Handling Left Recursion in Packrat Parsers

Ruedi Steinmann

March 26, 2009

Contents

1 Introduction

This paper describes an extension to packrat parsing ("Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking" by Bryan Ford), allowing it to handle all forms of left recursion.

The algorithm is roughly equivalent to the algorithm described in "Packrat Parsers Can Support Left Recursion" by Alessandro Warth, James R. Douglass and Todd Millstein. But another approach to explanation and implemenation is taken. It is strongly suggested to make sure you are familiar with the above paper up to and including section 3.3. Many concepts are reused in this paper, but introduced in a very breef way.

The algorithm is different from previous approaches by the way it detects left recursion. Detection occurs at the first encounter of left recursion, whereas other approaches keep track the recursion depth and stop as soon it exceeds the number of remaining tokens.

("Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars" by Paul C. Callaghan, Richard A. Frost and Rahmatullah Hafiz) ("A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time" by Richard A. Frost and Rahmatullah Hafiz)

We will first state rules and invariants which define the desired behaviour of our algorithm in an informal way. Then a framework written in Java is presented. It allows to use Java instead of pseudo code for the presentaion of the algorithm.

In the next section the algorithm is presented, along with the reasoning for it's correct behoviour.

Finally we come to the conclusion and the presentation of some experimental results.

2 Definitions

The grammar is represented as a set of productions. A production can be applied at a given input position. The result is either failure or a tuple of 'value', 'position' and 'tentative/definitive'.

'value' is a semantic value, specific to the application domain. 'position' is the next input position to be parsed after the production. 'tentative/definitive' declares if the result is to be considered definitive, or if the result is involved in a left recursion (more on this later).

Invariant 2.1. *The position of a production appliance attempt (AA) result is eqal to the position of the AA, or advanced.*

A production can apply other productions, recursively. This leads to the idea of the active production stack (APS). An APS entry contains the rule which is currently tried to be applied and the input position where the AA started. The APS grows from the bottom to the top.

Invariant 2.2. *An APS entry is pushed onto the stack before any other production appliance attempt is caused by that attempt.*

Theorem 2.3. *An APS entry input position is never before the input position of any entry below it.*

Proof: Follows from the fact that lower entries are applied first and from invariant 2.1. □

Invariant 2.4. *A given production at a given input position is pushed at most once onto the APS.*

2.1 Left Recursion

Rule 2.5. *Left recursion occurs and only occurs if an AA is started and the APS already contains an entry E which the given rule and input position. The active production of E is called the recursion's causing active production.*

Definition 2.6 (Reapplication)**.** *A tentative result can be reapplied by starting the application of the result's causing active production again.*

The next rules introduce the concept of tentative results. The whole parsing algorithm is based on reapplying productions which caused left recursion. Attributing such results tentative is at the heart of the reasoning process in this paper.

The tentative attribute indicates that a result is dependant of a left recursion, and is therefore not final.

Rule 2.7. *Let LR be a left recursion whose occurence does not depend an a tentative result (no infulence of a tentative result on the control flow).*

Let P be LR's active production and R be the result of the left recursive appliance attempt.

If P is beeing reapplied, R is the reapplied result. Otherwise, the value of R is failed, it is attributed tentative and the causing production is P.

Rule 2.8. *Whenever a left recursion occurs, depending on a tentative result A, a temporary result B is created as stated in the definitive case in rule 2.7.*

Let R denote the result of the whole AA. R's value is the value of B. R is tentative.

R's causing active production is the causing active production of A or B which lies deeper on the stack.

Invariant 2.9. *The tentative attribute is transitive, i.e. a result based on a tentative result is tentative itself, with the same causing active production.*

The following rules state how to turn a tentative result R into a definitive one.

Rule 2.10. *A tentative result R can be considered definitve if its reapplication does not return a grown result. (more input consumed)*

Rule 2.11. *A tentative result R can be considiered definitve if its reapplication fails.*

Please note that the reapplication of R could also return a definitive result, which is perfectly legal.

Rule 2.12. *If the return value R of an active production P is tentative, R's causing production may not be P.*

The invariants and rules make a first tentative result act like a 'seed', which is grown as long as progress is possible.

3 The Framework

Many papers use pseudo code to present their algorithms. This often suffers from semantic ambiguity. Therefore a concrete implementation of the algorithm in Java is presented. It uses literate programming to combine source code and documentation. ("Literate Programming" by Donald E. Knuth, submitted to THE COMPUTER JOURNAL)

The framework parses a language description and generates an internal representation of the productions. This representation is visualized as a forest using the dot program.

Then it reads an input file and parses it with the language description. The resulting parse tree is again visualized using dot.

3.1 Production Representation

Let's start with the representation of productions.

«Productions»:=

```
public class Productions
  extends HashMap<String, Production>
{
    Production start=null;

    @Override
    public Production put(
    String key,
    Production value)
  {
        if (start==null) start=value;
        return super.put(key, value);
    }

    public void check() {
        for (Production p:values()) {
            p.expression.check(this);
        }
    }

    private static final long
    serialVersionUID = 1L;

    void genDot(File fn){
        Dot dot=new Dot(fn);
        for (Production p:values()) {
            p.genDot(dot);
        }
        dot.close();
    }
}
```

«Production»:=

```
public class Production {
    public String name;
    public Expression expression;

  <<genDot>>
  <<apply>>
}
```

«Expression»:=

```
interface Expression {
  // generates a graph from the expression
    void genDot(Dot dot, String id);

  // tries to parse the expression
    MatchedExpression parse(Parser parser);

  // run after parsing the language,
  // before parsing the input
    void check(Productions productions);
}
```

The expression types are represented by classes implementing Expression.

«Choice»:=

```
public class Choice
  extends Vector<Expression>
  implements Expression
{
  <<body>>
}
```

«Sequence»:=

```
public class Sequence
  extends Vector<Expression>
  implements Expression
{
  <<body>>
}
```

«CharToken»:=

```
public class CharToken
  implements Expression
{
    public char ch;

    public CharToken(char charValue) {
        ch=charValue;    }

    <<body>>
}
```

«ProductionExpression»:=

```
public class ProductionExpression
  implements Expression
{
    public String identifier;

    public ProductionExpression(String string) {
        identifier=string;
    }

  <<body>>
}
```

3.2 Parsing the Language Description

For generating a parser for the language description, "VisualLangLab, A Visual LL(k) Parser Generator, Ver-0.826(3)" by Sanjay Dasgupta is used.

In VLL, grammars are produced using a GUI, and it provides a java-style lexer. Unfortunately, there is no BNF-like output option, so the following BNF is hand-written.

```
Input       = Production+ .
Production  = IDENT '=' Expression ';' .
Expression  = Sequence | Choice | Symbol
Sequence    = '(' Expression* ')'
Choice      = '[' Expression+ ']'
Symbol      = CHAR | ProductionExpression
ProductionExpression = IDENT
```

IDENT is a java identifier, CHAR is a java character constant.

The mapping from the language description syntax to the internal production representation should be self-explanatory.

After parsing is complete, Main.parsed() is called with the productions as argument. parsed() starts the check on the productions, the input parsing and the generation of the output graph.

3.3 AST Representation

Parsed input is stored in an AST. The classes used therefore are stored in the package packrat.matched

«MatchedProduction»:=

```
public class MatchedProduction {
    Production original;
    public MatchedExpression expression;

    public MatchedProduction(
    Production original,
    MatchedExpression expression)
  {
        this.original=original;
        this.expression=expression;
    }

  <<body>>
}
```

«MatchedExpression»:=

```
public interface MatchedExpression {
    void printAST(Dot dot, String parent);
}
```

«MatchedSequence»:=

```
public class MatchedSequence
  extends Vector<MatchedExpression>
  implements MatchedExpression
{
    Sequence original;

    public MatchedSequence(Sequence original) {
        this.original=original;
    }

  <<body>>
}
```

«MatchedChoice»:=

```
public class MatchedChoice
  implements MatchedExpression
{
    Choice original;
    MatchedExpression expression;

    public MatchedChoice(
    Choice original,
    MatchedExpression expression)
  {
        this.original=original;
        this.expression=expression;
    }

  <<body>>
}
```

«MatchedSymbol»:=

```
public class MatchedSymbol
  implements MatchedExpression
{
  ProductionExpression original;
    MatchedProduction production;

    public MatchedSymbol(
    ProductionExpression original,
    MatchedProduction production)
  {
        this.original=original;
        this.production=production;
    }

  <<body>>
}
```

«MatchedChar»:=

```
public class MatchedChar
  implements MatchedExpression
{
    CharToken original;
  <<body>>
}
```

4 Parsing Algorithm

4.1 Memo

The parsing algorithm stores results of AAs in a memo table.

Invariant 4.1. *Memo table entries represent definitive results.*

For any given position, an entry is stored exactly once. The entry can be modified later, but cannot be replaced by another entry. This is ensured by the access methods.
«Memo»:=

```
public class MemoEntry {
  Production production;
  long endPos;
  MatchedProduction result;
}

private HashMap<
    Long,
    HashMap<Production, MemoEntry>>
  memo =new HashMap<Long,
      HashMap<Production,MemoEntry>>();

public MemoEntry getMemoEntry(
  Production p,
  long startPos)
{
  if (!memo.containsKey(startPos)) return null;
  return memo.get(startPos).get(p);
}

// creates a new entry for given
// production/position, and returns it
public MemoEntry setMemoEntry(
  Production p,
  long pos)
{
  <<setMemoEntryBody>>
}
```

4.2 Active Production Stack

Due to the access requirements, the APS is represented using two datastructures. Access methods keep them consistent. An APSentry always contains a tentative result.

When pushing a fresh APSEntry E, the result stored in E is failed, and the causing active production is set to E. This satisfies invariant 4.2.

Invariant 4.2. *Let LR, P and R have the same meaning as in rule 2.7.*

Then the result stored in the APSEntry for P satisfies the requirements stated in rule 2.7 for R.

«APS»:=

```
public class APSEntry{
  Production production;
  long position;
  MatchedProduction result;
  APSEntry causingProduction;
  int depth;
  long endPos;
}

private HashMap<
  Long,
  HashMap <Production, APSEntry>>
  apsCache
    =new HashMap<
      Long,
      HashMap<Production,APSEntry>>();

private LinkedList<APSEntry>
  apsStack=new LinkedList<APSEntry>();

// return: entry or null
public APSEntry apsSearch(Production p, long pos) {
  if (apsCache.get(pos)==null) return null;
  return apsCache.get(pos).get(p);
}

// creates a new entry for given
// production/position, and returns it
public APSEntry apsPush(Production p, long pos) {
  <<apsPushBody>>
}

public void apsPop(){
  APSEntry e=apsStack.pop();
  apsCache.get(e.position)
    .remove(e.production);
}
```

4.3 Production Application

The return value of the production application has no way to express if the result is definitive or tentative. To pass this information, the parser object is used.

«Parser»:=

```
public class Parser {
    long pos=0;
    RandomAccessFile input;
    Productions productions;

    boolean resultTentative=false;
    APSEntry causingActiveProduction;

  <<Memo>>
  <<APS>>
  <<apsMarkRecursive()>>

    public Parser(Productions productions) {
        this.productions=productions;
    }

    public MatchedProduction parse(File fn) {
        try {
            input=new RandomAccessFile(fn,"r");
            return productions.start.apply(this);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch (ParseError e) {
            System.out.println(e.getMessage());
        }
        return null;
    }
}
```

We'll need the following invariants.

Invariant 4.3. *If a cache entry is present for a given production and position, no APS entry exists for the same rule and position.*

Invariant 4.4. *When Production.apply() is called and parser. resultTentative is true, the rule application depends on a tentative result with causing active production parser.causingActiveProduction.*

When Production.apply() is called and parser. reslutTentative is false, the rule application does not depend on a tentative result.

Invariant 4.5. *When Production.apply() returns and parser. resultTentative is true, the result is tentative with causing active production parser.causingActiveProduction.*

When Production.apply() returns and parser. reslutTentative is false, the rule application returned a definitive result.

The following is the outline of the method for production application. Some blocks were marked, in order to reference to them later in the text.
«apply»:=

```
// Helper Function for tracing
public MatchedProduction apply(Parser parser) {
  APSEntry apsEntry
    =parser.apsSearch(this, parser.pos);
   MemoEntry cacheEntry
      =parser.getMemoEntry(this,parser.pos);
  Trace.productionStart(
    parser,this,apsEntry,cacheEntry);
  MatchedProduction p=applyH(parser);
    Trace.productionEnd(parser,this,p);
    return p;
}

public MatchedProduction applyH(Parser parser) {
  MemoEntry cacheEntry
    =parser.getMemoEntry(this,parser.pos);

  if (cacheEntry!=null) {
    //BLOCK A
    parser.pos=cacheEntry.endPos;
    return cacheEntry.result;
  }

  APSEntry apsEntry
    =parser.apsSearch(this, parser.pos);

  if (apsEntry!=null) {
    // We've got some sort recursion
    <<Handle Recursion>>
  } else {
    <<Prepare Application>>
    // reapplication loop
    do {
      <<apply rule>>
      <<Return or Reapply>>
    } while (true);
  }
}
```

BLOCK A is reached only if a cache entry is present. By invariant 4.3, no APS entry for the current production is present. By invariant 2.2 and 2.5, no left recursion can have occured. By invariant 4.1, the result found in the cache entry is defintive.

The block does not affect parser parser.resultTentative and parser.causingActiveProduction.

If parser.resultTentative is true, by invariant 4.4, the invocation of BLOCK A depends on a tentative result. By invariant 2.9, the result has to be tentative, by 4.5.

If parser.resultTentative is false, the result is definitive, by 4.5.

4.3.1 Recursion Handling

By invariant 2.2 and rule 2.5, the following code snippet is executed whenever a recursion starts.

«Handle Recursion»:=

```
//BLOCK F
if (!parser.resultTentative
    || apsEntry.causingProduction.depth
     <parser.causingActiveProduction.depth)
{
        parser.causingActiveProduction
      =apsEntry.causingProduction;
}
parser.resultTentative=true;
parser.pos=apsEntry.endPos;
return apsEntry.result;
```

If parser.resultTentative is false, by 4.4, the AA does not depend on a tentative result. Thus rule 2.7 applies. By 4.2, the result stored in the APSEntry just has to be returned, which is accomplished by the code snippet.

If parser.resultTentative is true, by 4.4, the AA depends on a tentative result R. R's causing rule is stored in parser.causingActiveProduction. By rule 2.8, we have to choose the active production which lies deeper on the stack (smaller APSEntry.depth field).

4.3.2 Return or Reapply

After the production is applied, resulting in the result p, the following code snippet is executed. If it returns, the result is used as result of the AA. If it continues, the production is tried to be applied again.
«Return or Reapply»:=

```
if (!parser.resultTentative) {
  //BLOCK B
    cacheEntry
        =parser.setMemoEntry(this, startPos);

    cacheEntry.result=p;
    cacheEntry.endPos=parser.pos;

    parser.apsPop();
    return p;
} else {
  // tentative result
    if (parser.causingActiveProduction!=apsEntry) {
    //BLOCK C
        parser.apsPop();
        return p;
    } else {
        if (reapplied
        && (p==null || parser.pos<=lastEndPos)) {
      // reapplication failed or did not return
      // a grown result, make definitive
      //BLOCK D
            parser.apsPop();
            cacheEntry
            =parser.setMemoEntry(this, startPos);
            parser.resultTentative=false;
            parser.pos=apsEntry.endPos;

            cacheEntry.result=apsEntry.result;
            cacheEntry.endPos=apsEntry.endPos;

            return cacheEntry.result;
        } else {
      //BLOCK E
            // reapply
            apsEntry.result=p;
            apsEntry.endPos=parser.pos;
            lastEndPos=parser.pos;
            parser.pos=startPos;
            reapplied=true;
            Trace.retry(this);
            continue;
        }

    }
}
```

Showing correctness of the above code snippet requires quite a lot of work. We will look at each block separately.

- **BLOCK B** By invariant 4.5, the p is definitive. It is therefore save the result in the cache and return it. (see also invariant 4.1)
- **BLOCK C** As rule 2.12 does not apply in this case, the value can be returned.
- **BLOCK D** The result in the APSEntry was reapplied and did either fail or not grow. By rules 2.11 and 2.10, we can make the result definitive.
- **BLOCK E** In this block, the only thing allowed to do is to reapply the result

5 Results

To demonstrate the proper behaviour of the algorithm, some examples are presented. They all contain left recursive rules. For each example, first the language description is given, then the input and finally the resulting AST.

5.1 Expressions

Language:

```
input= (space expressionDash);

expressionDash = [add sub expressionDot];
add = (expressionDash '+' space expressionDash);
sub = (expressionDash '-' space expressionDash);

expressionDot = [mul div value];
mul = (expressionDot '*' space expressionDot);
div = (expressionDot '/' space expressionDot);

value = [number parenthesis];
parenthesis= ('(' space expressionDash ')' space);


number= ([(digit number) digit ] space);
digit= ['1' '2' '3' '4' '5' '6' '7' '8' '9' '0'];
```

```
space = [' ' ()];
```

Input:

```
1+2 *3+ 4
```

5.2 Java Snippet

Language:

```
Primary = PrimaryNoNewArray;
PrimaryNoNewArray=[
    ClassInstanceCreationExpression
    MethodInvocation
    FieldAccess
    ArrayAccess
    THIS];

ClassInstanceCreationExpression = [
    (NEW ClassOrInterfaceType EMPTYPARENTHESIS)
    (Primary DOT NEW Identifier EMPTYPARENTHESIS)];
MethodInvocation = ([(Primary DOT) ()] MethodName EMPTYPARENTHESIS);

FieldAccess = ([Primary SUPER] DOT Identifier);

ArrayAccess = ([Primary ExpressionName] '[' space Expression ']' space);

ClassOrInterfaceType = [ ClassName InterfaceTypeName ];
ClassName = ([ 'C' 'D'] space);
InterfaceTypeName  = ([ 'I' 'J'] space);

Identifier  = [([ 'x' 'y'] space) ClassOrInterfaceType];
MethodName = ([ 'm' 'n'] space);
ExpressionName = Identifier;
Expression = ([ 'i' 'j'] space);

EMPTYPARENTHESIS = ('(' space ')' space);
DOT = ('.' space);
NEW = ('n' 'e' 'w' space);
SUPER= ('s''u''p''e''r' space);
THIS = ('t''h''i''s' space);
space = [' ' ()];
```

Input:

```
this.x[i].m()
```

5.3 Freaky

Language:

```
sS = [( sS 's' sS) ()];
```

Input:

```
ssss
```

6 Conclusion

The extended packrat algorithm solves one of the most serious drawbacks of the packrat algorithm. Although the effect on parsing speed was not examined, it seems plausible, that the extension only adds a small penalty. Specially as the active production stack could be merged with the cache.

The handling of left recursion seems correct given the provided examples. It is based on a satisfying theoretical foundation, altough the proof is incomplete.

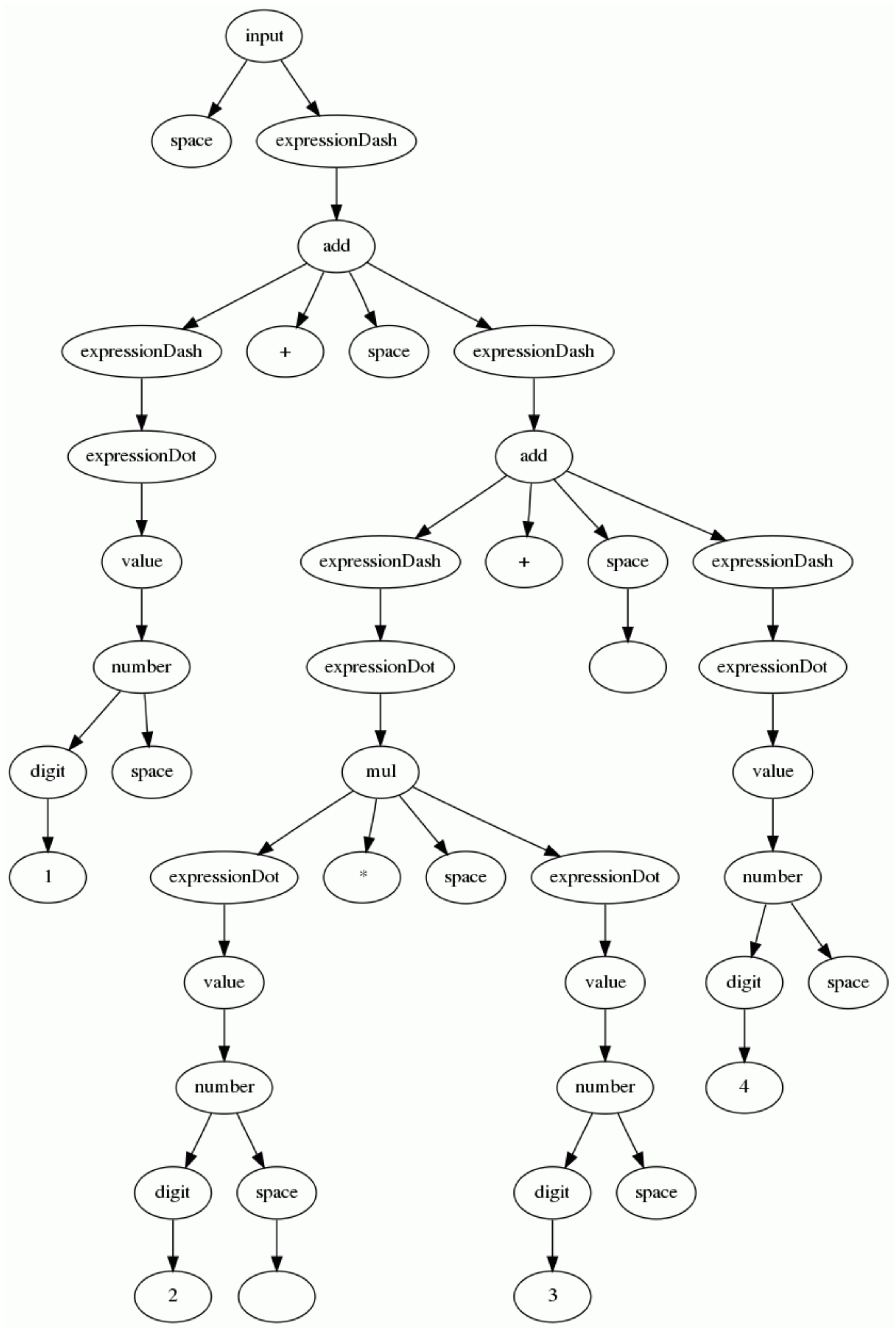

Figure 1: Expression

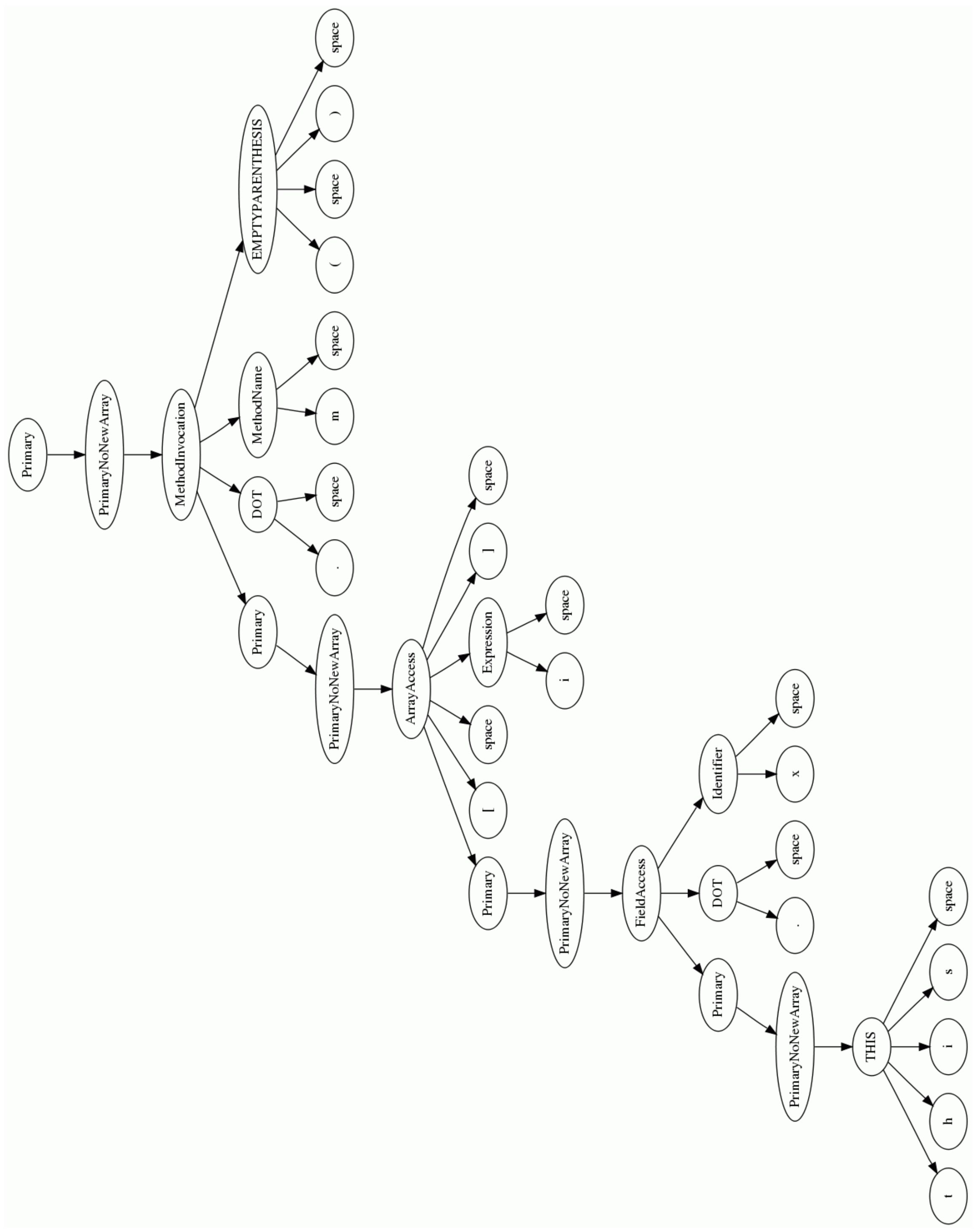

Figure 2: Java Snippet

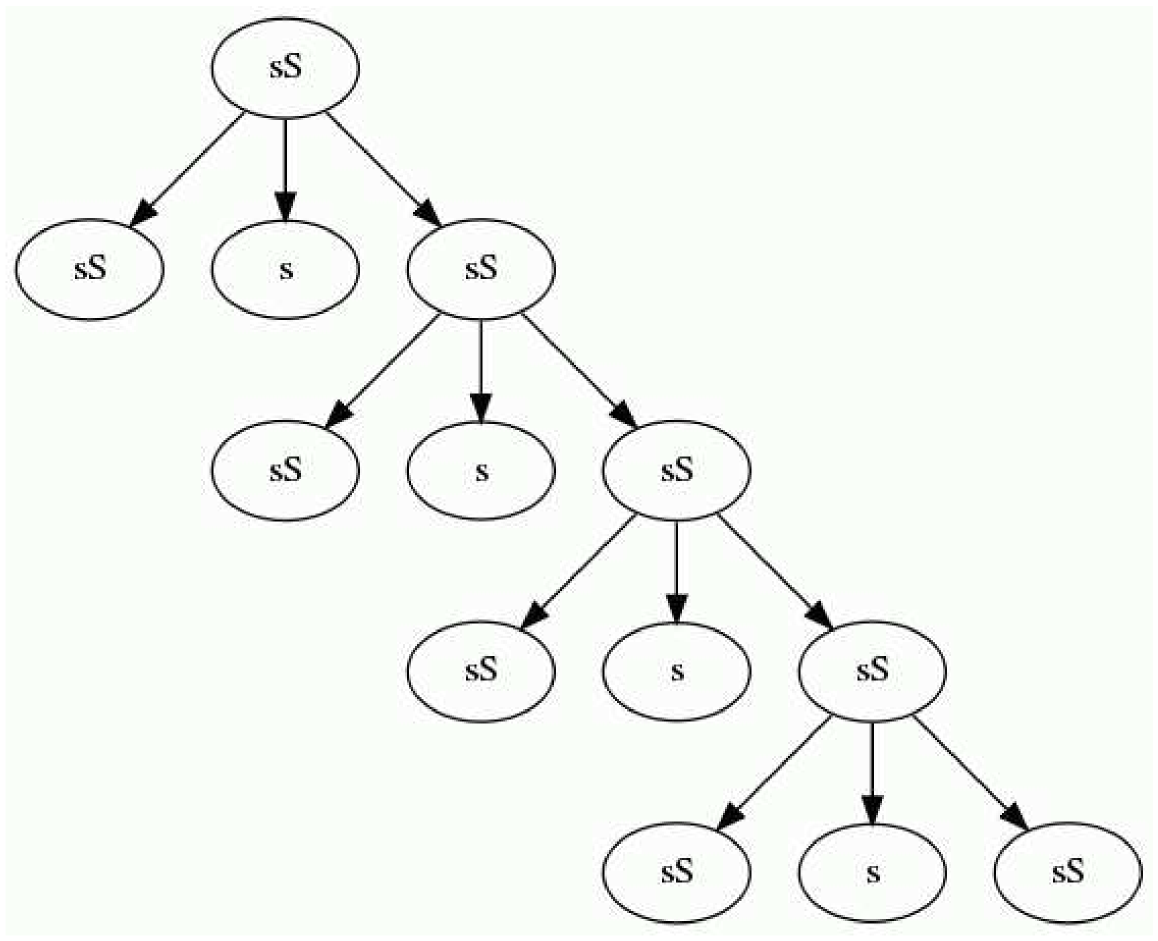

Figure 3: Freaky