      case st.TT_NUMBER: node = new Double(st.nval); tokenSet = NUMBER; break;
      case st.TT_WORD:   tokenSet = null; break;
      default:           tokenSet = parser.getLitSet(""+(char)st.ttype); break;
    }
    return true;
  }

  /** returns current input symbol. */
  public Set tokenSet () { return tokenSet; }

  /** returns associated value for current input symbol. */
  public Object node () { return node; }
}
```

Recognition

Given a grammar and a scanner, the two can be combined and executed:

```
$ # working directory and classpath
$ pwd; export CLASSPATH= ../../jars/oops.jar:..
/tutorial_examples/recog

$ # serialize the parser
$ java oops.boot.Oops arith.ebnf > arith.ser
This is oops version 1.0
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using oops generated by oops on Mon Jan 08 15:28:38 GMT+01:00 2001

$ # independently compile the scanner
$ javac Input.java

$ # execute the parser
$ java oops.Compile arith.ser recog.Input input.zb > /dev/null
No suitable Goal for rule expression found; will use a GoalAdapter.
No suitable Goal for rule sum found; will use a GoalAdapter.
No suitable Goal for rule product found; will use a GoalAdapter.
No suitable Goal for rule term found; will use a GoalAdapter.
```

The last step can be executed several times: the serialized parser is read from the file `arith.ser`, an instance of the scanner class `recog.Input` is created, the parser is started and calls the scanner for input symbols from the text file `input.zb`.

The scanner described in the previous section reads from a file, but standard input is also possible; see the reference manual section "Tools". Therefore, during each execution of `oops.Compile`, expressions terminated by semicolons are read and (hopefully) recognized.

If successful, recognition should be a quiet process. However, *oops* arranges for the parser to produce a representation of the recognized input and `oops.Compile` sends a serialized representation to standard output — if possible. For the time being, this output has been suppressed.

Debugging recognition

A parser generated by *oops* can be observed as it recognizes input (the output shown below was modified for the tutorial for clarification):

```
$ # working directory and classpath
$ pwd; export CLASSPATH= ../../jars/oops.jar:..
/tutorial_examples/recog

$ # execute parser in debug mode
$ java oops.Compile -d arith.ser recog.Input > /dev/null
oops.parser.Parser {
expression : ( [{  (  [  sum ]  ";" ) }] ) .
  lookahead {[ empty], ";", "(", NUMBER}
sum : ( (  product [{  ( (  "+" product ) | (  "-" product ) ) }] ) ) .
  lookahead { "(", NUMBER}
product : ( (  term [{  ( (  "*" term ) | (  "/" term ) ) }] ) ) .
  lookahead { "(", NUMBER}
term : ( NUMBER | (  "(" sum ")" ) ) .
  lookahead { "(", NUMBER}
}
2+3;                                          # standard input
1)      try recognition for expression[ 0]
2)      try recognition for sum[ 1]
3)      try recognition for product[ 2]
```

```
4)      try recognition for term[ 3]
5)      term[ 3]                          shift   NUMBER  2.0
6)      term[ 3]                          reduce
7)      product[ 2]                       shift   term[ 3]      2.0
8)      product[ 2]                       reduce
9)      sum[ 1]                           shift   product[ 2]   2.0
10)     sum[ 1]                           shift   "+"     null
11)     try recognition for product[ 4]
12)     try recognition for term[ 5]
13)     term[ 5]                          shift   NUMBER  3.0
14)     term[ 5]                          reduce
15)     product[ 4]                       shift   term[ 5]      3.0
16)     product[ 4]                       reduce
17)     sum[ 1]                           shift   product[ 4]   3.0
18)     sum[ 1]                           reduce
19)     expression[ 0]                    shift   sum[ 1]       2.0
20)     expression[ 0]                    shift   ";"     null
21)     ^D
21)     expression[ 0]                    reduce
```

The *oops* system provides a class `GoalDebugger` that produces the trace shown above. The trace animates the efforts of the parser, which first tries to find an `expression` (1), therefore a `sum` (2), therefore a `product` (3), and therefore a `term` (4). At this point the input, the `NUMBER 2`, matches what the `term` rule expects — named a `shift` in the trace (5).

Since the `NUMBER` is all a `term` requires, the rule

```
term: NUMBER | "(" sum ")";
```

is now finished (6) — called `reduce` in the trace — and the result from `reduce` is reported to the `product` expecting the `term` — another `shift` in the trace (7).

At the moment you can not understand the numbers `2.0` and `3.0` and the text `null` on the `shift` lines.

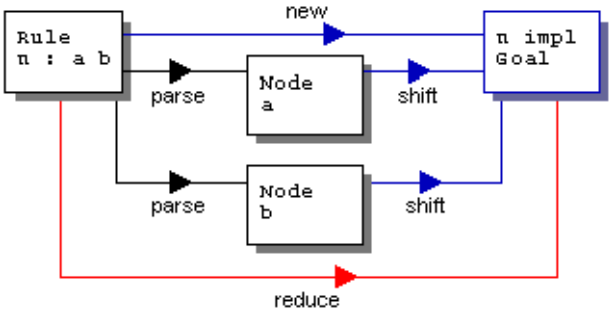
Goal interface

Most of the time the job of a parser is to build a tree representation of the acceptable input. The trace discussed above suggests a mechanism for an *oops* parser to perform actions such as building a tree as recognition progresses.

To understand the interface, first consider how a grammar is used to recognize input: A grammar consists of rules. A rule has a name as a left hand side and a grammar expression as a right hand side. Recognition starts by trying to find a match for the name of the first rule of the grammar. Finding a match for a name means finding a match for it's grammar expression.

The grammar expression consists of quoted strings, identifiers, and operations such as | to indicate an alternative, or various kinds of parentheses for grouping and repetition. A quoted string, termed a `Lit`, is matched if the scanner finds it in the input. An identifier either is a name from a rule, for which another grammar expression must be matched, or it is a category of input symbols such as numbers, termed a `Token`, one of which the scanner must find in the input. Identifiers for rules usually use lower case; for tokens they use upper case.

When a rule needs to be matched, the *oops* parser creates an object implementing the `Goal` interface. See the reference manual section "Goal interface". While the grammar expression of the rule is matched, this `Goal` object receives a `shift()` message as each `Lit`, `Token` and rule name is matched. Once the grammar expression is completely matched, the `Goal` object receives a final `reduce()` message.



The `shift()` messages include parameters: the `Lit` object, the `Token` object and a value object such as an actual number produced by the scanner for the `Token`, and an object created by an inferior `Goal` object in response to a `reduce()` message.

Different `Goal` classes with `shift()` and `reduce()` methods are used to implement actions for a parser.

By default, a `GoalAdapter` remembers the first non-`null` value object received through a `shift()` message and returns it for `reduce()`. A `GoalDebugger` additionally traces the messages. By default, however, if a class matching the name of a rule is found, it will be used rather than the default class.

Now you can understand the numbers and `null` on the `shift` lines in the "Debugging recognition" section from above. A `GoalDebugger` is used for every rule and remembers the first non-`null` value object received through a `shift()` message and returns it for `reduce()`.

Arithmetic interpreter

Continuing now with arithmetic expressions, let the scanner produce a `Double` for each `NUMBER`:

```
public boolean advance () throws IOException {
    if (atEnd()) return false;
    switch (st.nextToken()) {
      ...
      case st.TT_NUMBER:node = new Double(st.nval);
                        tokenSet = NUMBER; break;
      ...
    }
    return true;
}
```

If a `GoalAdapter` is created for each `term`, it will return this `Double` object or the object shifted for a `sum`.

In order to compute products, a class `product` must be implemented that receives two kinds of `shift()` messages for operators and operands. Operators are represented as quoted strings in the grammar.

```
product : term [{ "*" term | "/" term }];
```

and sent through `shift(Lit, Object)`:

```
protected char op;
public void shift(Lit sender, Object value) {
    op = sender.getBody().charAt(0);
}
```

This method simply remembers a character for the last product operator such as `'*'` for multiplication or `'/'` for division. An operand is represented as a `term`, i.e., it arrives through `shift(Goal,`

Object):

```
    public void shift(Goal sender, Object value) {
        if (result == null) {
          result = (Number) value;
          return;
        }
        Number left = result, right = (Number) value;
        switch (op) {
          case '*':
               result = new Double(left.doubleValue() * right.doubleValue());
               break;
          case '/':
               result = new Double(left.doubleValue() / right.doubleValue());
               break;
        }
      }
```

The second parameter was produced by sending `reduce()` to a `GoalAdapter` for a `term`, i.e., it can be a `Double` for a `NUMBER`.

If this is the first `term` of a `product`, no operator has been remembered and the `Double` of the first `term` is simply stored. For each subsequent `term` there is an intervening operator, which is combined with the stored value to form a new value. The final value is returned for `reduce()`:

```
    public Object reduce () {
        return result;
    }
```

For the `sum` rule a class `sum` must be implemented, that similarly deals with addition operators. Lastly, a class `expression` prints each `sum` as it is received:

/tutorial_examples/eval/expression.java

```
package eval;

import oops.parser.GoalAdapter;
import oops.parser.Goal;

public class expression extends GoalAdapter {
  public void shift(Goal sender, Object value)  {
    System.out.println(value);
  }

  public Object reduce() {
    return null;
  }
}
```

This `Goal` returns `null` upon `reduce()` to avoid a serialized output from `oops.Compile`.

Given a scanner, this can be executed:

```
    $ # working directory and classpath
    $ pwd; export CLASSPATH=../../jars/oops.jar:..
    /tutorial_examples/eval

    $ # compile all Goals and the scanner
    $ javac expression.java product.java sum.java Input.java

    $ # serialize the parser
    $ java oops.boot.Oops arith.ebnf > arith.ser
    This is oops version 1.0
    Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
    and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
```

```
Using oops generated by oops on Mon Jan 08 15:28:38 GMT+01:00 2001

$ # set property
$ props="-Doops.parser.DefaultGoalMakerFactory.prefix=eval"

$ # execute the parser
$ java $props oops.Compile arith.ser eval.Input input.zb
No suitable Goal for rule term found; will use a GoalAdapter.
7.2
25.0
$
```

The last step can be executed repeatedly: the serialized parser is read in from the file `arith.ser`, an instance of the scanner class `eval.Input` is created, and the parser is started and will call the scanner for input symbols from the text file `input.zb`.

The property sets the package name for the `Goal` classes. See the reference manual "GoalMaker, GoalMakerFactory and default factories".

A library for trees

Rather than evaluating an arithmetic expression, one can build a formula tree which can be evaluated later. If all nodes of the tree are subclasses of `Number`, the leaves of the tree, i.e., the literal operands, can be `Double` or `Long` objects created by the scanner. Each operator node implements methods such as `intValue()` to combine its operand subtrees and return the numerical result.

For this example, an abstract base class `Node` maps `int`, `short` and `byte` arithmetic to `longValue()` and `float` arithmetic to `doubleValue()`.

/tutorial_examples/tree/Node.java

```
package tree;

import java.io.Serializable;

/** base class to store and evaluate arithmetic expressions.
    Defines most value-functions so that subclasses need only deal
    with long and double arithmetic.
  */
public abstract class Node extends Number implements Serializable {

  /** maps byte arithmetic to long.
      @return truncated long value.
    */
  public byte byteValue () {
    return (byte)longValue();
  }

  /** maps float arithmetic to double.
      @return truncated double value.
    */
  public float floatValue () {
    return (float)doubleValue();
  }
  ...
```

Another abstract base class `Node.Binary` extends `Node` and supports two subtrees. Further subclasses, like `Node.Add`, implement the various arithmetic operations.

/tutorial_examples/tree/Node.java

```
/** represents a binary operator.
 * Must be subclassed to provide evaluation.
 */
protected abstract static class Binary extends Node {
  /** @serial left operand subtree. */
  protected Number left;

  /** @serial right operand subtree. */
  protected Number right;

  /** builds a node with two subtrees.
    @param left left subtree.
    @param right right subtree.
    */
  protected Binary (Number left, Number right) {
    this.left = left; this.right = right;
  }
}

/** implements addition. */
public static class Add extends Binary {
  /** builds a node with two subtrees.
  @param left left subtree.
  @param right right subtree.
  */
  public Add (Number left, Number right) {
    super(left, right);
  }

  /** @return sum of subtree values. */
  public long longValue () {
    return left.longValue() + right.longValue();
  }

  /** @return sum of subtree values. */
  public double doubleValue () {
    return left.doubleValue() + right.doubleValue();
  }
}
```

Tree building

The grammar for arithmetic expressions can be used with a new set of `Goal` classes to compile arithmetic expressions into arithmetic trees. In this case the `shift()` messages combine tree fragments and each `reduce()` message returns a (sub)tree. It turns out that a small change to the grammar can avoid the switch for operator decoding in the `shift()` methods and produce a much more elegant implementation:

/tutorial_examples/tree/arith.ebnf

```
    // arithmetic expressions

    expression      : [{ [  sum ]  ";" }];
    sum             : product [{  "+" product  | "-" product }];
    product         : term [{  product.mul  | product.div }];
    product.mul     : "*" term;
    product.div     : "/" term;
    term            : NUMBER | term.minus | "(" sum ")";
    term.minus      : "-" term;
```

`product` and `term` have been rewritten to employ new rules `product.mul`, etc. New `Goal` classes can be introduced for these rules which support a tree building operation `build`:

/tutorial_examples/tree/build.java

```
package tree;

import oops.parser.GoalAdapter;

public abstract class build extends GoalAdapter {
  public abstract Object build (Object a, Object b);
}
```

A `Goal` class such as `product.mul` knows how to combine two subtrees into a new tree representing multiplication of the two subtrees:

/tutorial_examples/tree/product.java

```
package tree;

import oops.parser.Goal;
import oops.parser.GoalAdapter;

public class product extends GoalAdapter {
  ...
  public static class mul extends build {
    public Object build (Object a, Object b) {
      return new Node.Mul((Number) a, (Number) b);
    }
  }
  ...
}
```

`product.mul` is also a `GoalAdapter`, i.e. for `reduce()` it ill return the subtree recognized by it's `term`.

A `Goal` class such as `product` now receives a first `shift()` with the subtree recognized by `term` and further `shift()` messages from `product.mul` or `product.div`:

/tutorial_examples/tree/product.java

```
 package tree;

import oops.parser.Goal;
import oops.parser.GoalAdapter;

public class product extends GoalAdapter {
  public void shift (Goal sender, Object node) {
    if (result == null) result = node;
    else result = ((build) sender).build(result, node);
  }
  ...
}
```

The first subtree is simply stored. All further subtrees are sent with `build()` back to their senders to form appropriate trees.

The `Goal` for `sum` uses a different strategy to build the tree. The scanner produces a `build` object as a value for the + and − operators:

/tutorial_examples/tree/Input.java

```
   public boolean advance () throws IOException {
    ...
    default:
      char op = (char) st.ttype;
      tokenSet = parser.getLitSet(""+op);
      switch (op) {
        case '+':
          node = new build() {
            public Object build (Object a, Object b) {
              return new Node.Add((Number) a, (Number) b);
            }
          };
          break;
        case '-':
          node = new build() {
            public Object build (Object a, Object b) {
              return new Node.Sub((Number) a, (Number) b);
            }
          };
          break;
      }
      break;
    }
    return true;
  }
```

`sum` now uses this object to build the new tree.

With this strategy the `Goal` class is much more compact and only one rule is needed, but the tree building code is divided between the `Goal` class `sum` and the scanner and necessitates a switch in the scanner.

/tutorial_examples/tree/sum.java

```
package tree;

import oops.parser.Goal;
import oops.parser.GoalAdapter;
import oops.parser.Lit;

public class sum extends GoalAdapter {
  protected build op;

  public void shift (Goal sender, Object node) {
    if (result == null) result = node;
    else result = op.build(result, node);
  }

  public void shift(Lit lit, Object node) {
    op = (build) node;
  }
}
```

In `expression` finally, a `Vector` of `Number` objects is constructed:

/tutorial_examples/tree/expression.java

```
package tree;

import java.util.Vector;
import oops.parser.Goal;
import oops.parser.GoalAdapter;

/** collect lines: [{ [ sum ] ';' }] in a Vector. */
public class expression extends GoalAdapter {
  /** stores a tree for each sum.
  @serial lines Vector with one tree per sum
  */
  protected Vector lines = new Vector();

  /** presents result of reduction.
  @param sender just received reduce().
  @param node was created by sender.
  */
  public void shift (Goal sender, Object node) {
    lines.addElement(node);
  }

  /** concludes rule recognition.
    @return Vector of sum trees.
    */
  public Object reduce () {
    return lines;
  }
}
```

Tree execution

If this parser is executed using `oops.Compile`, the topmost `reduce()` returns a non-`null` result which `oops.Compile` writes to standard output as a serialized object. Another program `tree.Go`

reads a serialized `Vector` of `Number` objects from standard input and prints the `Number` in some types.

/tutorial_examples/tree/Go.java

```
package tree;

import java.lang.ClassCastException;

import java.io.ObjectInputStream;
import java.util.Vector;

/** executes arithmetic expressions from standard input.
  */
public class Go {
  /** loads a vector with Number elements from standard input
      and evaluates them.
      @param args ignored
    */
  public static void main (String args []) {
    try {
      ObjectInputStream in = new ObjectInputStream(System.in);
      Vector lines = (Vector)in.readObject();
      System.out.println("byte\tshort\tint\tlong\tfloat\tdouble");
      for (int i = 0; i < lines.size(); ++ i) {
        Number n = (Number)lines.elementAt(i);
        System.out.println(n.byteValue()+"\t"+n.shortValue()
                           +"\t"+n.intValue()+"\t"+n.longValue()
                           +"\t"+n.floatValue()+"\t"+n.doubleValue());
      }
    } catch (Exception e) { System.err.println(e);}
  }
}
```

With the input...

/tutorial_examples/tree/input.zb

```
2+3;
32768 * 2147483648 + 128;
4 * 9223372036854775807;
4 * 9.2;
```

... the parser builds a `Number` tree for the arithmetic expression, `oops.Compile` serializes the result from `expression`, `Go` deserializes the `Vector` of `Number`s and prints the value in different types:

```
$ # working directory and classpath
$ pwd; export CLASSPATH=../../jars/oops.jar:..
/tutorial_examples/tree

$ # compile Goals and build
$ javac build.java expression.java product.java sum.java term.java
$ # compile scanner, Node and Go
$ javac Go.java Node.java Input.java

$ # serialize the parser
$ java oops.boot.Oops arith.ebnf > arith.ser
This is oops version 1.0
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using oops generated by oops on Mon Jan 08 15:28:38 GMT+01:00 2001
```

```
$ # set package property
$ props="-Doops.parser.DefaultGoalMakerFactory.prefix=tree"

$ # execute the parser
$ java $props oops.Compile arith.ser tree.Input input.zb > tree.ser

$ # execute Go
$ java tree.Go < tree.ser
byte    short   int     long    float   double
5       5       5       5       5.0     5.0
-128    128     128     70368744177792  7.0368744E13    7.0368744177792E13
-4      -4      -4      -4      3.6893488E19    3.689348814741911E19
36      36      36      36      36.8    36.8
```

`Go` demonstrates that the execution of the parser `arith.ser` constructs trees for the arithmetic expressions in `input.zb` which can be evaluated in different ways. This parser can be viewed as a compiler for arithmetic expressions which outputs executable code in the form of serialized trees.

3
Conflicts

oops checks if a grammar is LL(1) and therefore not ambiguous, i.e., if there is only one way (if any) in which the grammar matches any input string.

Grammars for typical programming languages such as Java often are not LL(1) because they do not decide the dangling else problem:

/tutorial_examples/conflicts/conflicts.ebnf

```
stmt : "stmt"
         | "if" "cond" "then" stmt [  "else" stmt ] ;
```

By convention for the input

```
if cond then
   if cond then stmt else stmt
```

the "else" is associated with the innermost "if", but the grammar does not require that.

When *oops* checks the grammar, the conflict is reported:

```
$ # set classpath
$ CLASSPATH=../../jars/oops.jar:..

$ # serialize the parser
$ java oops.boot.Oops conflicts.ebnf > conflicts.ser
This is oops version 1.0
Copyright Axel-Tobias Schreiner (axel@informatik.uni-osnabrueck.de)
and Bernd Kuehl (bernd@informatik.uni-osnabrueck.de).
Using oops generated by oops on Tue Jan 09 14:52:54 GMT+01:00 2001
oops.parser.Opt, warning:
[  (  "else" stmt ) ]: ambiguous, will shift
  lookahead {[ empty], "else"}
  follow {EOF, "else"}
```

The example from above shows that the parser does a shift:, i.e., recognizes the longest possible input stream and indeed associates the "else" with the inner most "if":

```
$ # compile the scanner
$ javac Input.java

$ # execute the parser
$ java oops.Compile -d conflicts.ser conflicts.Input
oops.parser.Parser {
stmt : ( "stmt" | (  "if" "cond" "then" stmt [  (  "else" stmt ) ] ) ) .
  lookahead {"stmt", "if"}
}
if cond then
   if cond then stmt else stmt
try recognition for stmt[0]
stmt[0] shift   "if"    null
stmt[0] shift   "cond"  null
stmt[0] shift   "then"  null
try recognition for stmt[1]
stmt[1] shift   "if"    null
stmt[1] shift   "cond"  null
stmt[1] shift   "then"  null
```

```
try recognition for stmt[ 2]
stmt[ 2]  shift   "stmt"  null
stmt[ 2]  reduce
stmt[ 1]  shift   stmt[ 2]  null
stmt[ 1]  shift   "else"  null          # here the else is shifted to
                                        # the second if.

try recognition for stmt[ 3]
stmt[ 3]  shift   "stmt"  null
^D
stmt[ 3]  reduce
stmt[ 1]  shift   stmt[ 3]  null
stmt[ 1]  reduce
stmt[ 0]  shift   stmt[ 1]  null
stmt[ 0]  reduce
```

4 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