Package ‘dparser’

November 13, 2017

Title Port of 'Dparser' Package

Version 0.1.8

Imports digest, methods

Suggests rex, covr, testthat, knitr, devtools,

Description A Scannerless GLR parser/parser generator. Note that GLR standing for ``generalized LR'', where L stands for ``left-to-right'' and
R stands for ``rightmost (derivation)''. For more information see <https://en.wikipedia.org/wiki/GLR_parser>. This parser is based on the Tomita (1987) algorithm. (Paper can be found at <http://acl-arc.comp.nus.edu.sg/archives/acl-arc-090501d3/data/pdf/anthology-PDF/J/J87/J87-1004.pdf>).
The original 'dparser' package documentation can be found at <http://dparser.sourceforge.net/>. This allows you to add mini-languages to R (like
RxODE's ODE mini-language Wang, Hallow, and James 2015 <DOI:10.1002/psp4.12052>) or to parse other languages like 'NONMEM' to automatically translate
them to R code. To use this in your code, add a LinkingTo dparser in your DESCRIPTION file and instead of using #include <dparse.h> use
#include <dparser.h>. This also provides a R-based port of the make_dparser <http://dparser.sourceforge.net/d/make_dparser.cat> command called
mkdparser(). Additionally you can parse an arbitrary grammar within R using the dparse() function, which works on most OSes and is mainly for grammar
testing. The fastest parsing, of course, occurs at the C level, and is suggested.

Depends R (>= 3.2.3)

License BSD_3_clause + file LICENSE

Encoding UTF-8

LazyData true

RoxygenNote 6.0.1

NeedsCompilation yes

Author Matthew Fidler [aut, cre],
John Plevyak [aut, cph]

Maintainer Matthew Fidler <matthew.fidler@gmail.com>

Repository CRAN

Date/Publication 2017-11-13 18:11:18 UTC

R topics documented:

`dparser-package` *A Scannerless GLR parser/parser generator*

Description

This package is based on the C dparser https://github.com/jplevyak/dparser

Details

DParser is an simple but powerful tool for parsing. You can specify the form of the text to be parsed using a combination of regular expressions and grammar productions. Because of the parsing technique (technically a scannerless GLR parser based on the Tomita algorithm) there are no restrictions. The grammar can be ambiguous, right or left recursive, have any number of null productions, and because there is no separate tokenizer, can include whitespace in terminals and have terminals which are prefixes of other terminals. DParser handles not just well formed computer languages and data files, but just about any wacky situation that occurs in the real world.

Features

- Powerful GLR parsing
- Simple EBNF-style grammars and regular expression terminals
- Priorities and associativities for token and rules
- Built-in error recovery
- Speculative actions (for semantic disambiguation)
- Auto-building of parse tree (optionally)
- Final actions as you go, or on the complete parse tree
- Tree walkers and default actions (multi-pass compilation support)
- Symbol table built for ambiguous parsing
- Partial parses, recursive parsing, parsing starting with any non-terminal
- Whitespace can be specified as a subgrammar
- External (C call interface) tokenizers and external terminal scanners

- Good asymptotically efficiency
- Comes with ANSI-C, Python and Verilog grammars
- Comes with full source
- Portable C for easy compilation and linking
- BSD license, so you can included it in your application without worrying about licensing

The result is natural grammars and powerful parsing.

The R based tree parsing in `dparse` creates dlls on the fly based on C code. By default the echoing of the compile is disabled, but you can change this by `options(dparser.echo.compile=FALSE)`

Garbage collection

There are two user options that control if the dlls for the grammars created by dparser will be deleted upon garbage collection or R exit if they are not associated with any active objects. These are:

`dpaser.rm.unnamed.parser.dll`**:** when `TRUE`, this remove parsers that are created from strings, or other memory-based items in R.

`dpaser.rm.unnamed.parser.dll`**:** when `TRUE`, this removes parsers created from grammar files.

Creating a Grammar for Parsing

Grammar Comments: Grammars can include C/C++ style comments

Example:

```
// My first grammar
E: E '+' E | "[abc]";
/* is this right? */
```

Grammar Productions: A production is the parts of your language your are trying to parse and are tpyically named. See https://en.wikipedia.org/wiki/Top-down_parsing

- The first production is the root of your grammar (what you will be trying to parse).
- Productions start with the non-terminal being defined followed by a colon ':', a set of right hand sides separated by '|' (or) consisting of elements (non-terminals or terminals).
- Elements can be grouped with parens '(', and the normal regular expression symbols can be used ('+' '*' '?' '|').
- Elements can be grouped with parens '(', and the normal regular expression symbols can be used ('+' '*' '?' '|').
- Elements can be repeated using '@', for example elem@3 or elem@1:3 for repeat 3 or between 1 and 3 times respectively.

Example:

```
 program: statements+ |  comment* (function |  procedure)?;
```

Note: Instead of using '[' ']' for optional elements we use the more familar and consistent '?' operator. The square brackets are reserved for speculative actions (below).

Global C code in Grammars: Since the main parsing of the language grammar is in C, intermixing C code with the grammar can be useful.

- Global (or static) C code can be intermixed with productions by surrounding the code with brackets '{}'.

Example:

```
{ void dr_s() { printf("Dr. S\n"); }
 S: 'the' 'cat' 'and' 'the' 'hat' { dr_s(); } | T;
 { void twain() { printf("Mark Twain\n"); }
     T: 'Huck' 'Finn' { twain(); };
```

Note: When parsing the grammar using mkdparse, the option `use_r_header = TRUE` will redefine `printf` to `Rprintf` to better comply with R packages.

Terminals The terminals are the peices of the language that are being parsed, like language keywords.

- Strings terminals are surrounded with single quotes. For example:

```
block: '{' statements* '}';
whileblock: 'while' '(' expression ')' block;
```

- Unicode literals can appear in strings or as charaters with U+ or u+. For example:

```
U+03c9 { printf("omega\n"); }
```

- Regular expressions are surrounded with double quotes. For example:

```
hexint: "(0x|0X)[0-9a-fA-F]+[uUlL]?";
```

Note: only the simple regular expression operators are currently supported. This include parens, square parens, ranges, and '*', '+', '?'.

- Terminal modifiers

Terminals can contain embbed escape codes. Including the standard C escape codes, the codes \x and \d permit inserting hex and decimal ASCII characters directly.

Tokens can be given a name by appending the $name option. This is useful when you have several tokens which which represent the same string (e.g. ','). For example,

```
function_call: function '(' parameter ( ',' $name 'parameter_comma' parameter) ')';
```

It is now possible to use $0.symbol == $string parameter_comma to differentiate ParseNode ($0) between a parameter comma node and say an initialization comma.

Terminals ending in '/i' are case insensitive. For example 'hi'/i matches 'HI', 'Hi' and "hI' in addition to 'hi'.

- External (C) Scanners

There are two types of external scanners, those which read a single terminal, and those which are global (called for every terminal). Here is an example of a scanner for a single terminal. Notice how it can be mixed with regular string terminals.

```
{
    extern char *ops;
    extern void *ops_cache;
    int ops_scan(char *ops, void *ops_cache, char **as,
              int *col, int *line, unsigned short *op_assoc, int *op_priority);
}

X: '1' (${scan ops_scan(ops, ops_cache)} '2')*;
```

The user provides the ’ops_scan’ function. This example is from tests/g4.test.g in the source distribution.

The second type of scanner is a global scanner:

```
{
   #include "g7.test.g.d_parser.h"
   int myscanner(char **s, int *col, int *line, unsigned short *symbol,
              int *term_priority, unsigned short *op_assoc, int *op_priority)
   {
       if (**s == 'a') {
           (*s)++;
           *symbol = A;
           return 1;
       } else if (**s == 'b') {
           (*s)++;
           *symbol = BB;
           return 1;
       } else if (**s == 'c') {
           (*s)++;
           *symbol = CCC;
           return 1;
       } else if (**s == 'd') {
           (*s)++;
           *symbol = DDDD;
           return 1;
       } else
           return 0;
   }
   ${scanner myscanner}
   ${token A BB CCC DDDD}

   S: A (BB CCC)+ SS;
   SS: DDDD;
```

Notice how the you need to include the header file generated by `mkdparse` which contains the token definitions.

- Tokenizers

Tokenizers are non-context sensitive global scanners which produce only one token for any given input string. Some programming languages (for example C) are easier to specify using a tokenizer because (for example) reserved words can be handled simply by lowering the terminal priority for identifiers.

```
S : 'if' '(' S ')' S ';' | 'do' S 'while' '(' S ')' ';' | ident;
ident: "[a-z]+" $term -1;
```

The sentence: **if (while) a;** is legal because **while** cannot appear at the start of **S** and so it doesn't conflict with the parsing of **while** as an **ident** in that position. However, if a tokenizer is specified, all tokens will be possible at each position and the sentense will produce a syntax error.

DParser provides two ways to specify tokenizers: globally as an option (-T) to mkdparse and locally with a ${declare tokenize ...} specifier (see the ANSI C grammar for an example). The ${declare tokenize ...} declartion allows a tokenizer to be specified over a subset of the parsing states so that (for example) ANSI C could be a subgrammar of another larger grammar. Currently the parse states are not split so that the productions for the substates must be disjoint.

- Longest Match

Longest match lexical ambiguity resolution is a technique used by separate phase lexers to help decide (along with lexical priorities) which single token to select for a given input string. It is used in the definition of ANSI-C, but not in C++ because of a snafu in the definition of templates whereby templates of templates (List<List <Int») can end with the right shift token ('»'). Since **DParser** does not have a separate lexical phase, it does not require longest match disambiguation, but provides it as an option.

There are two ways to specify longest match disabiguation: globally as an option (-l) to make_dparser or locally with with a $declare longest_match .... If global longest match disambiguation is **ON**, it can be locally disabled with $declare all_matches ... . As with Tokenizers above, local declarations operate on disjoint subsets of parsing states.

Priorities and Associativity: Priorities can very from MININT to MAXINT and are specified as integers. Associativity can take the values:

```
assoc : '$unary_op_right' | '$unary_op_left' | '$binary_op_right'
| '$binary_op_left' | '$unary_right' | '$unary_left'
| '$binary_right' | '$binary_left' | '$right' | '$left' ;
```

- Token Prioritites

Termininal priorities apply after the set of matching strings has been found and the terminal(s) with the highest priority is selected.

Terminal priorities are introduced after a terminal by the specifier **$term**. We saw an example of token priorities with the definition of **ident**.

Example:

```
S : 'if' '(' S ')' S ';' | 'do' S 'while' '(' S ')' ';' | ident;
ident: "[a-z]+" $term -1;
```

- Operator Priorities

Operator priorities specify the priority of a operator symbol (either a terminal or a non-terminal). This corresponds to the yacc or bison doesn't require a global tokenizer, operator priorities and associativities are specified on the reduction which creates the token. Moreover, the associativity includes the operator usage as well since it cannot be infered from rule context. Possible operator associativies are:

```
operator_assoc : '$unary_op_right' | '$unary_op_left' | '$binary_op_right'
                   | '$binary_op_left' | '$unary_right' | '$unary_left'
                   | '$binary_right' | '$binary_left';
```

Example:

```
E: ident op ident;
ident: '[a-z]+';
op: '*' $binary_op_left 2 |
    '+' $binary_op_left 1;
```

- Rule Priorities

Rule priorities specify the priority of the reduction itself and have the possible associativies:

```
rule_assoc: '$right' | '$left';
```

Rule and operator priorities can be intermixed and are interpreted at run time (not when the tables are built). This make it possible for user-defined scanners to return the associativities and priorities of tokens.

Actions: Actions are the bits of code which run when a reduction occurs. Example

```
S: this | that;
this: 'this' { printf("got this\n"); };
that: 'that' { printf("got that\n"); };
```

- Speculative Action

Speculative actions occur when the reduction takes place during the speculative parsing process. It is possible that the reduction will not be part of the final parse or that it will occur a different number of times. For example:

```
S: this | that;
this: hi 'mom';
that: ho 'dad';
ho: 'hello' [ printf("ho\n"); ];
hi: 'hello' [ printf("hi\n"); ];
```

Will print both 'hi' and 'ho' when given the input 'hello dad' because at the time hello is reduced, the following token is not known.

- Final Actions

Final actions occur only when the reduction must be part of any legal final parse (committed). It is possible to do final actions during parsing or delay them till the entire parse tree is constructed (see Options). Final actions are executed in order and in number according the the single final unambiguous parse.

```
S: A S 'b' | 'x';
A: [ printf("speculative e-reduce A\n"); ]
   { printf("final e-reduce A\n"); };
```

On input:

```
xbbb
```

Will produce:

```
speculative e-reduce A
final e-reduce A
final e-reduce A
final e-reduce A
```

- Embedded Actions

Actions can be embedded into rule. These actions are executed as if they were replaced with a synthetic production with a single null rule containing the actions. For example:

```
S: A { printf("X"); } B;
A: 'a' { printf("a"); };
B: 'b' { printf("b"); };
```

On input:

`ab`

Will produce:

```
aXb
```

Note that in the above example, the print("X") is evaluated in a context null rule context while in:

```
S: A (A B { printf("X"); }) B;
```

The print is evalated in the context of the "A B" subrule because it appears at the end of the subrule and is therefor treated as a normal action for the subrule.

- Pass Actions

DParser supports multiple pass compilation. The passes are declared at the top of the grammar, and the actions are associated with individual rules.

Example;

```
${pass sym for_all postorder}
${pass gen for_all postorder}

translation_unit: statement*;

statement
  : expression ';' {
    d_pass(${parser}, &$n, ${pass sym});
    d_pass(${parser}, &$n, ${pass gen});
  }
  ;

expression :  integer
  gen: { printf("gen integer\n"); }
  sym: { printf("sym integer\n"); }
  | expression '+' expression $right 2
  sym: { printf("sym +\n"); }
  ;
```

A pass name then a colon indicate that the following action is associated with a particular pass. Passes can be either for_all or for_undefined (which means that the automatic traversal only applies to rules without actions defined for this pass). Furthermore, passes can be postorder, preorder, and manual (you have to call d_pass yourself). Passes can be initiated in the final action of any rule.

- Default Actions

The special production "_" can be defined with a single rule whose actions become the default when no other action is specified. Default actions can be specified for speculative, final and pass actions and apply to each separately.

Example

```
_: { printf("final action"); }
    gen: { printf("default gen action"); }
    sym: { printf("default sym action"); }
    ;
```

Attributes and Action Specifiers Each of the language parser can have some global atrributes and actions associated with each part of the parsed code.

- Global State ($g)

Global state is declared by define'ing D_ParseNode_Globals (see the ANSI C grammar for a similar declaration for symbols). Global state can be accessed in any action with $g. Because DParser handles ambiguous parsing global state can be accessed on different speculative parses. In the future automatic splitting of global state may be implemented (if there is demand). Currently, the global state can be copied and assigned to $g to ensure that the changes made only effect subsequent speculative parses derived from the particular parse.

Example

```
[ $g = copy_globals($g);
$g->my_variable = 1;
]
```

The symbol table (see below) can be used to manage state information safely for different speculative parses.

- Parse Node State

Each parse node includes a set of system state variables and can have a set of user-defined state variables. User defined parse node state is declared by define'ing D_ParseNodeUser. The size of the parse node state must be passed into new_D_Parser() to ensure that the appropriate amount of space is allocated for parse nodes. Parse node state is accessed with:

$# number of child nodes

$$ user parse node state for parent node (non-terminal defined by the production)

$X (where X is a number) the user parse node state of element X of the production

$n the system parse node state of the rule node

$nX the system parse node state of element X of the production

The system parse node state is defined in dparse.h which is installed with DParser. It contains such information as the symbol, the location of the parsed string, and pointers to the start and end of the parsed string.

- Misc

${scope} the current symbol table scope

${reject} in speculative actions permits the current parse to be rejected

Symbol Table The symbol table can be updated down different speculative paths while sharing the bulk of the data. It defines the following functions in the file (dsymtab.h):

```
  struct D_Scope *new_D_Scope(struct D_Scope *st);
 struct D_Scope *enter_D_Scope(struct D_Scope *current, struct D_Scope *scope);
  D_Sym *NEW_D_SYM(struct D_Scope *st, char *name, char *end);
  D_Sym *find_D_Sym(struct D_Scope *st, char *name, char *end);
  D_Sym *UPDATE_D_SYM(struct D_Scope *st, D_Sym *sym);
  D_Sym *current_D_Sym(struct D_Scope *st, D_Sym *sym);
  D_Sym *find_D_Sym_in_Scope(struct D_Scope *st, char *name, char *end);
```

'new_D_Scope' creates a new scope below 'st' or NULL for a 'top level' scope. 'enter_D_Scope' returns to a previous scoping level. NOTE: do not simply assign ${scope} to a previous scope as any updated symbol information will be lost. 'commit_D_Scope' can be used in final actions to compress the update list for the top level scope and improve efficiency.

'find_D_Sym' finds the most current version of a symbol in a given scope. 'UPDATE_D_SYM' updates the value of symbol (creates a difference record on the current speculative parse path). 'current_D_Sym' is used to retrive the current version of a symbol, the pointer to which may have been stored in some other attribute or variable. Symbols with the same name should not be created in the same scope. The function 'find_D_Sym_in_Scope' is provided to detect this case.

User data can be attached to symbols by **define**'ing **D_UserSym**. See the ANSI C grammar for an example.

Here is a full example of scope usage (from tests/g29.test.g):

```
  #include <stdio.h>

  typedef struct My_Sym {
    int value;
  } My_Sym;
  #define D_UserSym My_Sym
  typedef struct My_ParseNode {
    int value;
    struct D_Scope *scope;
  } My_ParseNode;
  #define D_ParseNode_User My_ParseNode
}

ranslation_unit: statement*;

tatement
: expression ';'
{ printf("
| '{' new_scope statement* '}'
[ ${scope} = enter_D_Scope(${scope}, $n0.scope); ]
{ ${scope} = commit_D_Scope(${scope}); }
;

ew_scope: [ ${scope} = new_D_Scope(${scope}); ];
```

```
xpression
: identifier ':' expression
[
_Sym *s;
f (find_D_Sym_in_Scope(${scope}, $n0.start_loc.s, $n0.end))
rintf("duplicate identifier line
 = NEW_D_SYM(${scope}, $n0.start_loc.s, $n0.end);
->user.value = $2.value;
$.value = s->user.value;
]
| identifier '=' expression
[ D_Sym *s = find_D_Sym(${scope}, $n0.start_loc.s, $n0.end);
 = UPDATE_D_SYM(${scope}, s);
->user.value = $2.value;
$.value = s->user.value;
]
| integer
[ $$.value = atoi($n0.start_loc.s); ]
| identifier
[ D_Sym *s = find_D_Sym(${scope}, $n0.start_loc.s, $n0.end);
f (s)
$.value = s->user.value;
]
| expression '+' expression
[ $$.value = $0.value + $1.value; ]
;

nteger: "-?([0-9]|0(x|X))[0-9]*(u|U|b|B|w|W|L|l)*" $term -1;
dentifier: "[a-zA-Z_][a-zA-Z_0-9]*";
```

Whitespace Whitespace can be specified two ways: C function which can be user-defined, or as a subgrammar. The default whitespace parser is compatible with C/C++ #line directives and comments. It can be replaced with any user specified function as a parsing option (see Options).

Additionally, if the (optionally) reserved production **whitespace** is defined, the subgrammar it defines will be used to consume whitespace for the main grammar. This subgrammar can include normal actions.

Example

```
: 'a' 'b' 'c';
hitespace: "[ \t\n]*";
```

Whitespace can be accessed on a per parse node basis using the unctions: **d_ws_before** and **d_ws_after**, which return the tart of the whitespace before start_loc.s and after end respectively.

Ambiguities Ambiguities are resolved automatically based on priorities and associativities. In addition, when the other resolution techniques fail, user defined ambiguity resolution is possible. The default ambiguity handler produces a fatal error on an unresolved ambiguity. This behavior can be replaced with a user defined resolvers the signature of which is provided in dparse.h.

If the **verbose_level** flag is set, the default ambiguity andler will print out parenthesized versions of the ambiguous parse rees. This may be of some assistence in disambiguating a grammar.

Error Recovery DParser implements an error recovery scheme appropriate to scannerless parsers. I haven't had time to investigate all the prior work in this area, so I am not sure if it is novel. Suffice for now that it is optional and works well with C/C++ like grammars.

Parsing Options Parser are instantiated with the function new_D_Parser. The resulting data structure contains a number of user configurable options (see dparser.h). These are provided reasonable default values and include:

- **initial_globals** - the initial global variables accessable through $g
- **initial_skip_space_fn** - the initial whitespace function
- **initial_scope** - the initial symbol table scope
- **syntax_error_fn** - the function called on a syntax error
- **ambiguity_fn** - the function called on an unresolved ambiguity
- **loc** - the initial location (set on an error).

In addtion, there are the following user configurables:

- **sizeof_user_parse_node** - the sizeof D_ParseNodeUser
- **save_parse_tree** - whether or not the parse tree should be save once the final actions have been executed
- **dont_fixup_internal_productions** - to not convert the Kleene star into a variable number of children from a tree of reductions
- **dont_merge_epsilon_trees** - to not automatically remove ambiguities which result from trees of epsilon reductions without actions
- **dont_use_greediness_for_disambiguation** - do not use the rule that the longest parse which reduces to the same token should be used to disambiguate parses. This rule is used to handle the case (if then else?) relatively cleanly.
- **dont_use_height_for_disambiguation** - do not use the rule that the least deep parse which reduces to the same token should be used to disabiguate parses. This rule is used to handle recursive grammars relatiively cleanly.
- **dont_compare_stacks** - disables comparing stacks to handle certain exponential cases during ambiguous operator priority resolution.
- **commit_actions_interval** - how often to commit final actions (0 is immediate, MAXINT is essentially not till the end of parsing)
- **error_recovery** - whether or not to use error recovery (defaults ON)

An the following result values:

- **syntax_errors** - how many syntax errors (if **error_recovery** was on)

This final value should be checked to see if parse was successful.

Author(s)

Maintainer: Matthew Fidler <`matthew.fidler@gmail.com`>

Authors:

- John Plevyak <`jplevyak@gmail.com`> [copyright holder] `dparse` *Create R-based Dparser tree walking function based on grammar*

Description

Note R-based dparser tree walking works on Windows (with R tools) Mac, or Linux. Linking to arbitrary c grammars works on any platform.

Usage

```
dparse(grammar, start_state = 0, save_parse_tree = TRUE,
  partial_parses = FALSE, compare_stacks = TRUE,
  commit_actions_interval = 100, fixup = TRUE, fixup_ebnf = FALSE,
  nogreedy = FALSE, noheight = FALSE, use_file_name = TRUE,
  parse_size = 1024, verbose_level = 0, children_first = TRUE, ...)
```

Arguments

`grammar`	Dparser grammar
`start_state`	Start State (default 0)
`save_parse_tree`	Save Parse Tree (default TRUE)
`partial_parses`	Partial Parses (default FALSE)
`compare_stacks`	Compare Stacks (default TRUE)
`commit_actions_interval`	Commit Interval (default 100)
`fixup`	Fix-up Internal Productions (default FALSE)
`fixup_ebnf`	Fixup EBNF Productions (default FALSE)
`nogreedy`	No Greediness for Disambiguation (default FALSE)
`noheight`	No Height for Disambiguation (default FALSE)
`use_file_name`	Use File Name for syntax errors (default TRUE)
`parse_size`	Parser size (default 1024)
`verbose_level`	the level of verbosity when creating parser (default 0)
`children_first`	When TRUE, parse the children before the parent (default TRUE).
`...`	Parameters sent to `mkdparse`, with the exception of `use_r_header` which is forced to be `TRUE`.

Value

A function that allows parsing of a file based on the grammar supplied. This function would be able to parse arbitrary grammars the way you may want with your own user supplied function.

Garbage collection

There are two user options that control if the dlls for the grammars created by dparser will be deleted upon garbage collection or R exit if they are not associated with any active objects. These are:

`dpaser.rm.unnamed.parser.dll`: when `TRUE`, this remove parsers that are created from strings, or other memory-based items in R.

`dpaser.rm.unnamed.parser.dll`: when `TRUE`, this removes parsers created from grammar files.

See Also

`mkdparse`

Examples

```
## This creates the R based parsing function.  It requires
## compilation and runs on most OSes, with the exception of solaris.
## Windows requires Rtools to be installed.
f <- dparse(system.file("tran.g", package = "dparser"),children_first=FALSE);

## Once created, you may then use this function to parse an
## arbitrary syntax file.  For example:
f("
b       = -1
d/dt(X) = a*X + Y*Z;
d/dt(Y) = b*(Y - Z);
d/dt(Z) = -X*Y + c*Y - Z
if (t < 0.02 | t > 99.98){
    print
}
", function(name, value, pos, depth){
    ## This prints the name, value, position and depth passed to the
    ##parsing function.
    cat(sprintf("name:%s;value:%s;pos:%s;depth:%s\n", name, value, pos,
                 depth));
})

## You could use a better R parsing function; You could also use
## this as a starting place for your own C-based parser
```

`dpDefaultSkip` *Default skip function for darsing grammar*

Description

This function is to determine if children are parsed or skipped. By default, all children are parsed.

Usage

```
dpDefaultSkip(name, value, pos, depth)
```

Arguments

`name`	Production Name
`value`	Production Value
`pos`	terminal position. Negative values are parsed before children are parsed. A value of -1 means there are no children nodes. A value of -2 means there are children nodes. Otherwise, the terminal position is numbered starting at 0.
`depth`	Parsing Depth

Value

FALSE. If a comparable function returns TRUE, the children are not parsed.

Author(s)

Matthew L. Fidler

`dpReload` *Reload the R dparser dll*

Description

This can be useful if parsing one grammar seems to affect another grammar.

Usage

```
dpReload()
```

Author(s)

Matthew L. Fidler

mkdparse *mkdparse dparser grammer c*

Description

Make a dparser c file based on a grammer

Usage

```
mkdparse(file, outputFile, set_op_priority_from_rule = FALSE,
  right_recursive_BNF = FALSE, states_for_whitespace = TRUE,
  states_for_all_nterms = FALSE, tokenizer = FALSE,
  token_type = c("#define", "enum"), longest_match = FALSE,
  grammar_ident = "gram", ident_from_filename = FALSE, scanner_blocks = 4,
  write_line_directives = TRUE, write_header = c("IfEmpty", TRUE, FALSE),
  rdebug = FALSE, verbose = FALSE, write_extension = "c",
  use_r_header = TRUE)
```

Arguments

`file`	File name of grammer to parse.
`outputFile`	Output file name. Can be a directory. If so, the file name is determined by the input file.
`set_op_priority_from_rule`	Toggle setting of operator priority from rules. Setting of operator priorities on operator tokens can increase the size of the tables but can permit unnecessary parse stacks to be pruned earlier. (FALSE by default)
`right_recursive_BNF`	Toggle use of right recursion for EBNF productions. Do not change this unless you really know what you are doing. (FALSE by default)
`states_for_whitespace`	Toggle computing whitespace states. If 'whitespace' is defined in the grammar, then use it as a subparser to consume whitespace. (TRUE by default)
`states_for_all_nterms`	Toggle computing states for all non-terminals. Ensures that there is a unique state for each non-terminal so that a subparsers can be invoked for that non-terminal. (FALSE by default)
`tokenizer`	Toggle building of a tokenizer for START. When TRUE, instead of generating a unique scanner for each state (i.e. a 'scannerless' parser), generate a single scanner (tokenizer) for the entire grammar. This provides an easy way to build grammars for languages which assume a tokenizer (e.g. ANSI C). (FALSE by default)
`token_type`	Token type "#define" or "enum"

`longest_match`	Toggle longest match lexical ambiguity resolution. When TRUE the scanner only recognizing the longest matching tokens in a given state. This provides an easy way to build grammars for languages which use longest match lexical ambiguity resolution (e.g. ANSI-C, C++). (FALSE by default)
`grammar_ident`	Tag for grammar data structures so that multiple sets of tables can be included in one file/application. (defaults to 'gram')
`ident_from_filename`	Build the grammer tag from the file-name.
`scanner_blocks`	Number of blocks to which scanner tables are broken up into. Larger numbers permit more sharing with more overhead. 4 seems to be optimal for most grammars. (defaults to 4) files.
`write_line_directives`	Toggle writing of line numbers. Used to debug the parsing table generator itself. (TRUE by default)
`write_header`	Write header, FALSE : no, TRUE : yes, "IfEmpty" : only if not empty.
`rdebug`	Replace all actions in the grammar with actions printing productions, 1 : during the speculative parsing process (<-), 2 : when reduction is part of any legal final parse (<=), 3 : both, 4 : remove all actions from the grammar. Print the changed grammar to console. Useful for debugging or prototyping new, experimental grammars.
`verbose`	Increase verbosity.
`write_extension`	Set the extension of the generated code file. For C++ programs (for example) the extension can be set to .cpp with the option `write_extension="cpp"`. (`write_extension="c"` by default)
`use_r_header`	when TRUE, add R headers and swap printf for Rprintf. By default this is TRUE.

Details

Uses a grammer file to create a c file for parsing.

mkdparser is a scannerless GLR parser generator based on the Tomita algorithm. It is self-hosted and very easy to use. Grammars are written in a natural style of EBNF and regular expressions and support both speculative and final actions.

Value

Nothing. Outputs files instead.

Creating a Grammar for Parsing

Grammar Comments: Grammars can include C/C++ style comments

Example:

```
// My first grammar
E: E '+' E | "[abc]";
/* is this right? */
```

Grammar Productions: A production is the parts of your language your are trying to parse and are tpyically named. See https://en.wikipedia.org/wiki/Top-down_parsing

- The first production is the root of your grammar (what you will be trying to parse).
- Productions start with the non-terminal being defined followed by a colon ':', a set of right hand sides separated by '|' (or) consisting of elements (non-terminals or terminals).
- Elements can be grouped with parens '(', and the normal regular expression symbols can be used ('+' '*' '?' '|').
- Elements can be grouped with parens '(', and the normal regular expression symbols can be used ('+' '*' '?' '|').
- Elements can be repeated using '@', for example elem@3 or elem@1:3 for repeat 3 or between 1 and 3 times respectively.

Example:

```
program: statements+ |  comment* (function |  procedure)?;
```

Note: Instead of using '[' ']' for optional elements we use the more familar and consistent '?' operator. The square brackets are reserved for speculative actions (below).

Global C code in Grammars: Since the main parsing of the language grammar is in C, intermixing C code with the grammar can be useful.

- Global (or static) C code can be intermixed with productions by surrounding the code with brackets '{}'.

Example:

```
  { void dr_s() { printf("Dr. S\n"); }
   S: 'the' 'cat' 'and' 'the' 'hat' { dr_s(); } | T;
   { void twain() { printf("Mark Twain\n"); }
       T: 'Huck' 'Finn' { twain(); };
```

Note: When parsing the grammar using `mkdparse`, the option `use_r_header = TRUE` will redefine `printf` to `Rprintf` to better comply with R packages.

Terminals The terminals are the peices of the language that are being parsed, like language keywords.

- Strings terminals are surrounded with single quotes. For example:

```
block: '{' statements* '}';
whileblock: 'while' '(' expression ')' block;
```

- Unicode literals can appear in strings or as charaters with U+ or u+. For example:

```
U+03c9 { printf("omega\n"); }
```

- Regular expressions are surrounded with double quotes. For example:

```
hexint: "(0x|0X)[0-9a-fA-F]+[uUlL]?";
```

Note: only the simple regular expression operators are currently supported. This include parens, square parens, ranges, and '*', '+', '?'.

- Terminal modifiers

Terminals can contain embbed escape codes. Including the standard C escape codes, the codes \x and \d permit inserting hex and decimal ASCII characters directly.

Tokens can be given a name by appending the $name option. This is useful when you have several tokens which which represent the same string (e.g. ’,’). For example,

```
function_call: function '(' parameter ( ',' $name 'parameter_comma' parameter) ')';
```

It is now possible to use $0.symbol == $string parameter_comma to differentiate ParseNode ($0) between a parameter comma node and say an initialization comma.

Terminals ending in ’/i’ are case insensitive. For example ’hi’/i matches ’HI’, ’Hi’ and "hI’ in addition to ’hi’.

- External (C) Scanners

There are two types of external scanners, those which read a single terminal, and those which are global (called for every terminal). Here is an example of a scanner for a single terminal. Notice how it can be mixed with regular string terminals.

```
{
    extern char *ops;
    extern void *ops_cache;
    int ops_scan(char *ops, void *ops_cache, char **as,
              int *col, int *line, unsigned short *op_assoc, int *op_priority);
}

X: '1' (${scan ops_scan(ops, ops_cache)} '2')*;
```

The user provides the ’ops_scan’ function. This example is from tests/g4.test.g in the source distribution.

The second type of scanner is a global scanner:

```
{
   #include "g7.test.g.d_parser.h"
   int myscanner(char **s, int *col, int *line, unsigned short *symbol,
              int *term_priority, unsigned short *op_assoc, int *op_priority)
   {
       if (**s == 'a') {
           (*s)++;
           *symbol = A;
           return 1;
       } else if (**s == 'b') {
           (*s)++;
           *symbol = BB;
           return 1;
       } else if (**s == 'c') {
           (*s)++;
           *symbol = CCC;
           return 1;
       } else if (**s == 'd') {
           (*s)++;
```

```
            *symbol = DDDD;
            return 1;
        } else
            return 0;
    }
    ${scanner myscanner}
    ${token A BB CCC DDDD}

    S: A (BB CCC)+ SS;
    SS: DDDD;
```

Notice how the you need to include the header file generated by `mkdparse` which contains the token definitions.

- Tokenizers

Tokenizers are non-context sensitive global scanners which produce only one token for any given input string. Some programming languages (for example C) are easier to specify using a tokenizer because (for example) reserved words can be handled simply by lowering the terminal priority for identifiers.

```
S : 'if' '(' S ')' S ';' | 'do' S 'while' '(' S ')' ';' | ident;
ident: "[a-z]+" $term -1;
```

The sentence: **if (while) a;** is legal because **while** cannot appear at the start of **S** and so it doesn't conflict with the parsing of **while** as an **ident** in that position. However, if a tokenizer is specified, all tokens will be possible at each position and the sentense will produce a syntax error.

DParser provides two ways to specify tokenizers: globally as an option (-T) to `mkdparse` and locally with a ${declare tokenize ...} specifier (see the ANSI C grammar for an example). The ${declare tokenize ...} declartion allows a tokenizer to be specified over a subset of the parsing states so that (for example) ANSI C could be a subgrammar of another larger grammar. Currently the parse states are not split so that the productions for the substates must be disjoint.

- Longest Match

Longest match lexical ambiguity resolution is a technique used by separate phase lexers to help decide (along with lexical priorities) which single token to select for a given input string. It is used in the definition of ANSI-C, but not in C++ because of a snafu in the definition of templates whereby templates of templates (List<List <Int») can end with the right shift token ('»'). Since **DParser** does not have a separate lexical phase, it does not require longest match disambiguation, but provides it as an option.

There are two ways to specify longest match disabiguation: globally as an option (-l) to make_dparser or locally with with a $declare longest_match .... If global longest match disambiguation is **ON**, it can be locally disabled with $declare all_matches ... . As with Tokenizers above, local declarations operate on disjoint subsets of parsing states.

Priorities and Associativity: Priorities can very from MININT to MAXINT and are specified as integers. Associativity can take the values:

```
assoc : '$unary_op_right' | '$unary_op_left' | '$binary_op_right'
| '$binary_op_left' | '$unary_right' | '$unary_left'
| '$binary_right' | '$binary_left' | '$right' | '$left' ;
```

- Token Prioritites

Termininal priorities apply after the set of matching strings has been found and the terminal(s) with the highest priority is selected.

Terminal priorities are introduced after a terminal by the specifier **$term**. We saw an example of token priorities with the definition of **ident**.

Example:

```
S : 'if' '(' S ')' S ';' | 'do' S 'while' '(' S ')' ';' | ident;
ident: "[a-z]+" $term -1;
```

- Operator Priorities

Operator priorities specify the priority of a operator symbol (either a terminal or a non-terminal). This corresponds to the yacc or bison doesn't require a global tokenizer, operator priorities and associativities are specified on the reduction which creates the token. Moreover, the associativity includes the operator usage as well since it cannot be infered from rule context. Possible operator associativies are:

```
operator_assoc : '$unary_op_right' | '$unary_op_left' | '$binary_op_right'
                   | '$binary_op_left' | '$unary_right' | '$unary_left'
                   | '$binary_right' | '$binary_left';
```

Example:

```
E: ident op ident;
ident: '[a-z]+';
op: '*' $binary_op_left 2 |
    '+' $binary_op_left 1;
```

- Rule Priorities

Rule priorities specify the priority of the reduction itself and have the possible associativies:

```
rule_assoc: '$right' | '$left';
```

Rule and operator priorities can be intermixed and are interpreted at run time (not when the tables are built). This make it possible for user-defined scanners to return the associativities and priorities of tokens.

Actions: Actions are the bits of code which run when a reduction occurs. Example

```
S: this | that;
this: 'this' { printf("got this\n"); };
that: 'that' { printf("got that\n"); };
```

- Speculative Action

Speculative actions occur when the reduction takes place during the speculative parsing process. It is possible that the reduction will not be part of the final parse or that it will occur a different number of times. For example:

```
S: this | that;
this: hi 'mom';
that: ho 'dad';
ho: 'hello' [ printf("ho\n"); ];
hi: 'hello' [ printf("hi\n"); ];
```

Will print both 'hi' and 'ho' when given the input 'hello dad' because at the time hello is reduced, the following token is not known.

- Final Actions

Final actions occur only when the reduction must be part of any legal final parse (committed). It is possible to do final actions during parsing or delay them till the entire parse tree is constructed (see Options). Final actions are executed in order and in number according the the single final unambiguous parse.

```
S: A S 'b' | 'x';
A: [ printf("speculative e-reduce A\n"); ]
   { printf("final e-reduce A\n"); };
```

On input:

```
xbbb
```

Will produce:

```
speculative e-reduce A
final e-reduce A
final e-reduce A
final e-reduce A
```

- Embedded Actions

Actions can be embedded into rule. These actions are executed as if they were replaced with a synthetic production with a single null rule containing the actions. For example:

```
S: A { printf("X"); } B;
A: 'a' { printf("a"); };
B: 'b' { printf("b"); };
```

On input:

```
ab
```

Will produce:

```
aXb
```

Note that in the above example, the print("X") is evaluated in a context null rule context while in:

```
S: A (A B { printf("X"); }) B;
```

The print is evalated in the context of the "A B" subrule because it appears at the end of the subrule and is therefor treated as a normal action for the subrule.

- Pass Actions

DParser supports multiple pass compilation. The passes are declared at the top of the grammar, and the actions are associated with individual rules.

Example;

```
${pass sym for_all postorder}
${pass gen for_all postorder}

translation_unit: statement*;

statement
  : expression ';' {
    d_pass(${parser}, &$n, ${pass sym});
    d_pass(${parser}, &$n, ${pass gen});
  }
  ;

expression :  integer
  gen: { printf("gen integer\n"); }
  sym: { printf("sym integer\n"); }
  | expression '+' expression $right 2
  sym: { printf("sym +\n"); }
  ;
```

A pass name then a colon indicate that the following action is associated with a particular pass. Passes can be either for_all or for_undefined (which means that the automatic traversal only applies to rules without actions defined for this pass). Furthermore, passes can be postorder, preorder, and manual (you have to call d_pass yourself). Passes can be initiated in the final action of any rule.

- Default Actions

The special production "_" can be defined with a single rule whose actions become the default when no other action is specified. Default actions can be specified for speculative, final and pass actions and apply to each separately.

Example

```
_: { printf("final action"); }
    gen: { printf("default gen action"); }
    sym: { printf("default sym action"); }
    ;
```

Attributes and Action Specifiers Each of the language parser can have some global atrributes and actions associated with each part of the parsed code.

- Global State ($g)

Global state is declared by define'ing D_ParseNode_Globals (see the ANSI C grammar for a similar declaration for symbols). Global state can be accessed in any action with $g. Because DParser handles ambiguous parsing global state can be accessed on different speculative parses. In the future automatic splitting of global state may be implemented (if there is demand). Currently, the global state can be copied and assigned to $g to ensure that the changes made only effect subsequent speculative parses derived from the particular parse.

Example

```
[ $g = copy_globals($g);
$g->my_variable = 1;
]
```

The symbol table (see below) can be used to manage state information safely for different speculative parses.

- Parse Node State

Each parse node includes a set of system state variables and can have a set of user-defined state variables. User defined parse node state is declared by define'ing D_ParseNodeUser. The size of the parse node state must be passed into new_D_Parser() to ensure that the appropriate amount of space is allocated for parse nodes. Parse node state is accessed with:

$# number of child nodes

$$ user parse node state for parent node (non-terminal defined by the production)

$X (where X is a number) the user parse node state of element X of the production

$n the system parse node state of the rule node

$nX the system parse node state of element X of the production

The system parse node state is defined in dparse.h which is installed with DParser. It contains such information as the symbol, the location of the parsed string, and pointers to the start and end of the parsed string.

- Misc

${scope} the current symbol table scope

${reject} in speculative actions permits the current parse to be rejected

Symbol Table The symbol table can be updated down different speculative paths while sharing the bulk of the data. It defines the following functions in the file (dsymtab.h):

```
 struct D_Scope *new_D_Scope(struct D_Scope *st);
struct D_Scope *enter_D_Scope(struct D_Scope *current, struct D_Scope *scope);
 D_Sym *NEW_D_SYM(struct D_Scope *st, char *name, char *end);
 D_Sym *find_D_Sym(struct D_Scope *st, char *name, char *end);
 D_Sym *UPDATE_D_SYM(struct D_Scope *st, D_Sym *sym);
 D_Sym *current_D_Sym(struct D_Scope *st, D_Sym *sym);
 D_Sym *find_D_Sym_in_Scope(struct D_Scope *st, char *name, char *end);
```

'new_D_Scope' creates a new scope below 'st' or NULL for a 'top level' scope. 'enter_D_Scope' returns to a previous scoping level. NOTE: do not simply assign ${scope} to a previous scope as any updated symbol information will be lost. 'commit_D_Scope' can be used in final actions to compress the update list for the top level scope and improve efficiency.

'find_D_Sym' finds the most current version of a symbol in a given scope. 'UPDATE_D_SYM' updates the value of symbol (creates a difference record on the current speculative parse path). 'current_D_Sym' is used to retrive the current version of a symbol, the pointer to which may have been stored in some other attribute or variable. Symbols with the same name should not be created in the same scope. The function 'find_D_Sym_in_Scope' is provided to detect this case.

User data can be attached to symbols by **define**'ing **D_UserSym**. See the ANSI C grammar for an example.

Here is a full example of scope usage (from tests/g29.test.g):

```
#include <stdio.h>
```

```
  typedef struct My_Sym {
    int value;
  } My_Sym;
  #define D_UserSym My_Sym
  typedef struct My_ParseNode {
    int value;
    struct D_Scope *scope;
  } My_ParseNode;
  #define D_ParseNode_User My_ParseNode
}

ranslation_unit: statement*;

tatement
: expression ';'
{ printf("
| '{' new_scope statement* '}'
[ ${scope} = enter_D_Scope(${scope}, $n0.scope); ]
{ ${scope} = commit_D_Scope(${scope}); }
;

ew_scope: [ ${scope} = new_D_Scope(${scope}); ];

xpression
: identifier ':' expression
[
_Sym *s;
f (find_D_Sym_in_Scope(${scope}, $n0.start_loc.s, $n0.end))
rintf("duplicate identifier line
 = NEW_D_SYM(${scope}, $n0.start_loc.s, $n0.end);
->user.value = $2.value;
$.value = s->user.value;
]
| identifier '=' expression
[ D_Sym *s = find_D_Sym(${scope}, $n0.start_loc.s, $n0.end);
 = UPDATE_D_SYM(${scope}, s);
->user.value = $2.value;
$.value = s->user.value;
]
| integer
[ $$.value = atoi($n0.start_loc.s); ]
| identifier
[ D_Sym *s = find_D_Sym(${scope}, $n0.start_loc.s, $n0.end);
f (s)
$.value = s->user.value;
]
| expression '+' expression
```

```
[ $$.value = $0.value + $1.value; ]
;

nteger: "-?([0-9]|0(x|X))[0-9]*(u|U|b|B|w|W|L|l)*" $term -1;
dentifier: "[a-zA-Z_][a-zA-Z_0-9]*";
```

Whitespace Whitespace can be specified two ways: C function which can be user-defined, or as a subgrammar. The default whitespace parser is compatible with C/C++ #line directives and comments. It can be replaced with any user specified function as a parsing option (see Options).

Additionally, if the (optionally) reserved production **whitespace** is defined, the subgrammar it defines will be used to consume whitespace for the main grammar. This subgrammar can include normal actions.

Example

```
: 'a' 'b' 'c';
hitespace: "[ \t\n]*";
```

Whitespace can be accessed on a per parse node basis using the unctions: **d_ws_before** and **d_ws_after**, which return the tart of the whitespace before start_loc.s and after end respectively.

Ambiguities Ambiguities are resolved automatically based on priorities and associativities. In addition, when the other resolution techniques fail, user defined ambiguity resolution is possible. The default ambiguity handler produces a fatal error on an unresolved ambiguity. This behavior can be replaced with a user defined resolvers the signature of which is provided in dparse.h.

If the **verbose_level** flag is set, the default ambiguity andler will print out parenthesized versions of the ambiguous parse rees. This may be of some assistence in disambiguating a grammar.

Error Recovery DParser implements an error recovery scheme appropriate to scannerless parsers. I haven't had time to investigate all the prior work in this area, so I am not sure if it is novel. Suffice for now that it is optional and works well with C/C++ like grammars.

Parsing Options Parser are instantiated with the function new_D_Parser. The resulting data structure contains a number of user configurable options (see dparser.h). These are provided reasonable default values and include:

- **initial_globals** - the initial global variables accessable through $g
- **initial_skip_space_fn** - the initial whitespace function
- **initial_scope** - the initial symbol table scope
- **syntax_error_fn** - the function called on a syntax error
- **ambiguity_fn** - the function called on an unresolved ambiguity
- **loc** - the initial location (set on an error).

In addtion, there are the following user configurables:

- **sizeof_user_parse_node** - the sizeof D_ParseNodeUser
- **save_parse_tree** - whether or not the parse tree should be save once the final actions have been executed
- **dont_fixup_internal_productions** - to not convert the Kleene star into a variable number of children from a tree of reductions

- **dont_merge_epsilon_trees** - to not automatically remove ambiguities which result from trees of epsilon reductions without actions
- **dont_use_greediness_for_disambiguation** - do not use the rule that the longest parse which reduces to the same token should be used to disambiguate parses. This rule is used to handle the case (if then else?) relatively cleanly.
- **dont_use_height_for_disambiguation** - do not use the rule that the least deep parse which reduces to the same token should be used to disabiguate parses. This rule is used to handle recursive grammars relatiively cleanly.
- **dont_compare_stacks** - disables comparing stacks to handle certain exponential cases during ambiguous operator priority resolution.
- **commit_actions_interval** - how often to commit final actions (0 is immediate, MAXINT is essentially not till the end of parsing)
- **error_recovery** - whether or not to use error recovery (defaults ON)

An the following result values:

- **syntax_errors** - how many syntax errors (if **error_recovery** was on)

This final value should be checked to see if parse was successful.

Author(s)

Matthew L. Fidler for R interface, John Plevyak for dparser

Examples

```
## This makes the ANSI c grammar file to parse C code:
mkdparse(system.file("ansic.test.g", package = "dparser"),"ansic_gram.c", grammar_ident="ascii_C");
```

Index