The *Yapps* Parser Generator System

`http://theory.stanford.edu/~amitp/Yapps/`
Version 1.1

Amit J. Patel
http://www-cs-students.stanford.edu/ amitp/

1 Introduction

Yapps (Yet Another Python Parser System) is an easy to use parser generator that is written in Python and generates Python code. There are several parser generator systems already available for Python, including `PyLR`, `kjParsing`, `PyBison`, and `mcf.pars`, but I had different goals for my parser. Yapps is simple, is easy to use, and produces human-readable parsers. It is not fast, powerful, or particularly flexible. Yapps is designed to be used when regular expressions are not enough and other parser systems are too much: situations where you may write your own recursive descent parser.

Some unusual features of Yapps that may be of interest are:

1. Yapps produces recursive descent parsers that are readable by humans, as opposed to table-driven parsers that are difficult to read. A Yapps parser for a simple calculator looks similar to the one that Mark Lutz wrote by hand for *Programming Python.*

2. Yapps also allows for rules that accept parameters and pass arguments to be used while parsing subexpressions. Grammars that allow for arguments to be passed to subrules and for values to be passed back are often called *attribute grammars.* In many cases parameterized rules can be used to perform actions at "parse time" that are usually delayed until later. For example, information about variable declarations can be passed into the rules that parse a procedure body, so that undefined variables can be detected at parse time.

3. Yapps grammars are fairly easy to write, although there are some inconveniences having to do with LL(1) parsing that have to be worked around. For example, rules have to be left factored and rules may not be left recursive.

4. Yapps itself is written as a single Python module with no C extensions. Yapps produces parsers that are written entirely in Python, and can be self-contained. Options are provided to control how much code is imported from the Yapps module and how much is self-contained.

5. Yapps's scanner is context-sensitive, picking tokens based on the types of the tokens accepted by the parser. This can be helpful when implementing certain kinds of parsers, such as for a preprocessor.

There are several disadvantages of using Yapps over another parser system:

1. Yapps parsers are `LL(1)`, which is less powerful than `LALR` (used by `PyLR`) or `SLR` (used by `kjParsing`), so Yapps would not be a good choice for parsing complex languages.

2. The scanner that Yapps provides can only read from strings, not files, so an entire file has to be read in before scanning can begin. It is possible to build a custom scanner, though, so in cases where stream input is needed (from the console, a network, or a large file are examples), the Yapps parser can be given a custom scanner that reads from a stream instead of a string.

3. Yapps is not designed with efficiency in mind.

Yapps provides an easy to use parser generator that produces parsers similar to what you might write by hand. It is not meant to be a solution for all parsing problems, but instead an aid for those times you would write a parser by hand rather than using one of the more powerful parsing packages available.

2 Examples

In this section are several examples that show the use of Yapps. First, an introduction shows how to construct grammars and write them in Yapps form. This example can be skipped by someone familiar with grammars and parsing. Next is a Lisp expression grammar that produces a parse tree as output. This example demonstrates the use of tokens and rules, as well as returning values from rules. The third example is a calculator grammar that evaluates expressions during the process of parsing instead of producing a parse tree. This example also shows the parameterization of rules. The examples in this document use old-style (Python 1.4) regular expressions, except where noted. Yapps can also be used with new-style (Python 1.5) regular expressions; see section 5.5 for details.

2.1 Introduction to Grammars

A *grammar* for a natural language specifies how words can be put together to form large structures, such as phrases and sentences. A grammar for a computer language is similar in that it specifies how small components (called *tokens*) can be put together to form larger structures. In this section we will write a grammar for a tiny subset of English.

Simple English sentences can be described as being a noun phrase followed by a verb followed by a noun phrase. For example, in the sentence, "Jack sank the blue ship," the word "Jack" is the first noun phrase, "sank" is the verb, and "the blue ship" is the second noun phrase. In addition we should say what a noun phrase is; for this example we shall say that a noun phrase is an optional article (a, an, the) followed by any number of adjectives followed by a noun. The tokens in our language are the articles, nouns, verbs, and adjectives. The *rules* in our language will tell us how to combine the tokens together to form lists of adjectives, noun phrases, and sentences:

- `sentence: noun_phrase verb noun_phrase`
- `noun_phrase: opt_article adjective_list noun`
- `opt_article: article` *or* (nothing)
- `adjective_list: adjective adjective_list` *or* (nothing)

Notice that some things that we said easily in English, such as "optional article," are expressed by a separate rule. When we said, "any number of adjectives," we made a rule `adjective_list` that made it explicit: an adjective list is an adjective followed by a list of adjectives, or it's nothing at all.

A minor note about recursive descent parsers, like those that Yapps produces: the parser looks from left to right, so the rules have to be specified so that it can determine which alternative to use based on what's on the left. This is why we wrote that an adjective list was `adjective adjective_list` rather than `adjective_list adjective`. It is easy for Yapps to see that something starts with an adjective on the left, but it is much more difficult to determine whether it starts with an adjective list.

The grammar given above is close to a Yapps grammar. We also have to specify what the tokens are, and what to return every time a rule is matched. We'll just use ... for the return value, since they aren't important to this example. Return values (`-> <<...>>`) will be described in the next example.

```
parser TinyEnglish:
  ignore:          "[ \t\n\r]+"
  token noun:      "\\(Jack\\|spam\\|ship\\)"
  token verb:      "\\(sank\\|threw\\)"
  token article:   "\\(a\\|an\\|the\\)"
  token adjective: "\\(blue\\|red\\|green\\)"

  rule sentence:       noun_phrase verb noun_phrase     ->  <<...>>
  rule noun_phrase:    opt_article adjective_list noun  ->  <<...>>
  rule opt_article:    article                          ->  <<...>>
                     |                                  ->  <<...>>
  rule adjective_list: adjective adjective_list         ->  <<...>>
                     |                                  ->  <<...>>
```

The tokens are specified as Python *regular expressions*. Since Yapps produces Python code, you can write any regular expression that would be accepted by Python, and you should use extra backslashes when necessary. (*Note:* By default, Yapps uses old-style regular expressions; see section 5.5 if you want to use Python 1.5 style regular expressions.) In addition to tokens that you want to see (which are given names), you can also specify tokens to ignore, marked by the `ignore` keyword. In this parser we want to ignore whitespace.

The TinyEnglish grammar shows how you define tokens and rules, but it does not specify what should happen once we've matched the rules. In the next example, we will take a grammar and produce a *parse tree* from it.

2.2 Lisp Expressions

Lisp syntax, although hated by many, has a redeeming quality: it is simple to parse. In this section we will construct a Yapps grammar to parse Lisp expressions and produce a parse tree as output.

Defining the Grammar

The syntax of Lisp is simple. It has expressions, which are identifiers, strings, numbers, and lists. A list is a left parenthesis followed by some number of expressions (separated by spaces) followed by a right parenthesis. For example, `5`, `"ni"`, and `(print "1+2 = " (+ 1 2))` are Lisp expressions. Written as a grammar,

```
expr:   ID | STR | NUM | list
list:   ( seq )
seq:       | expr seq
```

In addition to having a grammar, we need to specify what value to return every time something is matched. For the tokens, which are strings, we just want to get the "value" of the token, attach its type (identifier, string, or number) in some way, and return it. For the lists, we want to construct and return a Python list.

The value to return is a Python expression enclosed in `<<...>>`. Within this expression, we can refer to the values returned by matching each part of the rule. In matching `expr seq`, we can refer to "expr" as the value returned by matching the `expr` part and to "seq" as the value returned by matching the `seq` part. Let's take a look at the rule:

```
    rule seq:                   ->  << [] >>
              | expr seq        ->  << [expr] + seq >>
```

If the sequence is empty, then we'll return an empty list. If it's not empty, then it's an expression followed by the rest of the sequence. The `seq` rule will return a list, so we can produce a new list by adding the one-element list `[expr]` to `seq`.

In a rule, tokens return the text that was matched. We may need to convert this text to a different form. For example, if a string is matched, we want to remove quotes and handle special forms like `\n`. If a number is matched, we want to convert it into a number. Also, for the Lisp parser we are going to give them type tags. Let's look at the return values for each of the tokens:

```
    rule expr:   ID             ->  << ('id',ID) >>
              | STR             ->  << ('str',eval(STR)) >>
              | NUM             ->  << ('num',atoi(NUM)) >>
```

If we get an identifier, then we just add the `'id'` tag and return the type and the value as a tuple. If we get a string, we want to remove the quotes and process any special backslash codes, so we run `eval` on the quoted string. If we get a number, we convert it to an integer with `atoi` and then return the number along with its type tag.

The complete grammar is specified as follows:

```
parser Lisp:
    ignore:      '[ \t\n\r]+'
    token NUM:   '[0-9]+'
    token ID:    '[-+*/!@%^&=.a-zA-Z0-9_]+'
    token STR:   '"\\([^\\"]+\|\\\\.\\)*"'

    rule expr:   ID             ->  << ('id',ID) >>
              | STR             ->  << ('str',eval(STR)) >>
              | NUM             ->  << ('num',atoi(NUM)) >>
              | list            ->  << list >>
    rule list:   "(" seq ")"    ->  << seq >>
    rule seq:                   ->  << [] >>
              | expr seq        ->  << [expr] + seq >>
```

One thing you may have noticed is that `"("` and `")"` appear in the `list` rule. These are *inline tokens*: they appear in the rules without being given a name with the `token` keyword. Inline tokens are more convenient to use, but since they do not have a name, the text that is matched cannot be used in the return value. They are best used for short simple patterns (usually punctuation).

Another thing to notice is that the number and identifier tokens overlap. For example, "487" matches both NUM and ID. In Yapps, the scanner only tries to match tokens that are acceptable to the parser. This rule doesn't help here, since both NUM and ID can appear in the same place in the grammar. There are two rules used to pick tokens if more than one matches. One is that the *longest* match is preferred. For example, "487x" will match as an ID (487x) rather than as a NUM (487) followed by an ID (x). The second rule is that if the two matches are the same length, the *first* one listed in the grammar is preferred. For example, "487" will match as an NUM rather than an ID because NUM is listed first in the grammar. Inline tokens have preference over any tokens you have listed.

Now that our grammar is defined, we can run Yapps to produce a parser, and then run the parser to produce a parse tree.

Running Yapps

In the Yapps module is a function `generate` that takes an input filename and writes a parser to another file. We can use this function to generate the Lisp parser, which is assumed to be in `lisp.g`.

```
% python
Python 1.4 (Nov 17 1996)  [GCC 2.7.2.1]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import yapps
>>> yapps.generate('lisp.g')
```

At this point, Yapps has written a file `lisp.py` that contains the parser. In that file are two classes (one scanner and one parser) and a function (called `parse`) that puts things together for you.

Alternatively, we can run Yapps from the command line to generate the parser file:

```
% python yapps.py lisp.g
```

After running Yapps either from within Python or from the command line, we can use the Lisp parser by calling the `parse` function. The first parameter should be the rule we want to match, and the second parameter should be the string to parse.

```
>>> import lisp
>>> lisp.parse('expr', '(+ 3 4)')
[('id', '+'), ('num', 3), ('num', 4)]
>>> lisp.parse('expr', '(print "3 = " (+ 1 2))')
[('id', 'print'), ('str', '3 = '), [('id', '+'), ('num', 1), ('num', 2)]]
```

The `parse` function is not the only way to use the parser; section 5.1 describes how to access parser objects directly.

We've now gone through the steps in creating a grammar, writing a grammar file for Yapps, producing a parser, and using the parser. In the next example we'll see how rules can take parameters and also how to do computations instead of just returning a parse tree.

2.3 Calculator

A common example parser given in many textbooks is that for simple expressions, with numbers, addition, subtraction, multiplication, division, and parenthesization of subexpressions. We'll write this example in Yapps, using parameterization of rules to pass partially computed values from one rule to another.

Unlike `yacc`, Yapps does not have any way to specify precedence rules, so we have to do it ourselves. We say that an expression is the sum of terms, and that a term is the product of factors, and that a factor is a number or a parenthesized expression:

```
expr:           factor expr_tail
expr_tail:         | "+" factor expr_tail | "-" factor expr_tail
factor:         term factor_tail
factor_tail:       | "*" term factor_tail | "/" term factor_tail
term:           NUM | "(" expr ")"
```

In order to evaluate the expression as we go, we should keep along an accumulator while evaluating the lists of terms or factors. In the clause `"+" factor expr_tail`, we want to add the evaluation of `factor` to the accumulator, and then pass it to `expr_tail` so that it can continue adding to or subtracting from the accumulator. The accumulator can be a parameter passed to the two tail rules. When we call the tail rule again, we pass in a new value for the accumulator. The full grammar is given below:

```
parser Calculator:
  token END: "\\'"
  token NUM: "[0-9]+"

  rule goal:           expr END                          -> << expr        >>
  rule expr:           factor expr_tail<<factor>>        -> << expr_tail   >>
  rule expr_tail<<v>>:                                   -> << v           >>
                     | "+" factor expr_tail<<v+factor>>  -> << expr_tail   >>
                     | "-" factor expr_tail<<v-factor>>  -> << expr_tail   >>
  rule factor:         term factor_tail<<term>>          -> << factor_tail >>
  rule factor_tail<<v>>:                                 -> << v           >>
                     | "*" term factor_tail<<v*term>>    -> << factor_tail >>
                     | "/" term factor_tail<<v/term>>    -> << factor_tail >>
  rule term:           NUM                               -> << atoi(NUM)   >>
                     | "(" expr ")"                      -> << expr        >>
```

The top-level rule is *goal*, which says that we are looking for an expression followed by the end of the string. The `END` token is needed because without it, it isn't clear when to stop parsing. For example, the string "1+3" could be parsed either as the expression "1" followed by the string "+3" or it could be parsed as the expression "1+3". By requiring expressions to end with `END`, the parser is forced to take "1+3".

In the two *tail* rules, the accumulator is named `v`. Before calling the tail rule again, the accumulator is modified by adding, subtracting, multiplying by, or dividing by the first element of the list. The recursive call to the tail rule will return the value of the accumulator after all changes have been made, so we just return the value that the recursive call returned. Those of you familiar with Scheme or ML programming will recognize this as a tail-recursive accumulator-style loop. Yapps recognizes this kind of recursion, and produces a `while` loop instead of a recursive function; see section 5.6 for details.

The calculator example shows how to process lists of elements using parameters to rules, as well as how to handle precedence of operators. In an appendix, we will extend the calculator example to work with variables and assignment statements.

Note: It's often important to put the `END` token in, so put it in unless you are sure that your grammar has some other non-ambiguous token marking the end of the program.

3 Grammars

Each Yapps grammar has a name, a list of tokens, and a set of production rules. A grammar named `X` will be used to produce a parser named `X` and a scanner anmed `XScanner`. As in Python, names are case sensitive, start with a letter, and contain letters, numbers, and underscores (_).

There are three kinds of tokens in Yapps: named, inline, and ignored. As their name implies, named tokens are given a name, using the token construct: `token` *`name`* `:` *`regexp`*. In a rule, the token can be matched by using the name. Inline tokens are regular expressions that are used in rules without being declared. Ignored tokens are declared using the ignore construct:

`ignore:` *`regexp`*. These tokens are ignored by the scanner, and are not seen by the parser. Often whitespace is an ignored token. The regular expressions used to define tokens are treated like those used in Python code, so backslashes and quotes may have to be doubled.

Production rules in Yapps ahve a name and a series of clauses, separated by bars (`|`): `rule` *`name`* `:` *`clause`* `|` ... `|` *`clause`*. If the rule is parameterized, the name should be followed by a list of parameter names in `<<...>>`. Each clause is a list of patterns to match, an arrow (`->`), and a return value enclosed in `<<...>>`. A pattern can be the name of a named token, a regular expression in quotes (inline token), or the name of a production rule (followed by arguments in `<<...>>`, if the rule has parameters). The return value can use the text matched by a named token by using the token name. It can use the value returned by a production rule by using the name of that rule. If a name `X` occurs in more than one pattern, the return value can use `X1`, `X2`, etc. to refer to the occurrences of `X`, numbered from left to right.

Whenever `<<...>>` is used, a legal Python expression should be put inside the brackets. Python *statements* and *comments* are not allowed. The token `>>` should not appear within the `<<...>>` section, even within a string, since Yapps does not attempt to parse strings, parentheses, brackets, and braces within the `<<...>>`. A workaround for strings is to put two strings together (`">" ">"`), or to use backslashes (`">\>"`).

3.1 Left Factoring

Yapps produces LL(1) parsers, which determine which clause to match based on the first token available. Sometimes the leftmost tokens of several clauses may be the same. The classic example is the *if/then/else* construct in Pascal:

```
rule stmt:  "if" expr "then" stmt "else" stmt  ->
                << ('If',expr,stmt1,stmt1) >>
          | "if" expr "then" stmt ->
                << ('If',expr,stmt,[]) >>
```

The left portions of the two clauses are the same, which presents a problem for the parser. The solution is *left-factoring*: the common parts are put into a separate rule:

```
rule stmt:  "if" expr "then" stmt if_rest<<expr,stmt>>  ->  << if_rest >>
rule if_rest<<e,s>>:  "else" stmt -> << ('If',e,s,stmt) >>
                    |             -> << ('If',e,s,[]) >>
```

In general, replace rules of the form:

```
rule A:   a b1 -> << E1 >>
        | a b2 -> << E2 >>
        | c3   -> << E3 >>
        | c4   -> << E4 >>
```

with two rules of the form:

```
rule A:   a A_rest<<a>> -> << A_rest >>
        | c3   -> << E3 >>
        | c4   -> << E4 >>
rule A_rest<<a>>:   b1 -> << E1 >>
                  | b2 -> << E2 >>
```

For each rule, Yapps can determine which clause to match based on the first token available. Unfortunately, the classic *if/then/else* situation is *still* ambiguous when you left-factor. Yapps can deal with this situation, but will report a warning; see section 3.3 for details.

3.2 Left Recursion

A common construct in grammars is for matching a list of patterns, sometimes separated with delimiters such as commas or semicolons. In LR-based parser systems, we can parse a list with something like this:

```
rule sum:  NUM              -> << NUM >>
         | sum "+" NUM      -> << (sum, NUM) >>
```

Parsing `1+2+3+4` would produce the output `(((1,2),3),4)`, which is what we want from a left-associative addition operator. Unfortunately, this grammar is *left recursive,* because the `sum` rule contains a clause that begins with `sum`. (The recursion occurs at the left side of the clause.)

We must restructure this grammar to be *right recursive* instead:

```
rule sum:  NUM              -> << NUM >>
         | NUM "+" sum      -> << (NUM, sum) >>
```

Unfortunately, using this grammar, `1+2+3+4` would be parsed as `(1,(2,(3,4)))`, which no longer follows left associativity. The rule also needs to be left-factored. Instead, we write the following:

```
rule sum:                NUM sum_rest<<NUM>> -> << sum_rest >>
rule sum_rest<<left>>:                                -> << left >>
                       | "+" NUM sum_rest<<(left,NUM)>> -> << sum_rest >>
```

Oddly enough, after left-factoring the resulting parser produces the desired left-associative parse tree. This version is also tail-recursive, so the resulting parser will be a loop instead of a recursive Python function.

In general, replace rules of the form:

```
rule A:  A a1 -> << E1 >>
       | A a2 -> << E2 >>
       | b3   -> << E3 >>
       | b4   -> << E4 >>
```

with two rules of the form:

```
rule A:  b3 A_rest<<E3>> -> << A_rest >>
       | b4 A_rest<<E4>> -> << A_rest >>
rule A_rest<<A>>:  a1 A_rest<<E1>> -> << A_rest >>
                 | a2 A_rest<<E2>> -> << A_rest >>
```

As with left-factoring, we have taken a rule that proved problematic for Yapps (and other LL(1) parser generators) and split it into two rules. With Yapps we can use parameterized rules to pass information from one rule to the other, so using two rules instead of one does not lead to any signficant restrictions on the kinds of grammars that can be written.

3.3 Ambiguous Grammars

In section 3.1 we saw the classic if/then/else ambiguity, which occurs because the "else ..." portion of an "if ... then ... else ..." construct is optional. Programs with nested if/then/else constructs can be ambiguous when one of the else clauses is missing:

```
if 1 then            if 1 then
    if 5 then            if 5 then
        x := 1;              x := 1;
    else             else
        y := 9;              y := 9;
```

The indentation shows that the program can be parsed in two different ways. (Of course, if we all would adopt Python's indentation-based structuring, this would never happen!) Usually we want the parsing on the left: the "else" should be associated with the closest "if" statement. In section 3.1 we "solved" the problem by using the following grammar:

```
rule stmt:  "if" expr "then" stmt if_rest<<expr,stmt>>  ->  << if_rest >>
rule if_rest<<e,s>>:  "else" stmt -> << ('If',e,s,stmt) >>
                    |             -> << ('If',e,s,[]) >>
```

Here, we have two clauses—one in which "else" is parsed and one in which nothing is parsed. The ambiguity is that if an "else" is present, it is not clear whether you want it parsed immediately or if you want it to be parsed by the outer "if".

Yapps will deal with the situation by reporting that there is an ambiguity (warning you that the parser may not work), but it will still produce a parser. Despite the warning, the parser will work in this case because it prefers the *first* matching clause, which tells Yapps to parse the "else". That is exactly what we want!

If you have the if/then/else ambiguity in your grammar, be sure to put the "else" clause before the empty clause. Yapps will produce a warning, but the parser will most likely work.

4 Customization

Both the parsers and the scanners can be customized. The parser is usually extended by subclassing, and the scanner can either be subclassed or completely replaced.

4.1 Customizing Parsers

If additional fields and methods are needed in order for a parser to work, Python subclassing can be used. (This is unlike parser classes written in static languages, in which these fields and methods must be defined in the generated parser class.) We simply subclass the generated parser, and add any fields or methods required. Expressions in the grammar can call methods of the subclass to perform any actions that cannot be expressed as a simple expression. For example, consider this simple grammar:

```
parser X:
    rule goal:  "something"  ->  << self.printmsg() >>
```

The `printmsg` function need not be implemented in the parser class `X`; it can be implemented in a subclass:

```
import Xparser

class MyX(Xparser.X):
    def printmsg(self):
        print "Hello!"
```

Since the grammar does not allow for arbitrary statements (only a single expression as a return value), this technique is useful for executing multiple statements to compute a return value.

4.2 Customizing Scanners

The generated parser class is not dependent on the generated scanner class. A scanner object is passed to the parser object's constructor in the `parse` function. To use a different scanner, write your own function to construct parser objects, with an instance of a different scanner. Scanner objects must have a `token` method that accepts an integer `N` as well as a list of allowed token types, and returns the Nth token, as a tuple. The default scanner raises `NoMoreTokens` if no tokens are available, and `SyntaxError` if no token could be matched. However, the parser does not rely on these exceptions; only the `parse` convenience function and the `print_error` error display function use those exceptions.

The tuples representing tokens have four elements. The first two are the beginning and ending indices of the matched text in the input string. The third element is the type tag, matching either the name of a named token or the quoted regexp of an inline or ignored token. The fourth element of the token tuple is the matched text. If the input string is `s`, and the token tuple is `(b,e,type,val)`, then `val` should be equal to `s[b:e]`.

5 Parser Mechanics

The parser class contains one method for each rule in the grammar. These methods return a pair, `(value,index)` where the value is a user-defined value and the index points to the next token that has not been used. The indices are the same as those passed to the `token` method in the scanner.

5.1 Parser Objects

Yapps produces as output two exception classes, a scanner class, a parser class, and a function `parse` that puts everything together. The `parse` function does not have to be used; instead, one can create a parser and scanner object and use them together for parsing.

```
def parse(rule, input):
    parser = X(XScanner(input))
    try: return getattr(parser, rule)()[0]
    except SyntaxError, s:
        try:
            import yapps
            yapps.print_error(input, s, parser.scanner)
        except ImportError:
            print 'Syntax Error',s.msg,'on line',1+count(input[:s.pos], '\n')
    except NoMoreTokens:
        print 'Ran out of input'
```

The `parse` function takes a name of a rule and an input string as input. It creates a scanner and parser object, then tries to execute the method in the parser object named `rule`. (The parser object contains one method for each of the rules in the grammar.) The parser method returns a pair: the value specified in the `<<...>>` part of the grammar, and the number of tokens that were read from the string. (The latter can be used if we wanted to parse something else from the string.) The `parse` function throws away the token count and just returns the first element of the tuple.

There are several situations in which the `parse` function would not be useful. If a different parser or scanner is being used, the token count is needed, or a exceptions are to be handled differently, a new `parse` function would be required. The supplied `parse` function can be used

as a template for writing a function for your own needs. An example of a custom parse function is the `generate` function in `Yapps.py`.

5.2 Context Sensitive Scanner

Unlike most scanners, the scanner produced by Yapps can take into account the context in which tokens are needed, and try to match only good tokens. For example, in the grammar:

```
parser IniFile:
   token ID:   "[a-zA-Z_0-9]+"
   token VAL:  ".*"

   rule pair:  ID "[ \t]*=[ \t]*" VAL "\n"
```

we would like to scan lines of text and pick out a name/value pair. In a conventional scanner, the input string `shell=progman.exe` would be turned into a single token of type `VAL`. The Yapps scanner, however, knows that at the beginning of the line, an `ID` is expected, so it will return `"shell"` as a token of type `ID`. Later, it will return `"progman.exe"` as a token of type `VAL`.

Context sensitivity decreases the separation between scanner and parser, but it is useful in parsers like `IniFile`, where the tokens themselves are not unambiguous, but *are* unambiguous given a particular stage in the parsing process.

Unfortunately, context sensitivity can make it more difficult to detect errors in the input. For example, in parsing a Pascal-like language with "begin" and "end" as keywords, a context sensitive scanner would only match "end" as the END token if the parser is in a place that will accept the END token. If not, then the scanner would match "end" as an identifier. To disable the context sensitive scanner in Yapps, use the `context-insensitive-scanner` option, either from the command line or in the parser:

```
Parser X:
    option:  "context-insensitive-scanner"
```

5.3 Internal Variables

There are two internal variables that may be of use. The current rule begins scanning the input string at character position `_start_`. The index of the current token being matched is `_pos_`. The token can be retrieved by accessing the scanner object, `self.scanner`. For example, to retrieve the four-element tuple representing the first token of the current rule, use `self.scanner.token(_pos_)`. A potentially useful combination of these two variables is to extract the portion of the input matched by the current rule. The second element of a token tuple is the end character position, so we want to extract the string from `_start_` to `self.scanner.token(_pos_)[1]`:

```
rule R: a b c ->
 << self.scanner.input[_start_:self.scanner.token(_pos_)[1]] >>
```

5.4 Pre- and Post-Parser Code

Sometimes the parser code needs to rely on helper variables, functions, and classes. A Yapps grammar can optionally be surrounded by double percent signs, to separate the grammar from Python code.

```
... Python code ...
%%
... Yapps grammar ...
%%
... Python code ...
```

The second `%%` can be omitted if there is no Python code at the end, and the first `%%` can be omitted if there is no extra Python code at all. (To have code only at the end, both separators are required.)

5.5 Python 1.5 Regular Expressions

If you want to use "new style" Python 1.5 regular expressions add an option to your grammar before you define tokens:

```
parser X:
    option:   "use-new-regexps"
    token:    ...
```

Yapps will then generate code to use the new regular expression module (`re`) rather than the old one (`regex`).

5.6 Optimizations Performed by Yapps

Yapps does several things that simplify the generated parser code, to make it faster, shorter, and more readable. Known changes to the internal variable `_pos_` are propagated to future statements instead of performing assignments to modify the variable. Also, some variables that would be assigned to, but never used, are not assigned. (The analysis of this situation is not as comprehensive as it could be.)

Yapps looks for a special case that commonly occurs when matching lists of patterns. Lists are commonly expressed with one of these two constructs:

```
    rule Xlist:       X Xlist -> << [X] + Xlist >>
                    |         -> << [] >>
    rule X:           ...

    rule Ylist<<r>>:  Y Ylist<<r+[Y]>> -> << Ylist >>
                    |                  -> << r >>
    rule Y:           ...
```

Both constructs produce the resulting list using recursion instead of looping (since looping is not available in Yapps). The second version, however, is called *tail-recursive*. To be a *tail call*, the return value in the recursive clause has to be simply the name of the last subrule matched; no computation can be performed on it before it is returned. The first clause of `Ylist` is a tail call because it returns `Ylist`, and the last subrule matched was `Ylist<<...>>`. To be tail recursive, it must be a tail call, and it must also be the same as the rule being defined. Since we are defining `Ylist` and returning `Ylist`, the clause is tail recursive.

Yapps will detect tail calls and make a minor optimization for them. The real win is in tail recursion: Yapps will turn the recursive function into a `while` loop.

6 Future Extensions

There are three extensions that are planned:

1. User defined actions inside clauses would allow parsing code to be mixed in with other code, which processes parts of a clause that have been parsed so far. For example, `ID {{ if ID not in self.declarations: raise InvalidIdentifier }} ":=" expr` would check, at parse time, whether the identifier has been declared, and if not, raise an exception. Only if the identifier had been declared would parsing continue to the ":=" and `expr`.
2. Parsing forms `X*` and `X+` would be very convenient for matching commonly occuring list patterns in languages.
3. At present, tokens must be strings. It would be convenient to also allow some standard regular expressions to be imported, most likely from the `tokenize` module, which defines regular expressions for Python strings, identifiers, numbers, and several other common tokens.
4. It would be nice to somehow automatically deal with end-of-string tokens. A common error I make is to forget to define END to be `"\\'"`, and then the resulting parser is unable to parse text that seems correct.

In addition, the possibility of backtracking and LL(k) parsing are being investigated.

A Grammar for Parsers

This is the grammar for parsers, without the return values. Everything inside `<...>` is a placeholder for something that is harder to read.

```
parser ParserDesc:
    ignore:      "[ \t\n\r]+"
    token END:   "\\'"
    token ATTR:  "<<\\([^>]+\\|>[^>]\\)*>>"
    token ID:    '[a-zA-Z_][a-zA-Z_0-9]*'
    token STR:   <python single and double quoted strings>

    rule Parser: "parser" ID ":" Options Tokens Rules END
    rule Options: <nothing> | "option" ":" STR Options
    rule Tokens:  <nothing>
                | "token" ID ":" STR Tokens
                | "ignore"   ":" STR Tokens
    rule Rules: <nothing>  |  "rule" ID OptParam ":" Clause Clauses Rules
    rule Clause: Subs "->" ATTR
    rule Clauses: <nothing> | "|" Clause Clauses
    rule OptParam: <nothing> | ATTR
    rule Subs: <nothing>
             | STR Subs
             | ID OptParam Subs
```

B Calculator Example

(An example, yet to be written, that shows how to create a calculator with variables, similar to the parser in Lutz's Python book.)

C References

1. ANTLR/PCCTS, by Terrence Parr, is available at The ANTLR Home Page.
2. PyLR, by Scott Cotton, is at his Starship page.
3. PyBison, by Scott Hassan, can be found at his Python Projects page.
4. mcf.pars, by Mike C. Fletcher, is available at his web page.
5. kwParsing, by Aaron Watters, is available at his Starship page.