

The Aurochs parser generator



© 2007 Exalead SA

Released under the GNU LGPL

<http://aurochs.fr/>

Author: Berke Durak

What is parsing?

Parsing takes text as a linear sequence of symbols and returns structured data “representing” its meaning.

```
(nude OR naked) AND britney  
AND NOT spears
```

Parsing – an example

Input text:

```
(nude OR naked) AND britney  
AND NOT spears
```

Parse tree:

```
And (  
  Or (Term "nude", Term "naked"),  
  And (Term "britney",  
    Not (Term "spears")))
```

Parsing techniques

- Hand-written parser
 - Only for the simplest cases
 - Difficult to write, maintain and debug
- Regular expressions
 - Mostly used for *lexing* or line-by-line parsing
 - Limited: cannot express nested constructs
 - Usually not good enough for full syntax (even for configuration files)

Lexing

- Transforms a sequence of characters into a sequence of tokens

```
(nude or naked) and britney  
and not spears
```

lexes as

```
[LPAREN] [WORD nude] [OP_OR] [WORD naked]  
[RPAREN] [OP_AND] [OP_NOT] [WORD spears]
```

- Usually done with regular expression technology

Parsing techniques – beyond lexing

- Context-free grammars
 - A natural, robust class of languages
 - Can express the syntax of many computer languages and large fragments of human languages
 - Can be easily interpreted as rewriting rules

Context-free Grammars

Example: Boolean queries

`expr ::=`

`expr 'AND' expr`

`| expr 'OR' expr`

`| 'NOT' expr`

`| '(' expr ')'`

`| WORD`

Disjunction is commutative

CFGs and subsets

- CFG drawbacks:
 - No efficient algorithm for parsing general CFG (dynamic programming with cubic complexity)
 - Ambiguity: can have many parse trees for a given input
- CFG subsets:
 - LL(k): recurse left, reduce left, lookahead k
 - LR(k): recurse left, reduce right, lookahead k
 - LALR(k): lookahead k (before reducing), recurse left, reduce right : used by Bison/Yacc

Flex & Bison / Lex & Yacc

- Regular expression lexer + LALR(1) grammar
- The standard (POSIX), conventional way of parsing
- Widely used and taught
- Two-phase parsing:
 - Lexical analysis with Lex/Flex (maximum-munch tokenization)
 - Syntax analysis with Bison/Yacc (LALR(1))

Using Lex & Yacc

- Split your language into a lexable part and a parseable part
- Write a lexer in a file “lexer.l”
- Write a parser in a file “parser.y”
- Use Lex to generate lexer code
- Use Yacc to generate parser code
- **Compile the code**
- Run, find bugs
- Repeat above steps

Parse trees & Yacc

- Yacc does not generate parse trees
- For every derived production, it executes a chunk of code
- Your “.y” file contains C code

```
if_then_else_statement:
```

```
IF_TK OP_TK expression CP_TK statement_nsi ELSE_TK statement
```

```
{ $$ = build_if_else_statement($2.location, $3, $5, $7); }
```

```
;
```

Problems with Yacc technology

- Technical:
 - Language must be split into a lexer and a parser
 - Cannot compose grammars with different lexers
 - Grammar files contain code
- Fundamental:
 - Parsing algorithm difficult to understand & implement
 - Ambiguities resolved thru obscure precedence rules that do not compose well

Parse expression grammars

- The input string is seen as an array of characters
 - ['b', 'r', 'i', 't', 'n', 'e', 'y', ' ', 'A', 'N', 'D', ' ', ' ', 's', 'p', 'e', 'a', 'r', 's', '\0']
 - Length: 19 (including EOF), positions : 1,2,3,...,19
- A PEG is made of a number of named productions (start, word, opand, space)

```
start ::= expr EOF;
```

```
expr ::= word opand expr | word;
```

```
opand ::= space "AND" space;
```

```
space ::= ' ' +;
```

```
word ::= [a-z] +;
```

Parse expression grammars

- Each production is a function that starts parsing at input and returns the remaining input, or NULL on failure
 - `char *parse_start(char *input)`
 - `char *parse_word(char *input)`
 - `char *parse_space(char *input)`
 - ...

Parse functions (C model)

- `parse_start("britney and spears")`
 - `parse_expr("britney and spears")`
 - `parse_word("britney and spears")`
 - Gives " and spears"
 - `parse_opand(" and spears")`
 - `parse_space(" and spears")`
 - Gives "and spears"
 - Literal "and"
 - Gives " spears"
 - `parse_space(" spears")`
 - Gives "spears"
 - `parse_word("spears")`
 - Gives ""

PEG parsing model

- The parsing model is thus very simple
 - Mutually recursive set of named parsing functions/expressions
 - A parsing function takes an input, parses it, and returns the remaining input, or NULL if it fails
- Basic parsing functions include:
 - Constant strings `"foo"`
 - Character classes `[a-zA-Z0-9]`
 - Beginning & end of file `BOF EOF`

Combining parse expressions

We can combine parse expressions using regular expression-like operators

– Sequence

- `"foo" " " "bar"`

– Iteration * +

- `[a-z]+`

– Option

- `'-'? [0-9]+`

– Ordered alternation |

- `"on"|"off"`

– Conjunction

- `[aeiou] & [a-z]+`

– Negated sequence

- `~ "spears" [a-z]+`

Calling other parse expressions

- Of course parse expressions can call other parse expressions recursively. Just type their name.
 - `sexpr ::= atom | '(' sexpr_list ')';`
 - `atom ::= [a-z]+;`
 - `sexpr_list ::= sexpr space sexpr_list;`

-

Ordered alternation – n00b pitfall 1

- Unlike Bison, Yacc, Menhir and others, *alternation is ordered from left to right, and matches are final.*
- *Consider the grammar*
 - `word ::= "newb" | "newbie";`
 - `start ::= word dot;`
 - `dot ::= '.';`
- ***The input "newbie." is not accepted!***
 - `word("newbie.") = "ie." (because "newb" matches)`
 - `start("newbie.") = dot(word("newbie")) = dot("ie.") = NULL.`

Left recursion – n00b pitfall 2

- Immediate left recursion is ***VERBODEN***
 - It will fail – i.e. it won't match
 - Yacc grammars are left recursive.
- Example:
 - `word ::= [a-z]+;`
 - `wordlist ::= wordlist ' ' word | word;`
- Won't parse – to compute `wordlist("foo")` we need to call `wordlist("foo")` which calls... Aurochs detects the loop and decides that the rule fails.

What about the parse tree?

- So far, the grammars only describe valid input.
- To get a parse tree, you use **constructors**.

```
word ::= <Word>{[a-z]+}</Word>;
```

```
wordlist ::= <WordList> wordlist_inner </WordList>;
```

```
wordlist_inner ::= word ' ' wordlist | word;
```

On input "foo bar baz" this gives:

```
<WordList>
```

```
  <Word>foo</Word> <Word>bar</Word> <Word>baz</Word>
```

```
</WordList>
```

Constructors

- `<tag> ... </tag>` adds a new node named `tag` in the parse tree
- `{ ... }` creates a token (PCDATA) whose contents is the substring of the input matched by the production.
- `attr: ...` adds an attribute named `attr` to the current node, whose value is the matched substring.
- Inner attributes and tokens are ignored.

How to use Aurochs?

- As a stand-alone tool:
 - You give it a grammar and an input, and it spits out a parse tree in XML-ish format.
 - `aurochs -parse foo.txt mygrammar.peg`
`>mytree`
- As a library
 - Only available from Ocaml
- As a parser generator
 - Takes a grammar and produces a binary parser automaton (.nog) file
 - You run that automaton with the C interpreter

How to use Aurochs?

- C library interfaced to Java, Exascript (thanks Maxime!) & Ocaml
- Load the `.nog` file (binary data)
- Unpack the binary, getting a nog program
- Repeat:
 - Read an input
 - Run the program on it, obtaining a DOM-like parse tree
 - Do whatever is appropriate with the tree

Current limitations

- Implementation is memory hungry
 - Using a sparse datastructure for memoization tables should solve this problem.
- No Unicode support
- Fast enough, but could be faster
 - Large room for implementing optimizations.
- Spaces must be explicitly taken care of
 - Relatively easy surrounding, appending, suffixing, outfixing operators

Advantages

- Grammar files are language-independent
- Can be used as a stand-alone tool
 - Very handy for developing a new grammar.
- Directly constructs a parse tree
 - Which is what most people would do with actions anyway.
- Easy to understand parsing model
 - No non-local precedence rules

Advantages (2)

- Unified lexical and syntactic analysis
 - No need for separate lexing
- Guaranteed linear-time performance
- Often more powerful than LALR
 - Thanks to conjunction and negation operators

Where is Aurochs used?

- Aurochs is already used at Exalead!
 - Jsurre Javascript checker
 - Query parsing
 - JSON parsing
 - ACAP parsing
- In the future:
 - The new Exacsript compiler front-end will probably use Aurochs

Thanks !

Acknowledgements

- Original idea: Bryan Ford's thesis: *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*
- Accelerating pushdown automata with memoization originally due to Cook
 - With extensions by Monien & Mogensen; also see my thesis
- Thanks to **Florian** for letting me work on this and making it open source
- Thanks to **Maxime** for the Exascript bindings and Exa/Java instantiation