The *Yapps* Parser Generator System

`http://theory.stanford.edu/~amitp/Yapps/`
Version 2

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

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 the fastest or most powerful parser. 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. The types of defined variables can be used in parsing as well—for example, if the type of `X` is known, we can determine whether `X(1)` is an array reference or a function call.

3. Yapps grammars are fairly easy to write, although there are some inconveniences having to do with ELL(1) parsing that have to be worked around. For example, rules have to be left factored and rules may not be left recursive. However, neither limitation seems to be a problem in practice.

   Yapps grammars look similar to the notation used in the Python reference manual, with operators like `*`, `+`, `|`, `[]`, and `()` for patterns, names (`tim`) for rules, regular expressions (`"[a-z]+"`) for tokens, and `#` for comments.

4. The Yapps parser generator is written as a single Python module with no C extensions. Yapps produces parsers that are written entirely in Python, and require only the Yapps run-time module (5k) for support.

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 `ELL(1)` (Extended 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. For example, allowing both `x := 5;` and `x;` as statements is difficult because we must distinguish based on only one token of lookahead. Seeing only `x`, we cannot decide whether we have an assignment statement or an expression statement. (Note however that this kind of grammar can be matched with backtracking; see section E.)

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.

Yapps 2.0 is easier to use than Yapps 1.0. New features include a less restrictive input syntax, which allows mixing of sequences, choices, terminals, and nonterminals; optional matching; the ability to insert single-line statements into the generated parser; and looping constructs `*` and `+` similar to the repetitive matching constructs in regular expressions. Unfortunately, the addition of these constructs has made Yapps 2.0 incompatible with Yapps 1.0, so grammars will have to be rewritten. See section **??** for tips on changing Yapps 1.0 grammars for use with Yapps 2.0.

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 expression evaluation grammar that evaluates during parsing (instead of producing a parse tree).

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: [article] adjective* noun`

Notice that some things that we said easily in English, such as “optional article,” are expressed using special syntax, such as brackets. When we said, “any number of adjectives,” we wrote `adjective*`, where the `*` means “zero or more of the preceding pattern”.

The grammar given above is close to a Yapps grammar. We also have to specify what the tokens are, and wh above is close to a Yapps grammar. We also have to specify what the tokens are, and what to do when a pattern is matched. For this example, we will do nothing when patterns are matched; the next example will explain how to perform match actions.

```
parser TinyEnglish:
  ignore:          "\\W+"
  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:    [article] adjective* noun
```

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. (*Note:* These are Python 1.5 regular expressions from the `re` module, not Python 1.4 regular expressions from the `regex` module.) 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.

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:   ( expr* )
```

In addition to having a grammar, we need to specify what to do 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.

Once some pattern is matched, we enclose a return statement enclosed in `{{...}}`. The braces allow us to insert any one-line statement into the parser. Within this statement, we can refer to the values returned by matching each part of the rule. After matching a token such as `ID`, "ID" will be bound to the text of the matched token. Let's take a look at the rule:

```
rule expr: ID   {{ return ('id', ID) }}
  ...
```

In a rule, tokens return the text that was matched. For identifiers, we just return the identifier, along with a "tag" telling us that this is an identifier and not a string or some other value. Sometimes 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. Let's look at the return values for the other tokens:

```
  ...
         | STR  {{ return ('str', eval(STR)) }}
         | NUM  {{ return ('num', atoi(NUM)) }}
  ...
```

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.

For matching a list, we need to do something slightly more complicated. If we match a Lisp list of expressions, we want to create a Python list with those values.

```
rule list: "\\("                  # Match the opening parenthesis
           {{ result = [] }}      # Create a Python list
           (
              expr                # When we match an expression,
              {{ result.append(expr) }}   # add it to the list
           )*                     # * means repeat this if needed
           "\\)"                  # Match the closing parenthesis
           {{ return result }}    # Return the Python list
```

In this rule we first match the opening parenthesis, then go into a loop. In this loop we match expressions and add them to the list. When there are no more expressions to match, we match the closing parenthesis and return the resulting. Note that `#` is used for comments, just as in Python.

The complete grammar is specified as follows:

```
parser Lisp:
    ignore:      '\\s+'
    token NUM:   '[0-9]+'
    token ID:    '[-+*/!@%^&=.a-zA-Z0-9_]+'
    token STR:   '"([^\\"]+|\\\\.)*"'

    rule expr:   ID     {{ return ('id', ID) }}
               | STR    {{ return ('str', eval(STR)) }}
               | NUM    {{ return ('num', atoi(NUM)) }}
               | list   {{ return list }}
    rule list: "\\("    {{ result = [] }}
               ( expr   {{ result.append(expr) }}
               )*
               "\\)"    {{ return result }}
```

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 or keywords).

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 1.5.1 (#1, Sep  3 1998, 22:51:17)  [GCC 2.7.2.3] on linux-i386
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.

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, evaluating the expression as we parse.

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 ( ("+"|"-") factor )*
    factor:         term   ( ("*"|"/") term )*
    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. Just as we kept a "result" variable to build a parse tree for Lisp expressions, we will use a variable to evaluate numerical expressions. The full grammar is given below:

```
parser Calculator:
    token END: "$"          # $ means end of string
```

```
token NUM: "[0-9]+"

rule goal:           expr END         {{ return expr }}

# An expression is the sum and difference of factors
rule expr:           factor           {{ v = factor }}
                   ( "[+]" factor       {{ v = v+factor }}
                   |  "-"  factor       {{ v = v-factor }}
                   )*                 {{ return v }}

# A factor is the product and division of terms
rule factor:         term             {{ v = term }}
                   ( "[*]" term         {{ v = v*term }}
                   |  "/"  term         {{ v = v/term }}
                   )*                 {{ return v }}

# A term is either a number or an expression surrounded by parentheses
rule term:           NUM              {{ return atoi(NUM) }}
                   | "\\(" expr "\\)" {{ return 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 rules with repetition, the accumulator is named `v`. After reading in one expression, we initialize the accumulator. Each time through the loop, we modify the accumulator by adding, subtracting, multiplying by, or dividing the previous accumulator by the expression that has been parsed. At the end of the rule, we return the accumulator.

The calculator example shows how to process lists of elements using loops, as well as how to handle precedence of operators.

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.

Calculator with Memory

In the previous example we learned how to write a calculator that evaluates simple numerical expressions. In this section we will extend the example to support both local and global variables.

To support global variables, we will add assignment statements to the "goal" rule.

```
rule goal:           expr END         {{ return expr }}
          | 'set' ID expr END         {{ global_vars[ID] = expr }}
                                      {{ return expr }}
```

To use these variables, we need a new kind of terminal:

```
rule term: ... | ID {{ return global_vars[ID] }}
```

So far, these changes are straightforward. We simply have a global dictionary `global_vars` that stores the variables and values, we modify it when there is an assignment statement, and we look up variables in it when we see a variable name.

To support local variables, we will add variable declarations to the set of allowed expressions.

```
rule term: ... | 'let' VAR '=' expr 'in' expr ...
```

This is where it becomes tricky. Local variables should be stored in a local dictionary, not in the global one. One trick would be to save a copy of the global dictionary, modify it, and then restore it later. In this example we will instead use *attributes* to create local information and pass it to subrules.

A rule can optionally take parameters. When we invoke the rule, we must pass in arguments. For local variables, let's use a single parameter, `local_vars`:

```
rule expr<<local_vars>>:   ...
rule factor<<local_vars>>: ...
rule term<<local_vars>>:   ...
```

Each time we want to match `expr`, `factor`, or `term`, we will pass the local variables in the current rule to the subrule. One interesting case is when we pass as an argument something *other* than `local_vars`:

```
rule term<<local_vars>>: ...
              | 'let' VAR '=' expr<<local_vars>>
                {{ local_vars = [(VAR, expr)] + local_vars }}
                'in' expr<<local_vars>>
                {{ return expr }}
```

Note that the assignment to the local variables list does not modify the original list. This is important to keep local variables from being seen outside the "let".

The other interesting case is when we find a variable:

```
global_vars = {}

def lookup(map, name):
    for x,v in map:  if x==name: return v
    return global_vars[name]
%%
   ...
   rule term<<local_vars>: ...
              | VAR {{ return lookup(local_vars, VAR) }}
```

The lookup function will search through the local variable list, and if it cannot find the name there, it will look it up in the global variable dictionary.

A complete grammar for this example, including a read-eval-print loop for interacting with the calculator, can be found in the examples subdirectory included with Yapps.

In this section we saw how to insert code before the parser. We also saw how to use attributes to transmit local information from one rule to its subrules.

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 should use the syntax defined in the `re` module, so some symbols may have to be backslashed.

Production rules in Yapps have a name and a pattern to match. If the rule is parameterized, the name should be followed by a list of parameter names in `<<...>>`. A pattern can be a simple pattern or a compound pattern. Simple patterns are the name of a named token, a regular expression in quotes (inline token), the name of a production rule (followed by arguments in `<<...>>`, if the rule has parameters), and single line Python statements (`{{...}}`). Compound patterns are sequences (`A B C ...`), choices (`A | B | C | ...`), options (`[...]`), zero-or-more repetitions (`...*`), and one-or-more repetitions (`...+`). Like regular expressions, repetition operators have a higher precedence than sequences, and sequences have a higher precedence than choices.

Whenever `{{...}}` is used, a legal one-line Python statement should be put inside the braces. The token `}}` should not appear within the `{{...}}` section, even within a string, since Yapps does not attempt to parse the Python statement. A workaround for strings is to put two strings together (`"}" "}"`), or to use backslashes (`"}\}"`). At the end of a rule you should use a `{{ return X }}` statement to return a value. However, you should *not* use any control statements (`return`, `continue`, `break`) in the middle of a rule. Yapps needs to make assumptions about the control flow to generate a parser, and any changes to the control flow will confuse Yapps.

The `<<...>>` form can occur in two places: to define parameters to a rule and to give arguments when matching a rule. Parameters use the syntax used for Python functions, so they can include default arguments and the special forms (`*args` and `**kwargs`). Arguments use the syntax for Python function call arguments, so they can include normal arguments and keyword arguments. The token >> should not appear within the `<<...>>` section.

In both the statements and rule arguments, you can use names defined by the parser to refer to matched patterns. You can refer to the text matched by a named token by using the token name. You can use the value returned by a production rule by using the name of that rule. If a name `X` is matched more than once (such as in loops), you will have to save the earlier value(s) in a temporary variable, and then use that temporary variable in the return value. The next section has an example of a name that occurs more than once.

Left Factoring

Yapps produces ELL(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 {{ then_part = stmt }}
                      "else" stmt {{ return ('If',expr,then_part,stmt) }}
          | "if" expr "then" stmt {{ return ('If',expr,stmt,[]) }}
```

(Note that we have to save the first `stmt` into a variable because there is another `stmt` that will be matched.) 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 together, and *then* a choice is made about the remaining part:

```
rule stmt:  "if" expr
              "then" stmt {{ then_part = stmt }}
              {{ else_part = [] }}
              [ "else" stmt {{ else_part = stmt }} ]
            {{ return ('If', expr, then_part, else_part) }}
```

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.

In general, replace rules of the form:

```
rule A:   a b1 {{ return E1 }}
        | a b2 {{ return E2 }}
```

```
        | c3   {{ return E3 }}
        | c4   {{ return E4 }}
```

with rules of the form:

```
rule A:    a ( b1 {{ return E1 }}
             | b2 {{ return E2 }}
             )
        | c3   {{ return E3 }}
        | c4   {{ return E4 }}
```

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              {{ return NUM }}
         | sum "+" NUM      {{ return (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              {{ return NUM }}
         | NUM "+" sum      {{ return (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 pattern as a loop instead:

```
rule sum:       NUM {{ v = NUM }}
                ( "[+]" NUM {{ v = (v,NUM) }} )*
                {{ return v }}
```

In general, replace rules of the form:

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

with rules of the form:

```
rule A:  ( b3 {{ A = E3 }}
         | b4 {{ A = E4 }} )
         ( a1 {{ A = E1 }}
         | a2 {{ A = E2 }} )*
         {{ return A }}
```

We have taken a rule that proved problematic for with recursion and turned it into a rule that works well with looping constructs.

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 {{ then_part = stmt }}
              {{ else_part = [] }}
              [ "else" stmt {{ else_part = stmt }} ]
            {{ return ('If', expr, then_part, else_part) }}
```

Here, we have an optional match of "else" followed by a statement. 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 matching when the else pattern when it can. 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!

For ambiguity cases with choices, Yapps will choose the *first* matching choice. However, remember that Yapps only looks at the first token to determine its decision, so `(a b | a c)` will result in Yapps choosing `a b` even when the input is `a c`. It only looks at the first token, `a`, to make its decision.

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.

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!"
```

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 (which calls `wrap_error_reporter`) 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]`.

The generated parsers do not the beginning or ending index. They use only the token type and value. However, the default error reporter uses the beginning and ending index to show the user where the error is.

Parser Mechanics

The base parser class (Parser) defines two methods, `_scan` and `_peek`, and two fields, `_pos` and `_scanner`. The generated parser inherits from the base parser, and contains one method for each rule in the grammar. To avoid name clashes, do not use names that begin with an underscore (_).

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, text):
    P = X(XScanner(text))
    return wrap_error_reporter(P, rule)
```

The `parse` function takes a name of a rule and an input string as input. It creates a scanner and parser object, then calls `wrap_error_reporter` to execute the method in the parser object named `rule`. The wrapper function will call the appropriate parser rule and report any parsing errors to standard output.

There are several situations in which the `parse` function would not be useful. If a different parser or scanner is being used, or 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`.

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, add the `context-insensitive-scanner` option to the grammar:

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

Context-insensitive scanning makes the parser look cleaner as well.

Internal Variables

There are two internal fields that may be of use. The parser object has two fields, `_pos`, which is the index of the current token being matched, and `_scanner`, which is the scanner object. The token itself can be retrieved by accessing the scanner object and calling the `token` method with the token index. However, if you call `token` before the token has been requested by the parser, it may mess up a context-sensitive scanner.[1] A potentially useful combination of these fields is to extract the portion of the input matched by the current rule. To do this, just save the scanner state (`_scanner.pos`) before the text is matched and then again after the text is matched:

```
  rule R:
      {{ start = self._scanner.pos }}
      a b c
      {{ end = self._scanner.pos }}
      {{ print 'Text is', self._scanner.input[start:end] }}
```

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.)

If the second `%%` is omitted, Yapps will insert testing code that allows you to use the generated parser to parse a file.

The extended calculator example in the Yapps examples subdirectory includes both pre-parser and post-parser code.

Representation of Grammars

For each kind of pattern there is a class derived from Pattern. Yapps has classes for Terminal, NonTerminal, Sequence, Choice, Option, Plus, Star, and Eval. Each of these classes has the following interface:

[1] When using a context-sensitive scanner, the parser tells the scanner what the valid token types are at each point. If you call `token` before the parser can tell the scanner the valid token types, the scanner will attempt to match without considering the context.

setup(*gen*) Set accepts-ϵ, and call *gen.changed()* if it changed. This function can change the flag from false to true but *not* from true to false.

update((gen)) Set FIRSTand FOLLOW, and call *gen.changed()* if either changed. This function can add to the sets but *not* remove from them.

output(*gen*, *indent*) Generate code for matching this rule, using *indent* as the current indentation level. Writes are performed using *gen.write*.

used(*vars*) Given a list of variables *vars*, return two lists: one containing the variables that are used, and one containing the variables that are assigned. This function is used for optimizing the resulting code.

Both *setup* and *update* monotonically increase the variables they modify. Since the variables can only increase a finite number of times, we can repeatedly call the function until the variable stabilized. The *used* function is not currently implemented.

With each pattern in the grammar Yapps associates three pieces of information: the FIRSTset, the FOLLOWset, and the accepts-ϵ flag.

The FIRSTset contains the tokens that can appear as we start matching the pattern. The FOLLOWset contains the tokens that can appear immediately after we match the pattern. The accepts-ϵ flag is true if the pattern can match no tokens. In this case, FIRSTwill contain all the elements in FOLLOW. The FOLLOWset is not needed when accepts-ϵ is false, and may not be accurate in those cases.

Yapps does not compute these sets precisely. Its approximation can miss certain cases, such as this one:

```
rule C: ( A* | B )
rule B: C [A]
```

Yapps will calculate `C`'s FOLLOWset to include `A`. However, `C` will always match all the `A`'s, so `A` will never follow it. Yapps 2.0 does not properly handle this construct, but if it seems important, I may add support for it in a future version.

Yapps also cannot handle constructs that depend on the calling sequence. For example:

```
rule R: U | 'b'
rule S:   | 'c'
rule T: S 'b'
rule U: S 'a'
```

The FOLLOWset for `S` includes `a` and `b`. Since `S` can be empty, the FIRSTset for `S` should include `a`, `b`, and `c`. However, when parsing `R`, if the lookahead is `b` we should *not* parse `U`. That's because in `U`, `S` is followed by `a` and not `b`. Therefore in `R`, we should choose rule `U` only if there is an `a` or `c`, but not if there is a `b`. Yapps and many other LL(1) systems do not distinguish `S b` and `S a`, making `S`'s FOLLOWset `a, b`, and making `R` always try to match `U`. In this case we can solve the problem by changing `R` to `'b' | U` but it may not always be possible to solve all such problems in this way.

Grammar for Parsers

This is the grammar for parsers, without any Python code mixed in. The complete grammar can be found in `parsedesc.g` in the Yapps distribution.

```
parser ParserDescription:
    ignore:      "\\s+"
    ignore:      "#.*?\r?\n"
```

```
token END:    "$"  # $ means end of string
token ATTR:   "<<.+?>>"
token STMT:   "{{.+?}}"
token ID:     '[a-zA-Z_][a-zA-Z_0-9]*'
token STR:    '[rR]?\'([^\\n\'\\\\]|\\\\.)*\'|[rR]?"([^\\n"\\\\]|\\\\.)*"'

rule Parser: "parser" ID ":"
               Options
               Tokens
               Rules
             END

rule Options:  ( "option" ":" STR )*
rule Tokens:   ( "token" ID ":" STR | "ignore"   ":" STR )*
rule Rules:    ( "rule" ID OptParam ":" ClauseA )*

rule ClauseA:  ClauseB ( '[|]' ClauseB )*
rule ClauseB:  ClauseC*
rule ClauseC:  ClauseD [ '[+]' | '[*]' ]
rule ClauseD:  STR | ID [ATTR] | STMT
             | '\\(' ClauseA '\\) | '\\[' ClauseA '\\]'
```

Upgrading

Yapps 2.0 is not backwards compatible with Yapps 1.0. In this section are some tips for upgrading:

1. Yapps 1.0 was distributed as a single file. Yapps 2.0 is instead distributed as two Python files: a *parser generator* (26k) and a *parser runtime* (5k). You need both files to create parsers, but you need only the runtime (`yappsrt.py`) to use the parsers.

2. Yapps 1.0 supported Python 1.4 regular expressions from the `regex` module. Yapps 2.0 uses Python 1.5 regular expressions from the `re` module. *The new syntax for regular expressions is not compatible with the old syntax.* Andrew Kuchling has a guide to converting regular expressions on his web page.

3. Yapps 1.0 wants a pattern and then a return value in `->` `<<...>>`. Yapps 2.0 allows patterns and Python statements to be mixed. To convert a rule like this:

   ```
   rule R: A B C -> << E1 >>
         | X Y Z -> << E2 >>
   ```

   to Yapps 2.0 form, replace the return value specifiers with return statements:

   ```
   rule R: A B C {{ return E1 }}
         | X Y Z {{ return E2 }}
   ```

4. Yapps 2.0 does not perform tail recursion elimination. This means any recursive rules you write will be turned into recursive methods in the parser. The parser will work, but may be slower. It can be made faster by rewriting recursive rules, using instead the looping operators `*` and `+` provided in Yapps 2.0.

Troubleshooting

- A common error is to write a grammar that doesn't have an END token. End tokens are needed when it is not clear when to stop parsing. For example, when parsing the expression `3+5`, it is not clear after reading `3` whether to treat it as a complete expression or whether the parser should continue reading. Therefore the grammar for numeric expressions should include an end token. Another example is the grammar for Lisp expressions. In Lisp, it is always clear when you should stop parsing, so you do *not* need an end token. In fact, it may be more useful not to have an end token, so that you can read in several Lisp expressions.

- If there is a chance of ambiguity, make sure to put the choices in the order you want them checked. Usually the most specific choice should be first. Empty sequences should usually be last.

- The context sensitive scanner is not appropriate for all grammars. You might try using the insensitive scanner with the `context-insensitive-scanner` option in the grammar.

- If performance turns out to be a problem, try writing a custom scanner. The Yapps scanner is rather slow (but flexible and easy to understand).

History

Yapps 1 had several limitations that bothered me while writing parsers:

1. It was not possible to insert statements into the generated parser. A common workaround was to write an auxilliary function that executed those statements, and to call that function as part of the return value calculation. For example, several of my parsers had an "append(x,y)" function that existed solely to call "x.append(y)".

2. The way in which grammars were specified was rather restrictive: a rule was a choice of clauses. Each clause was a sequence of tokens and rule names, followed by a return value.

3. Optional matching had to be put into a separate rule because choices were only made at the beginning of a rule.

4. Repetition had to be specified in terms of recursion. Not only was this awkward (sometimes requiring additional rules), I had to add a tail recursion optimization to Yapps to transform the recursion back into a loop.

Yapps 2 addresses each of these limitations.

1. Statements can occur anywhere within a rule. (However, only one-line statements are allowed; multiline blocks marked by indentation are not.)

2. Grammars can be specified using any mix of sequences, choices, tokens, and rule names. To allow for complex structures, parentheses can be used for grouping.

3. Given choices and parenthesization, optional matching can be expressed as a choice between some pattern and nothing. In addition, Yapps 2 has the convenience syntax `[A B ...]` for matching `A B ...` optionally.

4. Repetition operators `*` for zero or more and `+` for one or more make it easy to specify repeating patterns.

It is my hope that Yapps 2 will be flexible enough to meet my needs for another year, yet simple enough that I do not hesitate to use it.

Future Extensions

I am still investigating the possibility of LL(2) and higher lookahead. However, it looks like the resulting parsers will be somewhat ugly.

It would be nice to control choices with user-defined predicates.

The most likely future extension is backtracking. A grammar pattern like `(VAR ':=' expr)? {{ return Assign(VAR,expr) }} : expr {{ return expr }}` would turn into code that attempted to match `VAR ':=' expr`. If it succeeded, it would run `{{ return ... }}`. If it failed, it would match `expr {{ return expr }}`. Backtracking may make it less necessary to write LL(2) grammars.

References

1. The Python-Parser SIG is the first place to look for a list of parser systems for Python.

2. ANTLR/PCCTS, by Terrence Parr, is available at The ANTLR Home Page.

3. PyLR, by Scott Cotton, is at his Starship page.

4. John Aycock's Compiling Little Languages Framework.

5. PyBison, by Scott Hassan, can be found at his Python Projects page.

6. mcf.pars, by Mike C. Fletcher, is available at his web page.

7. kwParsing, by Aaron Watters, is available at his Starship page.