Guide to the use of Sable

Parts of the Sable File

1. Package,
2. Helper,
3. Tokens,
4. Ignored Tokens,
5. Productions.

Package

eg Package hi.compiler;

Defines what package the Java files will be in.

Helpers

This section defines regular expresions used in the lexer

Regular expresions made up of:

- characters either
  - in quotes’z’, ’S’,
  - or decimal 10, 13
- charactersets

  ```
  [’a’..’z’]
  [[’a’..’z’]+[’A’..’Z’]]
  [[’A’..’Z’]-’E’]
  ```

Regular expresions continued

- bracketed regular expression (<regexp>)

- an alternation of regular expressions eg:

```
'a'|['0'..'9']|'Z'
```

- a string eg : **'then'**

- a helper id eg : **alpha**

- a sequence of regular expressions

```
'a' 'c' ('d'|'e')
```

repetitions

- a regular expression followed by a $*$
  - eg: `alpha*` stands for zero or more alphas

- a regular expression followed by a $+$
  - eg: `digit+` stands for one or more digits

Helpers continued

regular expressions can be named

example

———-

Helpers

```
    digit = ['0'..'9'];
    lwcase = ['a'..'z'];
    upcase = ['A'..'Z'];
    letter = lwcase | upcase;
    alphanum = letter|digit;
```

helper and other names must be lower case

Tokens

```
begin='begin';
then ='then'
id = letter alphanum*;
number = digit+;
```

Productions

These list the non terminals of the language being defined eg:

```
program      = fndecl + ;
actuallist   = expr comma;
actualparams= {emptyparams}|
               {paramlist} actuallist* expr;
atom         = {literal} literal|
               {paramname} id|
               {comp}comp;
```

Note that each alternative in a multiway production was given a label in braces thus {paramlist}. At most one unlabeled alternative is allowed for each production.

First production is the root of the parse and should be called program.

Labelled sub-productions

Suppose a production has two occurences of the same non-terminal. For example

```
addexp= exp plus exp;
```

Sable forces us to label the two occurences differently:

```
addexp= [left]:exp plus [right]:exp;
```

Generated java

If we invoke sable on a sable language definition file with package name Foo we get a collection of directories as follows:

```
/Foo
   /analysis
   /lexer
   /node
   /parser
```

within the /Foo/node directory we get a collection of class files 1 for each production or branch of a production

thus the production `actuallist= expr comma;` produces a java file:

AActuallist.java

Alternative production classes

the production

```
actualparams= {emptyparams}|
              {paramlist} actuallist* expr;
```

produces 3 classes:

AParamlistActualparams.java

AEmptyparamsActualparams.java

PActualparams.java

PActualparams will be an abstract class:

```
/* This file was generated by SableCC (http://
package Hi.node;
public abstract class PActualparams extends No
{
}
```

which the other two classes implement

AEmptyparamsActualparams

```
/* This file was generated by SableCC (http://
package Hi.node;
import java.util.*;
import Hi.analysis.*;
public final class AEmptyparamsActualparams ex
{
public AEmptyparamsActualparams()
{
}
public Object clone()
{
return new AEmptyparamsActualparams();
}
public void apply(Switch sw)
{
((Analysis) sw).caseAEmptyparamsActualparams(t
}
```

```
public String toString()
{
return "";
}
void removeChild(Node child)
{
}
void replaceChild(Node oldChild, Node newChild
{
}
}
```

AParamlistActualparams

```
/* This file was generated by SableCC (http://
package Hi.node;
import java.util.*;
import Hi.analysis.*;
public final class AParamlistActualparams exte
{
    private final LinkedList _actuallist_ = new
    private PExpr _expr_;
    public AParamlistActualparams()
    {
    }
    public AParamlistActualparams(
        List _actuallist_,
        PExpr _expr_)
    {
        .....
    }
```

```
public AParamlistActualparams(
    XPActuallist _actuallist_,
    PExpr _expr_)
{
    ........
}
public Object clone()
{
    .....
}
```

The following is used by walker methods

```
public void apply(Switch sw)
{
    ((Analysis) sw).caseAParamlistActualpa
}
```

Field access methods

```
public LinkedList getActuallist()
{
    return _actuallist_;
}
public void setActuallist(List list)
{
    ......
}
public PExpr getExpr()
{
    return _expr_;
}
public void setExpr(PExpr node)
{
  ....
}
public String toString()
{
    return ""
        + toString(_actuallist_)
        + toString(_expr_);
}
```

The following are used to rewrite the tree during analysis

```
        void removeChild(Node child)
        {
             .....
        }
        void replaceChild(Node oldChild, Node newC
        {
            ....
        }
    }
```

The lexer

The lexer generated by sable has a main class in our case Hi.lexer.Lexer

this has a constructor that is passed in a PushbackReader which will typically be sugared input file. Its methods need not concern you. It is then passed to the parser when the parser is constructed.

Parser

The parser takes a lexer in its constructor and has one method of interest

parse() which returns a syntax tree for the file pointed to by the lexer, if there is a syntax error, an exception will be thrown.

Invoking the parser

```
package Hi;
import Hi.node.*;import Hi.lexer.*;import Hi.p
import java.io.*;
public class Main
{
public static void main(String[] arguments)
{// Assume single input file parameter
Parser parser;
try{
  FileReader r= new FileReader(arguments[0]+".
  PushbackReader pr = new PushbackReader( new
  Lexer lexer = new Lexer( pr );
  Node ast = new Parser(lexer).parse();
 }
 catch(Exception e)h{
  System.out.println(e);
  System.out.println("exit");
 }
}
}
```

Walkers

The basic classes here are DepthFirstAdapter and ReversedDepthFirstAdapter both in the directory analysis.

They visit each node in turn in the tree. For each class of node there are 2 methods

1. An In method that is called as the traversal goes down the tree

2. An Out method that is called as the traversal goes back up the tree.

```
public void
inAExternalBody(AExternalBody node)
 { defaultIn(node); }
public void
outAExternalBody(AExternalBody node)
 { defaultOut(node); }
```

These are typically overridden in a class you write that extends one of these adaptor classes. In the simplest case:

Pass 1 we go over the tree and build up symbol tables

Pass 2 we go over the tree and output assembler

Working store for the analysers is provided by a couple of hash tables, into which <key,value> pairs can be inserted: the In table and the Out table.

methods setIn(object,object) setOut(object, object), getIn(object)->object, getOut(object)->object are provided.

It is probably useful to add your own additional dictionary instances to your analysis classes for instance to hold types of variables etc.