ANTLR und Eclipse-Xtext – zwei ll Kumpane

Roman Weissgärber

email: r.w @ c.a

r: **roman**

w: **weissgaerber**

c: **chello**

a: **at**

Abstract:

Die Eclipse IDE erfreut sich nicht nur unter Java Entwicklern einer wachsenden Beliebtheit. Als modular aufgebaute Opensource-Plattform unterstützt sie die Entwicklung von eigenen Erweiterungen (Plugins) vielfältig.
Mittels der Eclipse-Xtext Komponente welche auf den Parsergenerator ANTLR zurückgreift, kann man relativ einfach Editoren für DSLs (Domain Specific Language) generieren. Die mittels einer Grammatik und mehr oder weniger simplen Code-Ergänzungen definierten Editoren sind in die Eclipse IDE integriert und weisen dann die üblichen Merkmale wie Syntax-Coloring, Hyperlinks, Outline-View usw. auf.

Als Beispiel für so eine DSL dienen uns die Dateien des Konfigurationssystems, welches im Linux-Kernel (Kconfig, defconfig) oder auch im Buildroot-Projekt Verwendung findet. Es wird zuerst auf die Entwicklung einer entsprechenden Grammatik mittels ANTLR eingegangen. Dann wird mittels Xtext ein Eclipse Editor generiert.

ANTLR ANother Tool for Language Recognition

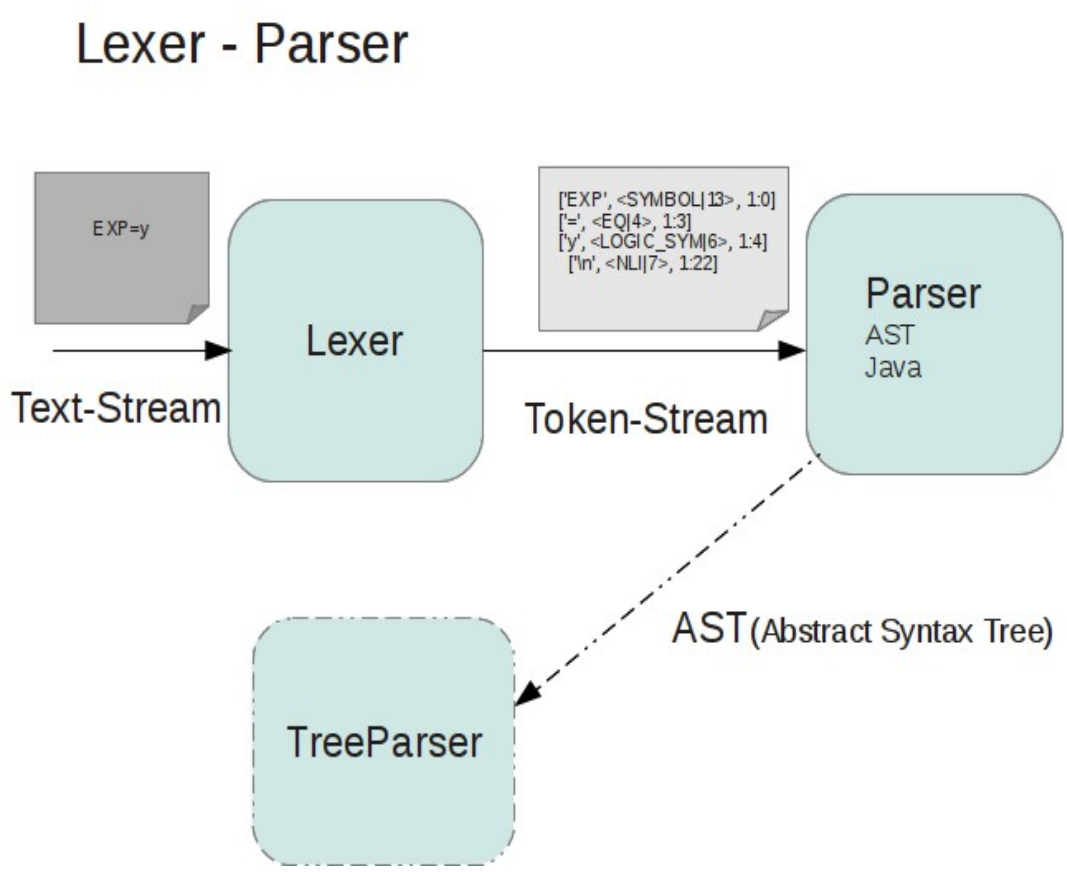
http://www.antlr.org

Hauptentwickler: Terence Parr

- Parsergenerator-Framework um Compiler, Interpreter und ähnliches zu erstellen
- ll(*)-Grammatik, Top Down Parser
- Java, verschiedene Target-Sprachen zB. Java, Javascript, C
- Lexer, Parser, AST-TreeParser, (Stringtemplates)
- GPL – General Purpose Language (C, Java, …) zB. alternativer Parser für openJDK
- DSL – Domain Specific Language (SQL, Linux Kconfig-Dateien,…)

ANTLR IDE

- Eclipse 3.7- Indigo (zB. Eclipse IDE for Java Developers), Dynamic Language Toolkit (DLTK) Core 3.0.0, GEF 3.7.0 mit Zest 1.3.0
  http://www.eclipse.org
- ANTLR IDE v2.1.2
  http://antlrv3ide.sourceforge.net
- ANTLR v3.4
  http://www.antlr.org



ANTLR Grammatik (EBNF Extended Backus-Naur Form)

- *Regeln* NL='\n';
- *Sequenz* KW_IF='if'; KW_IF='i' 'f';
- *Alternative* WS=' '|'\t';
- *Komplementärmenge* ~'\n'
- *Klammern* ()
- *Wertebereich* 'a'..'d', 'a'|'b'|'c'|'d'
- *Quantoren* ? *0,1 mal,* * *0..un mal,* + *1..un mal*
- *Java Code einfügen* {}

DSL (Domain Specific Language)

Das Dateiformat zum Abspeichern der Konfiguration des Linux Kernel (kbuild-System) „defconfig“ wird als Beispiel für eine DSL herangezogen. Ein defconfig-File besteht aus mehreren Zeilen. Jede Zeile enthält entweder eine Config-Element-Zuweisung bestehend aus *SYMBOL* '=' *VALUE* oder eine Kommentarzeile beginnend mit '#'.

VALUE kann entweder ein String mit doppelten oder einfachen Anführungszeichen oder eine Integer- oder Hexadezimalkonstante sein. *VALUE* kann aber auch einen dreiwertigen Logikwert darstellen (y, m, n).

Kommentarzeilen können entweder einfache Kommentarzeilen oder spezielle verneinende Kommentare mit impliziter Zuweisung der Form

 # SYMBOL is not set

sein. Diese Kommentarform ist gleichbedeutend mit:

SYMBOL=n

Hier das Testfile das im folgenden Verwendung findet:

```
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES    = y

CONFIG_LOG_BUF_SHIFT=19
# comment
CONFIG_ZBOOT_ROM_BSS= 0x80ff
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
# CONFIG_INPUT_MOUSE is not set
CONFIG_SPI=m
```

Input Beispiel (test/a.defcon)

Für alle Java-Klassen der folgenden Beispiele findet das Java-Default-Package Verwendung.

Lexer

Es folgt die Lexer Grammatik DefconfigLexer.g welche mit ANTLR in eine Java-Klasse Defconfig.java übersetzt wird.

```
lexer grammar DefconfigLexer;
options { language = Java; }

NLI : '\n';
WSS : (' '|'\t')+ {$channel=HIDDEN;};

EQ : '=';

LOGIC_SYM : 'y'|'m'|'n';
S_HEX : '0' ('x'|'X') ('0'..'9'|'A'..'F'|'a'..'f')+;
S_INT : '-'? ('0'..'9')+;
SYMBOL : ('A'..'Z'|'a'..'z'|'0'..'9'|'_')+;

STRING : ( '"' s=STR_d '"' | '\'' s=STR_s '\'' ) {setText($s.text);};
  fragment STR_d: ((Esc | ~(('\\'|'"'))))*;
  fragment STR_s: ((Esc | ~(('\\'|'\''))))*;
  fragment Esc : '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'\''|'\\');

SL_COMMENT_NOT : '# ' SYMBOL ' is not set' {setText($SYMBOL.text);};
SL_COMMENT : '#' ~('\n')* {$channel=HIDDEN;};
```

DefconfigLexer.g (ANTLR Grammatik/Lexer)

Mit dem folgenden Testprogramm kann man den Lexer ausführen.

```
import java.io.FileInputStream;

import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.Lexer;
import org.antlr.runtime.Token;

public class TestLexer {
  public static void main(String[] args) throws Exception {

    FileInputStream fis = new FileInputStream("test/a.defcon");
    ANTLRInputStream input = new ANTLRInputStream(fis);

    Lexer lexer = new ExtDefconfigLexer(input); // mit Tokennamen
//  Lexer lexer = new DefconfigLexer(input); // ohne Tokennamen

    Token t;
    do {
      t = lexer.nextToken();
      System.out.println(t);
    } while (t.getType() != Token.EOF);
  }
}
```

TestLexer.java

Als Eingabe für den Lexer dient das Testfile.

```
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES    = y

CONFIG_LOG_BUF_SHIFT=19
# comment
CONFIG_ZBOOT_ROM_BSS= 0x80ff
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
# CONFIG_INPUT_MOUSE is not set
CONFIG_SPI=m
```

Input Beispiel (test/a.defcon)

Mit den obigen Testdaten erhalten wir folgende Token-Ausgabe:

```
['CONFIG_EXPERIMENTAL', <SYMBOL|13>, 1:0]
['=', <EQ|4>, 1:19]
['y', <LOGIC_SYM|6>, 1:20]
[' ', <WSS|16>,channel=99, 1:21]
  ['\n', <NLI|7>, 1:22]
['CONFIG_MODULES', <SYMBOL|13>, 2:0]
['  \t', <WSS|16>,channel=99, 2:14]
['=', <EQ|4>, 2:17]
[' ', <WSS|16>,channel=99, 2:18]
['y', <LOGIC_SYM|6>, 2:19]
  ['\n', <NLI|7>, 2:20]
  ['\n', <NLI|7>, 3:0]
['CONFIG_LOG_BUF_SHIFT', <SYMBOL|13>, 4:0]
['=', <EQ|4>, 4:20]
['19', <S_INT|15>, 4:21]
  ['\n', <NLI|7>, 4:23]
['# comment', <SL_COMMENT|8>,channel=99, 5:0]
  ['\n', <NLI|7>, 5:9]
['CONFIG_ZBOOT_ROM_BSS', <SYMBOL|13>, 6:0]
['=', <EQ|4>, 6:20]
[' ', <WSS|16>,channel=99, 6:21]
['0x80ff', <S_HEX|14>, 6:22]
  ['\n', <NLI|7>, 6:28]
['CONFIG_UEVENT_HELPER_PATH', <SYMBOL|13>, 7:0]
['=', <EQ|4>, 7:25]
['/sbin/hotplug', <STRING|10>, 7:26]
  ['\n', <NLI|7>, 7:41]
['CONFIG_INPUT_MOUSE', <SL_COMMENT_NOT|9>, 8:0]
  ['\n', <NLI|7>, 8:31]
['CONFIG_SPI', <SYMBOL|13>, 9:0]
['=', <EQ|4>, 9:10]
['m', <LOGIC_SYM|6>, 9:11]
['<EOF>', <EOF|-1>, 9:12]
```

Output Lexer (DefconfigLexer.g,TestLexer.java)

Ein weiterer Durchlauf folgt nun mit einer Fehlerhaften Eingabe.

```
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES    = y
mmm
CONFIG_LOG_BUF_SHIFT=19??
# comment
CONFIG_ZBOOT_ROM_BSS= 0x80ff
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
# CONFIG_INPUT_MOUSE is not set
CONFIG_SPI=m
```

Input Beispiel mit Fehlern (test/a_err.defcon)

Da es sich bei *mmm* um ein Symbol handelt, erkennt der Lexer hier keinen Fehler. Dieser Fehler wird erst vom Parser gemeldet. Die *??* werden jedoch erkannt.

```
['CONFIG_EXPERIMENTAL', <SYMBOL|13>, 1:0]
['=', <EQ|4>, 1:19]
['y', <LOGIC_SYM|6>, 1:20]
[' ', <WSS|16>,channel=99, 1:21]
  ['\n', <NLI|7>, 1:22]
['CONFIG_MODULES', <SYMBOL|13>, 2:0]
['  \t', <WSS|16>,channel=99, 2:14]
['=', <EQ|4>, 2:17]
[' ', <WSS|16>,channel=99, 2:18]
['y', <LOGIC_SYM|6>, 2:19]
  ['\n', <NLI|7>, 2:20]
['mmm', <SYMBOL|13>, 3:0]
  ['\n', <NLI|7>, 3:3]
['CONFIG_LOG_BUF_SHIFT', <SYMBOL|13>, 4:0]
['=', <EQ|4>, 4:20]
['19', <S_INT|15>, 4:21]
  ['\n', <NLI|7>, 4:25]
line 4:23 no viable alternative at character '?'
line 4:24 no viable alternative at character '?'
['# comment', <SL_COMMENT|8>,channel=99, 5:0]
  ['\n', <NLI|7>, 5:9]
['CONFIG_ZBOOT_ROM_BSS', <SYMBOL|13>, 6:0]
['=', <EQ|4>, 6:20]
[' ', <WSS|16>,channel=99, 6:21]
['0x80ff', <S_HEX|14>, 6:22]
  ['\n', <NLI|7>, 6:28]
['CONFIG_UEVENT_HELPER_PATH', <SYMBOL|13>, 7:0]
['=', <EQ|4>, 7:25]
['/sbin/hotplug', <STRING|10>, 7:26]
  ['\n', <NLI|7>, 7:41]
['CONFIG_INPUT_MOUSE', <SL_COMMENT_NOT|9>, 8:0]
  ['\n', <NLI|7>, 8:31]
['CONFIG_SPI', <SYMBOL|13>, 9:0]
['=', <EQ|4>, 9:10]
['m', <LOGIC_SYM|6>, 9:11]
['<EOF>', <EOF|-1>, 9:12]
```

Output Lexer Error (DefconfigLexer.g,TestLexer.java)

Um bei der Ausgabe der Token auch den Tokennamen zu erhalten, wird folgende Hilfsklasse benötigt.
Da der ANTLR-Lexer keine Tabelle zur Übersetzung der Tokennummer in einen Tokennamen zur Verfügung stellt, wird folgende Klasse benötigt um die Tokennamen zu Tokennummer Zuordnung aus der Lexer-Klasse mittels Reflexion zu erhalten. Die Token sind als *static final public int* Felder in der DefconfigLexer-Klasse abgelegt. Wobei der Feldname dem Tokennamen entspricht.
Die mittels dieser neuen Lexer-Klasse überschriebene nextToken-Methode liefert Token mit einer toString-Methode, welche auch den Tokennamen ausgibt.

```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.util.HashMap;

import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.CommonToken;
import org.antlr.runtime.Token;


public class ExtDefconfigLexer extends DefconfigLexer {
  static private HashMap<Integer, String> tokenNames =
                                    new HashMap<Integer, String> ();
  static {
    final int mod = Modifier.STATIC | Modifier.FINAL | Modifier.PUBLIC;

    for (Field fd: DefconfigLexer.class.getDeclaredFields()) {
      if (fd.getModifiers()== mod &&
                "int".equals(fd.getType().getCanonicalName())) {
        try {
          tokenNames.put(fd.getInt(DefconfigLexer.class), fd.getName());
        } catch (Exception e) {}
      }
    }
  }

  public ExtDefconfigLexer(ANTLRInputStream input) {
    super(input);
  }

  public Token nextToken() {
    return new CommonToken (super.nextToken()) {
      private static final long serialVersionUID = 1L;

      public String toString() {
        String text = getText().replaceAll("\n","\\\\n").
                  replaceAll("\r","\\\\r").replaceAll("\t","\\\\t");

        return (getText().contains("\n")?"  ":"") +
           "['"+text+"', <"+tokenNames.get(getType())+"|"+getType()+">"+
           (getChannel()>0?(",channel="+getChannel()):"")+", "+
           getLine()+":"+getCharPositionInLine()+"]\n";
      }
    };
  }
}
```

ExtDefconfigLexer.java

Parser

Der folgende Parser enthält keinerlei Aktionen und erzeugt auch keinen AST.

```
parser grammar DefconfigParser;

options {
  language = Java;
  tokenVocab = DefconfigLexer;
}

ce_list: ce?  (eol ce)*  eol? EOF;

ce:  SYMBOL EQ LOGIC_SYM | SL_COMMENT_NOT | SYMBOL EQ val;

val: STRING | S_HEX | S_INT;

eol: NLI+;
```

DefconfigParser.g (ANTLR Grammatik/Parser ohne Aktionen)

```
import java.io.FileInputStream;

import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.Lexer;


public class TestParser {
      public static void main(String[] args) throws Exception {

            FileInputStream fis = new FileInputStream("test/a.defcon");
            ANTLRInputStream input = new ANTLRInputStream(fis);

            Lexer lexer = new DefconfigLexer(input);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            DefconfigParser parser = new DefconfigParser(tokens);
            parser.ce_list();
      }
}
```

TestParser.java

Eine fehlerfreie Eingabe erzeugt hier keine Ausgabe. Werden jedoch Fehler eingefügt, dann erhält man Fehlermeldungen.

```
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES    = y
mmm
CONFIG_LOG_BUF_SHIFT=19??
# comment
CONFIG_ZBOOT_ROM_BSS= 0x80ff
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
# CONFIG_INPUT_MOUSE is not set
CONFIG_SPI=m
```

Input Beispiel mit Fehlern (test/a_err.defcon)

```
line 3:0 no viable alternative at input 'mmm'
line 4:23 no viable alternative at character '?'
line 4:24 no viable alternative at character '?'
```

Output Parser Error (DefconfigLexer.g,DefconfigParser.g,TestParser.java)

Parser-Java

```
parser grammar DefconfigParserJava;

options {
  language = Java;
  tokenVocab = DefconfigLexer;
}

@members {
  public java.util.List<cE> ceList = new java.util.LinkedList<cE>();
  public class cE {
     public String id, val;
     public cE(String i, String v) {val=v; id=i;}
  }
}

ce_list: (ce1=ce {ceList.add($ce1.ce_);})?
                     (eol ce2=ce {ceList.add($ce2.ce_);})*  eol? EOF;

ce returns [cE ce_]:
    id=SYMBOL EQ v=LOGIC_SYM {$ce_=new cE($id.text, $v.text);}
  | id=SL_COMMENT_NOT        {$ce_=new cE($id.text, "n");}
  | id=SYMBOL EQ val         {$ce_=new cE($id.text, $val.text);};

val: STRING|S_HEX|S_INT;

eol: NLI+;
```

DefconfigParserJava.g (ANTLR Grammatik/Parser mit Javacode)

```
import java.io.FileInputStream;
import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.Lexer;

public class TestParserJava {
    public static void main(String[] args) throws Exception {

        FileInputStream fis = new FileInputStream("test/a.defcon");
        ANTLRInputStream input = new ANTLRInputStream(fis);

        Lexer lexer = new DefconfigLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        DefconfigParserJava parser = new DefconfigParserJava(tokens);
        parser.ce_list();
        for (DefconfigParserJava.cE ce: parser.ceList) {
            if(ce!=null)
                 System.out.println("export "+ce.id+"='"+ce.val+"'");
        }
    }
}
```

TestParserJava.java

Um Aktionen auszuführen, kann man die Grammatik mit Java-Programm-Code ergänzen. In diesen Beispiel wird eine Liste von Config-Elementen aufgebaut. Dies geschieht durch die parser.ce_list()-Methode die den Parser aufruft. Anschließend wird die Liste ausgegeben.

Ein Testlauf ergibt:

```
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES    = y

CONFIG_LOG_BUF_SHIFT=19
# comment
CONFIG_ZBOOT_ROM_BSS= 0x80ff
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
# CONFIG_INPUT_MOUSE is not set
CONFIG_SPI=m
```

Input Beispiel (test/a.defcon)

```
export CONFIG_EXPERIMENTAL='y'
export CONFIG_MODULES='y'
export CONFIG_LOG_BUF_SHIFT='19'
export CONFIG_ZBOOT_ROM_BSS='0x80ff'
export CONFIG_UEVENT_HELPER_PATH='/sbin/hotplug'
export CONFIG_INPUT_MOUSE='n'
export CONFIG_SPI='m'
```

Output Parser/Java (DefconfigLexer.g,DefconfigParserJava.g,TestParserJava.java)

Parser-AST

```
parser grammar DefconfigParserAST;

options {
  language = Java;
  output = AST;
  tokenVocab = DefconfigLexer;
}

tokens { CE; CEList; }

ce_list: ce?  (eol ce)*  eol? EOF -> ^(CEList ce*);

ce:
    id=SYMBOL EQ v=LOGIC_SYM -> ^(CE $id $v)
  | id=SL_COMMENT_NOT        ->
              ^(CE $id {new CommonTree(new CommonToken(LOGIC_SYM,"n"))})
  | id=SYMBOL EQ val         -> ^(CE $id val)
;

val: STRING|S_HEX|S_INT;

eol: NLI+;
```

DefconfigParserAST.g (ANTLR Grammatik/Parser mit AST-Generierung)

```
tree grammar DefconfigTreeParser;

options {
  language = Java;
  tokenVocab = DefconfigParserAST;
  ASTLabelType = CommonTree;
}

ce_list: ^(CEList  ce*) ;

ce: ^(CE sym val)
            {System.out.println("export "+ $sym.sy+"='"+ $val.v+"'");};

sym returns [String sy]:
  ( s=SL_COMMENT_NOT
  | s=SYMBOL
  )      {sy = $s.text;}
;

val returns [String v]:
  ( s=S_INT
  | s=S_HEX
  | s=STRING
  | s=LOGIC_SYM
  )   {v =$s.text;}
;
```

DefconfigTreeParser.g (ANTLR Grammatik/AST-Parser)

```java
import java.io.FileInputStream;

import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.CommonTokenStream;
import org.antlr.runtime.Lexer;
import org.antlr.runtime.tree.CommonTree;
import org.antlr.runtime.tree.CommonTreeNodeStream;

public class TestParserAST {
      public static void main(String[] args) throws Exception {

            FileInputStream fis = new FileInputStream("test/a.defcon");
            ANTLRInputStream input = new ANTLRInputStream(fis);

            Lexer lexer = new DefconfigLexer(input);
            CommonTokenStream tokens = new CommonTokenStream(lexer);

            DefconfigParserAST parser = new DefconfigParserAST(tokens);
            DefconfigParserAST.ce_list_return r =parser.ce_list();
            CommonTree t = (CommonTree)r.getTree();

            System.out.println(t.toStringTree());

            CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
      //    nodes.setTokenStream(tokens);
            DefconfigTreeParser walker = new DefconfigTreeParser(nodes);
            walker.ce_list();
      }
}
```

TestParserAST.java

Eine alternative Möglichkeit ist mit dem Parser einen AST (Abstract Syntax Tree) aufbauen zu lassen und diesen dann mit einem TreeParser, der wiederum mit Java-Programm-Code ergänzt wird, abarbeiten zu lassen.

```
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES    = y

CONFIG_LOG_BUF_SHIFT=19
# comment
CONFIG_ZBOOT_ROM_BSS= 0x80ff
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
# CONFIG_INPUT_MOUSE is not set
CONFIG_SPI=m
```

Input Beispiel (test/a.defcon)

Hier erfolgt die Ausgabe des AST (CEList (CE ...) ...) nach dem ersten Parser-Schritt (ParserAST) und danach die Ausgabe der Config-Elemente durch den TreeParser.

```
(CEList (CE CONFIG_EXPERIMENTAL y) (CE CONFIG_MODULES y) (CE
CONFIG_LOG_BUF_SHIFT 19) (CE CONFIG_ZBOOT_ROM_BSS 0x80ff) (CE
CONFIG_UEVENT_HELPER_PATH /sbin/hotplug) (CE CONFIG_INPUT_MOUSE n) (CE
CONFIG_SPI m))

export CONFIG_EXPERIMENTAL='y'
export CONFIG_MODULES='y'
export CONFIG_LOG_BUF_SHIFT='19'
export CONFIG_ZBOOT_ROM_BSS='0x80ff'
export CONFIG_UEVENT_HELPER_PATH='/sbin/hotplug'
export CONFIG_INPUT_MOUSE='n'
export CONFIG_SPI='m'
```

Output Parser-AST (DefconfigLexer.g,DefconfigParserAST.g,DefconfigTreeParser.g,TestParserAST.java)

Eclipse - Xtext

http://www.eclipse.org/Xtext

Hauptentwickler: itemis AG

- Grammatik ähnlich ANTLR, Lexer und Parser in einer Datei
- generiert AST und Eclipse-Editor
- AST: Eclipse EMF Ecore Modell
- Outline
- Syntax-Coloring
- validieren, Quick Fix
- Templates

Vorgangsweise:

- Eclipse 3.7 (Indigo) mit Xtext 2.0.1 installieren
- new Xtext Project (new/other/Xtext/Xtext Project)
  Project Name: site.rw.defcon
  Language Name: site.rw.defcon.Defcon
  Extension: defcon
- im src Verzeichnis des ersten Projektes im *Defcon*.xtext File (enthält Greeting-Beispiel) die entsprechende Grammatik eintragen. Die Zeilen **grammar** ... und **generate** ... sollen dem *Language Name* entsprechen (im Greetings-Beispiel sollten die richtigen Einträge Name/URI aufscheinen).
- MWE2 Workflow mit Run ausführen.
- Fertiges Projekt ausführen
- Anpassungen in den generierten Projekten durchführen

```
grammar site.rw.defcon.Defcon
hidden( WSS, SL_COMMENT)
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

generate defcon "http://www.rw.site/defcon/Defcon"

ceList: {ceList} (ces+=ce)? (eol ces+=ce)* eol?;

ce:
          {ceString} name=SYMBOL EQ v=value
        | {ceLogic} name=SYMBOL EQ v=LOGIC_SYM
        | {ceLogic} name=SL_COMMENT_NOT
;

value :
          S_HEX
        | S_INT
        | STRING
;

eol: NLI+;

// Lexer Rules
terminal NLI : '\n';
terminal WSS : (' '|'\t')+ ;

terminal EQ : '=';

terminal LOGIC_SYM : ('y'|'n'|'m');
terminal S_HEX : '0' ('x'|'X') ('0'..'9'|'A'..'F'|'a'..'f')+;
terminal S_INT : '-'? ('0'..'9')+;
terminal SYMBOL : ('A'..'Z'|'a'..'z'|'0'..'9'|'_')+;

terminal STRING : ( '"' STR_d '"' | '\'' STR_s '\'' );
terminal fragment STR_d: ((Esc | !(('\\'|'"'))))*;
terminal fragment STR_s: ((Esc | !(('\\'|'\''))))*;
terminal fragment Esc : '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'\''|'\\');

terminal SL_COMMENT_NOT : '# ' SYMBOL ' is not set';
terminal SL_COMMENT : '#' !('\n')*;
```

Defcon.xtext

Das Testfile, mit dem nach entsprechenden Adaptionen erhaltenen Xtext Eclipse Editor, geöffnet:

```
CONFIG_EXPERIMENTAL=y
CONFIG_MODULES    = y

CONFIG_LOG_BUF_SHIFT=19
# comment
CONFIG_ZBOOT_ROM_BSS= 0x80ff
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
# CONFIG_INPUT_MOUSE is not set
CONFIG_SPI=m
```

Xtext Editor /Syntax-Coloring