Recursive descent parser

From Wikipedia, the free encyclopedia

In computer science, a **recursive descent parser** is a kind of top-down parser built from a set of mutually recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the productions of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.[1]

A *predictive parser* is a recursive descent parser that does not require backtracking. Predictive parsing is possible only for the class of LL(*k*) grammars, which are the context-free grammars for which there exists some positive integer *k* that allows a recursive descent parser to decide which production to use by examining only the next *k* tokens of input. The LL(*k*) grammars therefore exclude all ambiguous grammars, as well as all grammars that contain left recursion. Any context-free grammar can be transformed into an equivalent grammar that has no left recursion, but removal of left recursion does not always yield an LL(*k*) grammar. A predictive parser runs in linear time.

Recursive descent with backtracking is a technique that determines which production to use by trying each production in turn. Recursive descent with backtracking is not limited to LL(k) grammars, but is not guaranteed to terminate unless the grammar is LL(k). Even when they terminate, parsers that use recursive descent with backtracking may require exponential time.

Although predictive parsers are widely used, and are frequently chosen if writing a parser by hand, programmers often prefer to use a table-based parser produced by a parser generator, either for an LL(*k*) language or using an alternative parser, such as LALR or LR. This is particularly the case if a grammar is not in LL(*k*) form, as transforming the grammar to LL to make it suitable for predictive parsing is involved. Predictive parsers can also be automatically generated, using tools like ANTLR.

Predictive parsers can be depicted using transition diagrams for each non-terminal symbol where the edges between the initial and the final states are labelled by the symbols (terminals and non-terminals) of the right side of the production rule.[2]

Contents [hide]

1 Example parser
  1.1 C implementation
2 See also
3 References
4 External links

Example parser [edit]

The following EBNF-like grammar (for Niklaus Wirth's PL/0 programming language, from *Algorithms + Data Structures = Programs*) is in LL(1) form:

```
program = block "." .

block =
    ["const" ident "=" number {"," ident "=" number} ";"]
    ["var" ident {"," ident} ";"]
    {"procedure" ident ";" block ";"} statement .

statement =
    ident ":=" expression
    | "call" ident
    | "begin" statement {";" statement } "end"
    | "if" condition "then" statement
```

Wikipedia The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Čeština
Deutsch
한국어
日本語
Português
Русский
Српски / srpski
Українська
Edit links

```
    | "while" condition "do" statement .

condition =
    "odd" expression
    | expression ("="|"#"|"<"|"<="|">"|">=") expression .

expression = ["+"|"-"] term {("+"|"-") term} .

term = factor {("*"|"/") factor} .

factor =
    ident
    | number
    | "(" expression ")" .
```

Terminals are expressed in quotes. Each nonterminal is defined by a rule in the grammar, except for *ident* and *number*, which are assumed to be implicitly defined.

C implementation [edit]

What follows is an implementation of a recursive descent parser for the above language in C. The parser reads in source code, and exits with an error message if the code fails to parse, exiting silently if the code parses correctly.

Notice how closely the predictive parser below mirrors the grammar above. There is a procedure for each nonterminal in the grammar. Parsing descends in a top-down manner, until the final nonterminal has been processed. The program fragment depends on a global variable, *sym*, which contains the current symbol from the input, and the function *nextsym*, which updates *sym* when called.

The implementations of the functions *nextsym* and *error* are omitted for simplicity.

```
typedef enum {ident, number, lparen, rparen, times, slash, plus,
    minus, eql, neq, lss, leq, gtr, geq, callsym, beginsym, semicolon,
    endsym, ifsym, whilesym, becomes, thensym, dosym, constsym, comma,
    varsym, procsym, period, oddsym} Symbol;

Symbol sym;
void nextsym(void);
void error(const char msg[]);

int accept(Symbol s) {
    if (sym == s) {
        nextsym();
        return 1;
    }
    return 0;
}

int expect(Symbol s) {
    if (accept(s))
        return 1;
    error("expect: unexpected symbol");
    return 0;
}

void factor(void) {
    if (accept(ident)) {
        ;
    } else if (accept(number)) {
        ;
    } else if (accept(lparen)) {
        expression();
        expect(rparen);
    } else {
        error("factor: syntax error");
        nextsym();
    }
}

void term(void) {
```

```
    factor();
    while (sym == times || sym == slash) {
        nextsym();
        factor();
    }
}

void expression(void) {
    if (sym == plus || sym == minus)
        nextsym();
    term();
    while (sym == plus || sym == minus) {
        nextsym();
        term();
    }
}

void condition(void) {
    if (accept(oddsym)) {
        expression();
    } else {
        expression();
        if (sym == eql || sym == neq || sym == lss || sym == leq || sym == gtr ||
sym == geq) {
            nextsym();
            expression();
        } else {
            error("condition: invalid operator");
            nextsym();
        }
    }
}

void statement(void) {
    if (accept(ident)) {
        expect(becomes);
        expression();
    } else if (accept(callsym)) {
        expect(ident);
    } else if (accept(beginsym)) {
        do {
            statement();
        } while (accept(semicolon));
        expect(endsym);
    } else if (accept(ifsym)) {
        condition();
        expect(thensym);
        statement();
    } else if (accept(whilesym)) {
        condition();
        expect(dosym);
        statement();
    } else {
        error("statement: syntax error");
        nextsym();
    }
}

void block(void) {
    if (accept(constsym)) {
        do {
            expect(ident);
            expect(eql);
            expect(number);
        } while (accept(comma));
        expect(semicolon);
    }
    if (accept(varsym)) {
        do {
            expect(ident);
        } while (accept(comma));
        expect(semicolon);
```

```
    }
    while (accept(procsym)) {
        expect(ident);
        expect(semicolon);
        block();
        expect(semicolon);
    }
    statement();
}

void program(void) {
    nextsym();
    block();
    expect(period);
}
```

See also [edit]

Computer science portal

- JavaCC – a recursive descent parser generator
- Coco/R – a recursive descent parser generator
- ANTLR – a recursive descent parser generator
- Parsing expression grammar – another form representing recursive descent grammar
- Spirit Parser Framework – a C++ recursive descent parser generator framework requiring no pre-compile step
- Tail recursive parser – a variant of the recursive descent parser
- parboiled (Java) – a recursive descent PEG parsing library for Java
- Recursive ascent parser
- bnf2xml Markup input with XML tags using advanced BNF matching. (a top town LL recursive parser, front to back text, no compiling of lexor is needed or used)
- Parse::RecDescent: A versatile recursive descent Perl module.
- pyparsing: A versatile Python recursive parsing module that is not recursive descent (python-list post).
- Jparsec a Java port of Haskell's Parsec module.

References [edit]

1. ^ Burge, W.H. (1975). *Recursive Programming Techniques*. ISBN 0-201-14450-6.
2. ^ Aho, Alfred V.; Sethi, Ravi; Ullman, Jeffrey (1986). *Compilers: Principles, Techniques and Tools* (first ed.). Addison Wesley. p. 183.

This article is based on material taken from the Free On-line Dictionary of Computing prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

- *Compilers: Principles, Techniques, and Tools*, first edition, Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman, in particular Section 4.4.
- *Modern Compiler Implementation in Java, Second Edition*, Andrew Appel, 2002, ISBN 0-521-82060-X.
- *Recursive Programming Techniques*, W.H. Burge, 1975, ISBN 0-201-14450-6
- *Crafting a Compiler with C*, Charles N Fischer and Richard J LeBlanc, Jr, 1991, ISBN 0-8053-2166-7.
- *Compiling with C# and Java*, Pat Terry, 2005, ISBN 0-321-26360-X, 624
- *Algorithms + Data Structures = Programs*, Niklaus Wirth, 1975, ISBN 0-13-022418-9
- *Compiler Construction*, Niklaus Wirth, 1996, ISBN 0-201-40353-6

External links [edit]

- Introduction to Parsing - an easy to read introduction to parsing, with a comprehensive section on recursive descent parsing
- How to turn a Grammar into C code - a brief tutorial on implementing recursive descent parser
- Simple mathematical expressions parser in Ruby
- Simple Top Down Parsing in Python
- Jack W. Crenshaw: *Let's Build A Compiler* (1988-1995), in Pascal, with assembly language output, using a "keep it simple" approach
- Functional Pearls: Monadic Parsing in Haskell

Categories: Parsing algorithms