Indentation Sensitive Languages

Leonhard Brunauer
lbrunau@cosy.sbg.ac.at

Bernhard Mühlbacher
bmuehl@cosy.sbg.ac.at

July 14, 2006

Abstract

In this work, we present a theoretical and practical approach for dealing with a complier related problem. Although, most of modern programming languages use explicit grouping of nested statements by means of special begin and end tokens, there are programming languages that use indentation for this purpose. The most famous example for such a language is Python. We show that languages, using this kind of syntax, cannot be generated by a context free grammar and present an extension to context free grammars that is suitable to handle the problem of determining the nesting level. Furthermore, we present a set of techniques that make it possible to construct an efficient parser out of it.

1 Introduction

Most of todays common imperative programming languages allow arbitrary nesting of statements by using some kind of `BEGIN` and `END` tokens (or some variant thereof, like C's '{' and '}'). Some languages, however, determine the nesting level by counting indentation symbols. Usually, each statement has its own line and the compiler counts the number of tabulators - or whichever indentation symbol else - at the beginning of each line. Examples for this kind of languages are Python and Haskell.

Example 1. First of all, let us give an example to make this difference more obvious.

```
def sign(i: int) -> int:
    if i < 0:
        return -1
    elif i == 0:
        return 0
    else:
        return 1
```

Creating a grammar for a language that uses indentation for determining the nesting level - named *indentation sensitive language* in the following - is not generally possible, as we will see in Section 2. Python and Haskell solve the problem by doing some preprocessing by translating indentation symbols to

`BEGIN` and `END` like tokens. The actual implementation can be found in [1] and [2], respectively.

In this work we tried to define an extension to context free grammars, that is able to recognize indentation sensitive languages. Using such an extended grammar, there is no need to perform preprocessing. Although, this type of grammar is more powerful than the context free one, we can construct efficient parsers for it and show how to do so in Section 3.

2 Grammar

A first look at the problem of indentation suggests, that a grammar that describes an indentation sensitive language must have some concept of counting. The need for counters immediately arises, because an automaton that recognizes an indentation sensitive language must keep track of the current indentation level and count the level for each new line. Furthermore, it must be able to compare counters.

Definition 1. Let Σ be an alphabet. A *counter* C_t^n for some symbol $t \in \Sigma$ and some number $n \in \mathbb{N}$ is defined by:

1. $C_t^0 = \varepsilon$ and
2. $C_t^{n+1} = C_t^n \circ t$, where $\circ$ is the concatenation symbol.

Referring to Definition 1, a counter obviously generates a string of length n that does not contain any other symbol than t. We define an indentation sensitive grammar to be a context free grammar extended by counters.

Definition 2. An indentation sensitive grammar ISG is a 5-tuple (V, Σ, C, R, S), where

1. V is a finite set of *non-terminal* symbols,
2. Σ is a finite set of *terminal* symbols,
3. C is a set of *counters*

$$C = \{C_t^n \mid t \in \Sigma \text{ and } n \in \mathbb{N}\}$$

4. R is a finite set of *rules*, where the left-hand side is a string of zero or one counter and exactly one non-terminal and the right-hand side is a string of non-terminals, terminals, and counters, and
5. $S \in V$ is the *start symbol*.

This definition is an extension to that of a context free grammar CFG. A closer look at it suggests that it is more powerful than a CFG since a counter might be used to hold context information.

Example 2. Let us look at a simplified programming language that is generated by an indentation sensitive language. The language has a statement that requires nesting *if*, a simple statement `assignment`, and a symbol for indentation $\rightarrow$. A program may look like

```
if True:
→ if True:
→   → a = 1
```

The grammar for this language is defined by $G = (V, \Sigma, C, R, S)$, where

1. $V = \{S, \langle Stmt\rangle, \langle Simple\rangle, \langle Nesting\rangle, \langle Cond\rangle\}$
2. $\Sigma = \{$“if“, identifier, intLiteral, “ : “, “ = “, newline, $\rightarrow\}$
3. $C = \{C^n_{\rightarrow} \mid n \in \mathbb{N}\}$
4. R is defined by:

$$\begin{aligned}
S &\rightarrow C^0_{\rightarrow}\langle Stmt\rangle \mid \varepsilon \\
C^n_{\rightarrow}\langle Stmt\rangle &\rightarrow C^n_{\rightarrow}\langle Stmt\rangle C^n_{\rightarrow}\langle Stmt\rangle \mid C^n_{\rightarrow}\langle Nesting\rangle \mid C^n_{\rightarrow}\langle Simple\rangle \\
C^n_{\rightarrow}\langle Nesting\rangle &\rightarrow C^n_{\rightarrow} \text{“if“ } \langle Cond\rangle \text{ “ : “ newline } C^{n+1}_{\rightarrow}\langle Stmt\rangle \\
\langle Simple\rangle &\rightarrow \text{identifier “ = “ intLiteral newline} \\
\langle Cond\rangle &\rightarrow \text{identifier}
\end{aligned}$$

Theorem 1. *G is not context free.*

Proof. We assume that G is a context free grammar and obtain a contradiction. Let p be the pumping length for G that is guaranteed to exist by the pumping lemma. For the sake of clarity, we will abbreviate “if identifier: newline” by if and “identifier = intLiteral newline” by ass. This is no loss of generality since a sequence of terminals cannot be pumped anyway. We use the string

$$s = if \circ \rightarrow \circ if \circ \rightarrow \circ \rightarrow \circ if \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ if \circ \ldots \circ (\rightarrow)^p \circ ass$$

or in a more readable representation:

$$\begin{aligned}
s \;=\; & if \circ \\
& \rightarrow \circ\, if \circ \\
& \rightarrow \circ \rightarrow \circ\, if \circ \\
& \rightarrow \circ \rightarrow \circ \rightarrow \circ\, if \circ \\
& \vdots \\
& \underbrace{\rightarrow \circ \ldots\ldots\ldots \circ \rightarrow}_{p \text{ times}} \circ\, ass
\end{aligned}$$

Clearly s is a member of G and of length at least p. The pumping lemma states that s can be pumped, but we show that it cannot.

1. $vxy =$ "if": pumping v leads to a string like $if \circ if \ldots$ and such a string is not in of G, because if requires the next statement's indentation level to be increased by one, i.e. the next line must have an additional $\rightarrow$ symbol.

2. $vxy =$ "$if \circ \rightarrow$": pumping v, we also get a string that is not in G, if $i > 1$ (we pump more than once). The string will look like:

$$\begin{aligned} & if \circ \\ & \rightarrow \circ\ if \circ \\ & \rightarrow \circ\ if \circ \end{aligned}$$

and consequently violates the mandatory increment of the indentation level after if, too.

3. $vxy =$ " $\rightarrow$ ": pumping v again results in a string that is not in G. The string will look like:

$$if \circ \rightarrow \circ \rightarrow \circ \ldots$$

Since the indentation level of a statement following if has to be incremented by exactly one, this string is not in G, too.

4. $vxy =$ "ass": in this case condition 1 of the pumping lemma is violated if $i = 0$, because an assignment ass is required.

5. $vxy =$ "$\underbrace{\rightarrow \circ \ldots \circ \rightarrow}_{p\ times} \circ\ ass$": this string violates condition 1 if $i = 0$, because there is no successive ass. Furthermore, condition 3 is violated, because $|v| > p$.

□

2.1 Upper bounds for Counters

Property 1. *Let C_t^n be a counter. The counter's target value n is finite and bounded by the length of the input string.*

Proof. This property is obviously true because the number of counter symbols cannot exceed the input string. □

For the grammar of Example 2, we can even improve upon this upper-bound.

Theorem 2. *Let n be the length of the input string and m the number of derivations. If we allow to increase the indentation level after each derivation by at most k, then the highest possible indentation level is $O(\sqrt{n})$*

Proof. The worst case example for $k = 1$ is as follows:

$$\begin{aligned} s \quad = \quad & if \circ \\ & \rightarrow \circ\ if \circ \\ & \rightarrow \circ \rightarrow \circ\ if \circ \\ & \vdots \\ & \underbrace{\rightarrow \circ \rightarrow \circ \rightarrow \circ\ if \circ}_{m} \end{aligned}$$

The maximal indentation level $m - 1$. We can state the following equation:

$$n = \sum_{i=1}^{m} i = \frac{m * (m+1)}{2} = \frac{(m^2 + m)}{2}$$

Or for some arbitrary choice of k:

$$n = k * \sum_{i=1}^{m} i = k * \frac{(m^2 + m)}{2}$$

If we solve this quadratic equation, we get:

$$m^2 + m - \frac{2 * n}{k} = 0$$

$$\Rightarrow m = -\frac{1}{2} + \sqrt{\frac{1}{4} + \frac{2n}{k}} = \frac{1}{2} * \left(-1 + \sqrt{1 + \frac{8 * n}{k}}\right)$$

□

3 Parser

Since the initial motivation for this project was related to programming languages, we choose to solve our problem by using the rather pragmatic approach of implementing a parser, rather then constructing some automaton that is able to recognize an indentation sensitive language. Counters are a small add-on to CfG, so we tried to construct a parser that is only a slight modification of some already existing parser architecture. For a purely practical reason, we tried to implement an LL recursive descent parser, because this architecture is easier to construct and understand than those of LR parsers. From this point on, we will thus focus on this parser architecture and not further consider LR parsers.

When constructing a parser, efficiency is an important issue. However, a parser for an indentation sensitive language does not perform well per se. Some modifications of the grammar may be necessary to get rid of nondeterminism and backtracking. In the following, we will show some techniques to do this.

3.1 Eliminating left recursion

Left recursions are virtually always undesirable, because it may introduce nondeterminism. For a recursive descent parser, nondeterminism may result in an infinite recursion, thus preventing the parser form terminating. Left recursions are introduced by rules of the form

$$A \to A\alpha \mid \beta$$

In this case $FIRST(A) = \beta$, thus the choice introduces nondeterminism. Fortunately, techniques for eliminating left recursions exist. For instance, we can get around it by introducing an extra rule, as described in [3]

$$\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \varepsilon \end{aligned}$$

Example 3. Taking again Example 2, the rule

$$C^n_{\to}\langle Stmt\rangle \quad \to \quad C^n_{\to}\langle Stmt\rangle C^n_{\to}\langle Stmt\rangle \mid C^n_{\to}\langle Nesting\rangle \mid C^n_{\to}\langle Simple\rangle$$

includes left recursion. Using the above rule, we can make our grammar deterministic by splitting it to

$$\begin{aligned} C^n_{\rightarrow}\langle Stmt\rangle &\rightarrow C^n_{\rightarrow}\langle Nesting\rangle C^n_{\rightarrow}\langle Stmt\rangle' \mid C^n_{\rightarrow}\langle Simple\rangle C^n_{\rightarrow}\langle Stmt\rangle' \\ C^n_{\rightarrow}\langle Stmt\rangle' &\rightarrow C^n_{\rightarrow}\langle Stmt\rangle C^n_{\rightarrow}\langle Stmt\rangle' \mid \varepsilon \end{aligned}$$

or even simpler

$$\begin{aligned} C^n_{\rightarrow}\langle Stmt\rangle &\rightarrow C^n_{\rightarrow}\langle Nesting\rangle C^n_{\rightarrow}\langle Stmt\rangle' \mid C^n_{\rightarrow}\langle Simple\rangle C^n_{\rightarrow}\langle Stmt\rangle' \\ C^n_{\rightarrow}\langle Stmt\rangle' &\rightarrow C^n_{\rightarrow}\langle Stmt\rangle \mid \varepsilon \end{aligned}$$

It should be obvious that this grammar is deterministic and it gets even more obvious, if we look at the $FIRST$ symbols.

$$\begin{aligned} FIRST(\langle Nesting\rangle) &= \{\text{“if“}\} \\ FIRST(\langle Simple\rangle) &= \{\text{identifier}\} \\ FIRST(\langle Stmt\rangle) &= FIRST(\langle Nesting\rangle) \cup FIRST(\langle Simple\rangle) \\ &= \{\text{“if“}, \text{identifier}\} \end{aligned}$$

3.2 Counter binding

To make efficient parsing of indentation sensitive languages possible, we introduce a concept called *counter binding*. A counter C^n_t can be bounded to a variable A, which means that in further derivation steps A must not appear without this counter, i.e, the variable and its counter are treated as if they were a single variable. To enable deterministic parsing, we require that $t \notin FIRST(A)$.

Binding of a counter C^n_t and a variable A is done explicitly, if $\overline{C^n_t A}$ appears on the RHS of a rule. If such a rule is applied, only rules of the form $\overline{C^n_t A} \rightarrow \ldots$ can be used for further derivations, in particular, no rules of the form $A \rightarrow \ldots$ must be used.

The motivation for introducing counter binding is that constructing a parser can be done almost the standard way. A counter that is bounded to a variable is considered to be just another variable, while a counter that is not bounded is treated like a sequence of terminals. It may be worthwhile noting, that the number of variables is not finite in the strict sense, but may depend on the length of the input string. This is the case, because $\overline{C^n_t A}$ and $\overline{C^m_t A}$ are different variables if $n \neq m$. The set of rules, on the other hand, is really finite.

Example 4. Let us look again at Example 2 and apply counter binding to it. If we eliminate left recursion, as we did in Example 3, the grammar looks as follows

$$\begin{aligned} S &\rightarrow \overline{C^0_{\rightarrow}\langle Stmt\rangle} \mid \varepsilon \\ \overline{C^n_{\rightarrow}\langle Stmt\rangle} &\rightarrow \overline{C^n_{\rightarrow}\langle Nesting\rangle}\ \overline{C^n_{\rightarrow}\langle Stmt\rangle'} \mid C^n_{\rightarrow}\langle Simple\rangle\ \overline{C^n_{\rightarrow}\langle Stmt\rangle'} \\ \overline{C^n_{\rightarrow}\langle Stmt\rangle'} &\rightarrow \overline{C^n_{\rightarrow}\langle Stmt\rangle} \mid \varepsilon \\ \overline{C^n_{\rightarrow}\langle Nesting\rangle} &\rightarrow C^n_{\rightarrow}\text{“if“}\ \langle Cond\rangle\ \text{“ : “ newline}\ \overline{C^{n+1}_{\rightarrow}\langle Stmt\rangle} \\ \langle Simple\rangle &\rightarrow \text{identifier “ = “ intLiteral newline} \\ \langle Cond\rangle &\rightarrow \text{identifier} \end{aligned}$$

In every derivation, $\langle Stmt \rangle$ is bounded to a counter, since the binding is done immediately when deriving the start variable S. Therefore, every rule that derives $\langle Stmt \rangle$ must have $\overline{C^n_{\rightarrow} \langle Stmt \rangle}$ on its LHS.

This also applies to $\langle Nesting \rangle$ and $\langle Stmt \rangle'$, however it does not for $\langle Simple \rangle$ and $\langle Cond \rangle$. These variables are never bounded to counters and hence no special rules have to be introduced.

3.3 Eliminating Backtracking

Backtracking is a process, where the scanner is reseted to some state in the past (i.e., tokens are "unread" somehow). This is necessary, if the choice among two derivation rules cannot be made by looking at their $FIRST$ symbols. Of course, this is undesirable for parser construction.

As we will see in the following, the concept of counters adds nondeterminism to a grammar. Later on we will show how to get rid of it, which is our last step towards an efficient parser.

Consider the rule

$$\overline{C^n_t A} \quad \rightarrow \quad \overline{C^n_t B_1} \mid \overline{C^n_t B_2}$$

Obviously, we cannot assume that $FIRST(\overline{C^n_t B_1}) \cup FIRST(\overline{C^n_t B_2}) = \emptyset$. In this simple case however, the choice among the two options is rather easy if $FIRST(B_1) \cup FIRST(B_2) = \emptyset$, because we can simply delay it and try to read n t-tokens first.

In case we have

$$\overline{C^n_t A} \quad \rightarrow \quad \overline{C^{n+c}_t B_1} \mid \overline{C^n_t B_2}$$

for some $c \in \mathbb{N}_+$, the problem gets trickier. Again, we try to delay the choice to avoid backtracking. Let us first try to fetch $n + c$ t-tokens from the scanner. If the fetch fails due to some other token appearing, we would have to backtrack and reset the token stream to the first t. Obviously, that doesn't make sense since its easier to hold additional information of how much ts have already been read.

We therefore add the additional field *value* to a counter C^n_t. (It should be noted, that *value* is a variable field, while t and n are constant). *value* is initially set to zero and incremented by one, every time a t is fetched from the scanner. Let us introduce the following naming convention:

Definition 3. We say that a counter C^n_t is *applied*, if we try to fetch t-tokens until $value = n$. We say that a counter has not been fully applied if $value < n$.

Using this convention and the above example, we try to apply C^{n+c}_t and if the counter has not been fully applied, we switch to the second option and try to apply C^n_t.

Example 5. Let us come back to our example and take the resulting grammar after eliminating left recursion and introducing variable binding as shown in Example 4. Let us assume we want to derive $\overline{C^n_{\rightarrow} \langle Stmt \rangle}$ and we apply the rule

$$\overline{C^n_{\rightarrow} \langle Stmt \rangle} \quad \rightarrow \quad \overline{C^n_{\rightarrow} \langle Nesting \rangle} \; \overline{C^n_{\rightarrow} \langle Stmt \rangle'}$$

Applying the rule for $\overline{C^n_{\rightarrow}\langle Nesting\rangle}$ yields

$$\overline{C^n_{\rightarrow}\langle Stmt\rangle} \quad \Rightarrow^* \quad C^n_{\rightarrow} \text{ “if“ } \langle Cond\rangle \text{ “ : “ newline } \overline{C^{n+1}_{\rightarrow}\langle Stmt\rangle}\ \overline{C^n_{\rightarrow}\langle Stmt\rangle'}$$

An *if* statement requires at least one nested statement. Let us assume this is a simple statement, then we get

$$\overline{C^n_{\rightarrow}\langle Stmt\rangle} \quad \Rightarrow^* \quad C^n_{\rightarrow} \text{ “if“ } \langle Cond\rangle \text{ “ : “ newline } C^{n+1}_{\rightarrow}\ \langle Simple\rangle\ \overline{C^{n+1}_{\rightarrow}\langle Stmt\rangle'}\ \overline{C^n_{\rightarrow}\langle Stmt\rangle'}$$

Next we try to derive $C^{n+1}_{\rightarrow}\langle Stmt\rangle'$ and start to read $\rightarrow$ symbols. However, $\langle Stmt\rangle'$ may be derived to ε and hence we may have to reset the scanner and “unread” all the $\rightarrow$'s already read. The above technique to eliminate backtracking helps us to omit this step.

3.4 Parser construction

Before starting the actual implementation, we have to take some preprocessing steps. First we have to eliminate left recursion, bind counters to variables, and eliminate backtracking as described in Sections 3.1, 3.2, and 3.3, respectively.

Functions that derive a bounded counter, must be passed additional information of the counter state. If we have a counter C^n_t, this information includes the *symbol* to be counted t, the *target* value n, and an additional *value*. The value is used for backtracking as described in Section 3.3.

Example 6. In the following, we present the essential parts of a compiler implementation for the grammar defined in Example 2. All sample code is written in Python (the complete parser code can be found in Appendix B). As we mentioned in Section 3.3, the data structure of a counter is made up of three fields:

```
class Counter(object):
    def __init__(self, sym, target, value):
        self.sym = sym        # symbol to be counted (constant)
        self.target = target  # target value (constant)
        self.value = value    # current value (variable)
```

We now need a function for applying the counter. This function tries to fetch the symbol to be counted until the counter's value equals its target value.

```
def applyCounter(self, counter):
    while counter.target > counter.value:
        if self.current <> counter.sym:
            break             # stop on any other symbol

        self.readToken()
        counter.value = counter.value + 1
```

All parser functions that do not derive a bounded counter are implemented straight forward, so we do not mention them further. Let us look at the function for $\overline{C^n_t\langle Stmt\rangle}$, which derives a counter bounded to a variable. `boundedStatement_` is the function deriving $\overline{C^n_t\langle Stmt\rangle'}$.

```
def boundedStatement(self, counter):
    self.applyCounter(counter)

    if counter.value <> counter.target:
        self.error("Counter mismatch before statement")

    # nested statement
    if self.current == ifToken:
        self.boundedNesting(counter)

    # simple statement
    elif self.current == identifier:
        self.simple()

        # parse next statement within this level
        counter.value = 0
        self.boundedStatement_(counter)

    else:
        self.error("Expecting \'if\' or identifier")
```

Let us now turn to backtracking. Backtracking is used within *if* statements. There must be at least one statement that within the statement sequence of this *if*. For every further statement, the parser does not know in advance which level this statement belongs to until it has counted all indentation symbols. Note, that the counter for the nested statement sequence is really a new object.

```
def boundedNesting(self, counter):
    self.applyCounter(counter)

    if counter.value <> counter.target:
        self.error("Counter mismatch before if")

    if self.current == ifToken:
        self.readToken()
        self.condition()

        if self.current == colon:
            self.readToken()
        else:
            self.error("Expecting \':\'")

        if self.current == newline:
            self.readToken()
        else:
            self.error("Expecting newline")

        c = Counter(counter.sym, counter.target + 1, 0)
        self.boundedStatement(c)

        if c.value < c.target:
```

```
            # switch back to old level
            counter.value = c.value
            self.boundedStatement_(counter)
    else:
        self.error("if expected")
```

4 Conclusion

In this work, we showed that programming languages that use indentation for determining the nesting level, cannot generally be described by a context free grammar. We also presented a definition of how to extend context free grammars to solve this problem. A major focus of this work was put on describing a way to construct an efficient parser out of an indentation sensitive grammar. We presented several generally applicable techniques, and used them to construct a parser for a simplified programming language.

References

[1] G. van Rossum. *Python Reference Manual*, 2006. Available at: `http://www.python.org/`.

[2] S.P. Jones. *The Haskell 98 report*, 2002. Available at: `http://www.haskell.org/`.

[3] J.D. Ullman A.V. Aho, R. Sethi. *Compilers*. Addison Wesley, 1986.

A Lexical Analyzer (lexer.py)

```
import string

unknown = 0
error = 1
identifier = 2
integerLiteral = 3
stringLiteral = 4
assign = 5
colon = 6
indent = 7
newline = 8
eof = 9

# keywords
ifToken = 10

keywords = ['if']


def isLetter(c):
    return c in string.ascii_letters

def isDigit(c):
    return c in string.digits

def getTokenNr(keyword):
    return ifToken + keywords.index(keyword)


class Lexer(object):
    '''Lexical analyzer for a python-like syntax. This lexer
       recognizes a regular language.
    '''

    def __init__(self, filename):
        '''Init a new lexer on an input stream for FILENAME.
        '''
        self.input = open(filename, 'r')
        self.line = 1
        self.column = 0
        self.__readChar()

    def __iter__(self):
        return self

    def next(self):
        '''Get a list of tokens by reading the input stream.
        '''

        if not self.current:
            return  (eof, '')
```

```
        while self.current == ' ':
            self.__readChar()

        attr = ''
        token = unknown

        # identifier and keywords
        if isLetter(self.current):
            attr = self.current
            self.__readChar()

            while isLetter(self.current) or isDigit(self.current):
                attr = attr + self.current
                self.__readChar()

            if attr in keywords:
                token = getTokenNr(attr)
                attr = ''
            else:
                token = identifier

        # integer literals
        elif isDigit(self.current):
            attr = self.current
            self.__readChar()

            while isDigit(self.current):
                attr = attr + self.current
                self.__readChar()

            token = integerLiteral

        # assignment
        elif self.current == '=':
            token = assign
            self.__readChar()

        elif self.current == ':':
            token = colon
            self.__readChar()

        elif self.current == '\t':
            token = indent
            self.__readChar()

        elif self.current == '\n':
            token = newline
            self.__readChar()

        return (token, attr)

    def __del__(self):
        '''Perform clean up on deletion.
        '''
```

```
        self.input.close()

    def __readChar(self):
        self.current = self.input.read(1)
        if self.current == "\n":
            self.line = self.line + 1
            self.column = 0
        else:
            self.column = self.column + 1


if __name__ == '__main__':
    l = Lexer('test.py')
    for token in l:
        print token
```

B Parser (parser.py)

```
from lexer import *


class Counter(object):
    """ A counter has three fields:
        - counter symbol (constant)
        - counter target (constant)
        - counter value (variable)
        A counter is said to be satisfied, if target equals value.
    """
    def __init__(self, sym, target, value):
        self.sym = sym
        self.target = target
        self.value = value


class Parser(object):
    """ Parser for a simplifed programming language described
        by an indentation sensitive grammar.
    """

    def __init__(self, filename):
        """ Init lexer for the input file and set current token.
        """

        self.lexer = Lexer(filename)
        self.current = (0, '')
        self._readToken()

    def parse(self):
        """ Run the actual parser.
        """

        self._program()
```

```
def _readToken(self):
    """ Read a new token from scanner.
    """

    self.current = self.lexer.next()
    print "reading ", self.current

def _mark(self, msg):
    """ Print an error message and stop parsing.
    """

    raise Exception("Error at %d,%d: %s" %(self.lexer.line, self.lexer.column, msg))

def _applyCounter(self, counter):
    """ Try to read counter symbols until the counters
        value equals the counters target.
    """

    while counter.target > counter.value:
        if self.current[0] <> counter.sym:
            break

        self._readToken()
        counter.value = counter.value + 1

def _program(self):
    """ Start variable in grammar.
        S -> C_t^0 <Stmt>'
    """

    self._boundedStatement_(Counter(indent, 0, 0))

def _boundedStatement_(self, counter):
    """ C_t^n <Stmt>' -> C_t^n <Stmt> | epsilon
    """

    self._applyCounter(counter)

    if counter.value == counter.target and self.current[0] <> eof:
        self._boundedStatement(counter)

def _boundedStatement(self, counter):
    """ C_t^n <Stmt> -> C_t^n <Nesting> | C_t^n <Simple>
    """

    self._applyCounter(counter)

    if counter.value <> counter.target:
        self._mark("Counter mismatch before statement")

    print "level ", counter.target

    # nested statement
    if self.current[0] == ifToken:
```

```
            self._boundedNesting(counter)

        # simple statement
        elif self.current[0] == identifier:
            self._simple()

            # use current counter to parse next statement within this level
            counter.value = 0
            self._boundedStatement_(counter)

        else:
            self._mark("Expecting \'if\' or identifier")

    def _boundedNesting(self, counter):
        """ C_t^n <Nesting> -> C_t^n if <Cond> : \n C_t^n+1 C_t^n <Stmt>'
        """

        self._applyCounter(counter)

        if counter.value <> counter.target:
            self._mark("Counter mismatch before if")

        # nested statement
        if self.current[0] == ifToken:
            self._readToken()
            self._condition()

            if self.current[0] == colon:
                self._readToken()
            else:
                self._mark("Expecting \':\'")

            if self.current[0] == newline:
                self._readToken()
            else:
                self._mark("Expecting newline")

            c = Counter(counter.sym, counter.target + 1, 0)
            self._boundedStatement(c)

            if c.value < c.target:
                # switch back to old level
                counter.value = c.value
                self._boundedStatement_(counter)
        else:
            self._mark("if expected")

    def _condition(self):
        """ <Cond> -> identifier
        """

        if self.current[0] == identifier:
            self._readToken()
        else:
```

```
            self._mark("identifier expected")

    def _simple(self):
        """ <Simple> -> identifier = intLiteral
        """

        if self.current[0] == identifier:
            self._readToken()
        else:
            self._mark("identifier expected")

        if self.current[0] == assign:
            self._readToken()
        else:
            self._mark("Expecting \'=\'")

        if self.current[0] == integerLiteral:
            self._readToken()
        else:
            self._mark("Expecting integer")

        if self.current[0] == newline:
            self._readToken()
        else:
            self._mark("newline expected")


if __name__ == "__main__":
    import sys

    if len(sys.argv) > 1:
        p = Parser(sys.argv[1])
        p.parse()
    else:
        print "Usage: %s INFILE" % sys.argv[0]
```