Handmade or tool-built?

On the evolution of a parser generator written in D

Kai Nacke

May 10 @ DConf 2019

kai.nacke @ {gmail.com, redstar.de}
Motivation

Writing a parser by hand is easy ... ... and boring!

Why not use a tool?
My goals for a parser generator

My tool should
• generate a parser body from a grammar description in EBNF
• allow the grammar to be augmented with code
• provide some error correction feature
• work standalone and should be CTFE-enabled

My tool should not
• generate a lexer
• have big runtime dependencies
Runtime architecture

- The lexer is a range (InputRange / ForwardRange)
- The (optional) preprocessor filters the range
- The parser does syntax analyzing on the range
- Only part of parser is generated
Interface to parser

• The generated code requires the following functions / properties

  Token tok;

  alias TokenKind = typeof(Token.kind);

  void advance() { }

  bool expect(TokenKind kind) { }

  bool consume(TokenKind kind) { }

• TokenKind must be an enumeration

• Member names are derived from token names

• Interface is still under development!
Tools for parser generation

In the C/C++ world
• yacc and bison
• ANTLR
• Coco/R
• ... and many more!

In the D world
• PEG
• ANTLR
• and now: LLtool

• PEG and ANTLR are excellent tools
• PEG has a different approach to parsing
• ANTLR comes with a huge runtime library
Example: simple expressions

```bash
%token number
%start Expr
%
Expr
   = Term ( ( "+" | "-" ) Term )*
   .

Term
   = Factor ( ( "*" | "/" ) Factor )*
   .

Factor
   = number
   | "(" Expr ")"
   .
```
Internal data structure

- Grammar is stored as graph
- Graph elements are of type **Node**
- Graph can be visualized with dot (specify `-d` on command line)
Internal data structure - attributes

**inner**: Pointer to content of sequence/alternative/group

**link**: Arbitrary list e.g. list of nonterminal occurrences

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeType type;</td>
</tr>
<tr>
<td>Cardinality card;</td>
</tr>
<tr>
<td>string name;</td>
</tr>
<tr>
<td>bool derivesEpsilon;</td>
</tr>
<tr>
<td>bool isProductive;</td>
</tr>
<tr>
<td>bool isReachable;</td>
</tr>
<tr>
<td>TerminalSet firstSet;</td>
</tr>
<tr>
<td>TerminalSet followSet;</td>
</tr>
</tbody>
</table>

**next**: Pointer to next node in sequence

**back**: Pointer to parent node
Only used if last node in sequence
Myth #1: generated parsers are slow

From Oberon-2 grammar

Statement = ...
  | "IF" Expr "THEN" StatementSeq "END"
  | ...

Generated D code

else if (tok.kind == TokenKind.KW_IF) {
  advance();
  parseExpr();
  consume(TokenKind.KW_THEN);
  parseStatementSeq();
  consume(TokenKind.KW_END);
}

The generated code reflects the grammar. No performance penalty added.
Myth #2: Generators are not flexible enough

From the LLtool grammar:

rule
   = (. Node node; .)  
   nonterminal<node>
   "="       
   rhs<node.link>
   (. node.link.back = node; .)
   "."  

• Add {. code .} in any places  
• Pass < parameters > as needed

The generated D code:

```d
void parseRule() {
    Node node;
    parseNonterminal(node);
    consume(TokenKind.Equal);
    parseRhs(node.link);
    node.link.back = node;
    consume(TokenKind.Period);
}
```
Myth #3: Bad error messages

• A hand-generated error message from the Oberon-2 lexer:

```pascal
v := 1A;
^^
Error: 22,13: Found hex constant without trailing H
```

• Error message based on parser-provided information:

```pascal
PROCEDURE (l : List) Get* : Integer;
^*
Error: 25,31: Expected ; but found :
```

• Can we do better? A human can spot that ( ) is missing...
LL what?

• Recursive descent parsers belong to the LL(1) class

• This acronym means:
  • L – the input is read from left to right
  • L – the leftmost nonterminal is expanded first
  • 1 – one token look-ahead is used

• For most programming languages there is no LL(1) grammar
What are LL(1) conflicts?

• The parser uses the current state (= program counter) and the next token to decide about next move
• A conflict occurs if there is more than one possibility for next move
• Example from Oberon-2 grammar:

```plaintext
DeclSeq = ... ProcDecl ";"; ForwardDecl ";"; ... .

ProcDecl = "PROCEDURE" (Receiver)? IdentDef ... .

ForwardDecl = "PROCEDURE" "^" (Receiver)? IdentDef ... .
```

State: in DeclSeq
Next token: “PROCEDURE”
Call ProcDecl or ForwardDecl?
More LL(1) conflicts

• Left recursion also creates LL(1) conflicts
  
  ```
  StatementList = StatementList Statement | .
  Statement = ... ";" .
  ```

• Defines a list of statements, separated by ;

• Can you spot the problem?

  ```
  void parseStatementList() {
    if (tok.kind.among(/* List of tokens */)) {
      parseStatementList();
      parseStatement();
    }
    /* ... */
  }
  ```
LL(1) conflict resolution: Grammar rewriting

• Rewrite grammar
  E.g. rewrite the statement list

\[
\text{StatementList} = \text{StatementList} \text{ Statement} | .
\]
\[
\text{Statement} = ... ";" .
\]

as

\[
\text{StatementList} = ( \text{Statement} )* .
\]

• In some cases result can be difficult to understand
LL(1) conflict resolution: Adding resolvers

• Add custom code to guide decision at runtime
• Syntax is `%if (. bool expression .)`
• Only allowed where LL(1) conflict occurs
• Can use additional information; e.g.

    QualIdent = ( %if (. isModule() .) ident "." )? ident.

uses a symbol table lookup in the D function:

    bool isModule() {
        return tok.val in modules;
    }
Handling of grammar variants

• Language families often have a lot of syntax in common
  • C and C++
  • PIM4 and ISO version of Modula-2

• It is desirable to build one parser for one language family

• Is this possible with a parser generator?
Grammar variants: the token trick

• A lot of rules is triggered by special keywords
  • E.g. `class` is a keyword in C++ but not in C

• Use the following approach
  • The lexer recognizes only identifiers
  • The preprocessor maps keyword identifiers to keyword tokens, based on language family
  • The parser does not see keyword token and does not handle this case
Grammar variants: the variant selector

• The token trick does not always help
  • E.g. there is no special keyword

• I am working on a special feature: the variant selector

• Idea: mark variant specific element

DefinitionModule = ("GENERIC")?!generic "DEFINTIION" "MODULE" identifier ";" .

• Requires bool property **generic** in the parser
The variant selector looks cool, but ...

- It makes elements “invisible”
  - Can introduce non-reachable rules – an error today

- Can unintentionally make elements optional

```
DefinitionModule
    = "DEFINTIION" "MODULE" identifier ";";
    | ("GENERIC" "DEFINTIION" "MODULE" identifier ";")!generic
```

- Requires more thought!
More ideas

• Add a look-ahead heuristic for resolvers
• From Oberon-2 grammar

    Import = ( ident "::" )? ident .

• LL(1) conflict because `ident` is start and successor of `( )`?
• Resolver is based on one more token look-ahead

    bool isAlias() { return lexer.save.moveFront.kind == TokenKind.ColonEqual; }

• Can be generated automatically ... but it is tricky (ANTLR does it)
Even more ideas

• Create LRtool – a parser generator for SLR(1)/LALR(1) grammars
• Output as recursive ascent-descent parser (no parsing tables!)
  • Either via data flow analysis or extended left-corner parsing
• Needs much more investigation
Feedback welcome!

• Clone the source from https://github.com/redstar/LLtool

• Create an issue at https://github.com/redstar/LLtool/issues

• Write me an e-mail
Thank you!
Backup
Syntax of input file

%token identifier, code
%token argument, string
%start lltool
%%
lltool = ( header )? ( rule )+.

header = ( "%start" identifier 
| "%token" tokenlist 
| "%eoi" identifier )* "%%" .
tokenlist = tokendecl ("," tokendecl )* .
tokendecl = (identifier | string) 
( "=" identifier )? .

rule = nonterminal "=" rhs "." .
nonterminal = identifier ( argument )? .
rhs = sequence ( "|" sequence )* .
sequence = ( group 
| identifier ( argument )? 
| string | code 
| "%if" code )* .
group = "(" rhs ( ")" 
| ")?" 
| ")*" 
| ")+" ) .