CHAPTER 2

Parsing

The main ideas covered in this chapter are

lexical analysis: the identification of tokens (i.e., words) within sequences of characters.

parsing: the identification of sentence structure within sequences of tokens.

In general, the syntax of the source code for a language is called its *concrete syntax*. The concrete syntax of P_0 specifies which programs, expressed as sequences of characters, are P_0 programs. The process of transforming a program written in the concrete syntax (a sequence of characters) into an abstract syntax tree is traditionally subdivided into two parts: *lexical analysis* (often called scanning) and *parsing*. The lexical analysis phase translates the sequence of characters into a sequence of *tokens*, where each token consists of several characters. The parsing phase organizes the tokens into a *parse tree* as directed by the grammar of the language and then translates the parse tree into an abstract syntax tree.

It is feasible to implement a compiler without doing lexical analysis, instead just parsing. However, scannerless parsers tend to be slower, which mattered back when computers were slow, and sometimes still matters for very large files.

The Python Lex-Yacc tool, abbreviated PLY [**2**], is an easy-to-use Python imitation of the original `lex` and `yacc` C programs. Lex was written by Eric Schmidt and Mike Lesk [**14**] at Bell Labs, and is the standard lexical analyzer generator on many Unix systems. YACC stands from Yet Another Compiler Compiler and was originally written by Stephen C. Johnson at AT&T [**12**]. The PLY tool combines the functionality of both `lex` and `yacc`. In this chapter we will use the PLY tool to generate a lexer and parser for the P_0 subset of Python.

2.1. Lexical analysis

The lexical analyzer turns a sequence of characters (a string) into a sequence of tokens. For example, the string

```
'print 1 + 3'
```

will be converted into the list of tokens

```
['print','1','+','3']
```

Actually, to be more accurate, each token will contain the token `type` and the token's `value`, which is the string from the input that matched the token.

With the PLY tool, the types of the tokens must be specified by initializing the `tokens` variable. For example,

```
tokens = ('PRINT','INT','PLUS')
```

To construct the lexical analyzer, we must specify which sequences of characters will map to each type of token. We do this specification using regular expressions. The term "regular" comes from "regular languages", which are the (particularly simple) class of languages that can be recognized by a finite automaton. A "language" is a set of strings. A *regular expression* is a pattern formed of the following core elements:

(1) a character, e.g. `a`. The only string that matches this regular expression is `'a'`.
(2) two regular expressions, one followed by the other (concatenation), e.g. `bc`. The only string that matches this regular expression is `'bc'`.
(3) one regular expression or another (alternation), e.g. `a|bc`. Both the string `'a'` and `'bc'` would be matched by this pattern (i.e., the language described by the regular expression `a|bc` consists of the strings `'a'` and `'bc'`).
(4) a regular expression repeated zero or more times (Kleene closure), e.g. `(a|bc)*`. The string `'bcabcbc'` would match this pattern, but not `'bccba'`.
(5) the empty sequence (epsilon)

The Python support for regular expressions goes beyond the core elements and includes many other convenient short-hands, for example `+` is for repetition one or more times. If you want to refer to the actual character `+`, use a backslash to escape it. Section 4.2.1 Regular Expression Syntax of the Python Library Reference gives an in-depth description of the extended regular expressions supported by Python.

Normal Python strings give a special interpretation to backslashes, which can interfere with their interpretation as regular expressions. To avoid this problem, use Python's raw strings instead of normal strings by prefixing the string with an `r`. For example, the following specifies the regular expression for the `'PLUS'` token.

```
t_PLUS =   r'\+'
```

The `t_` is a naming convention that PLY uses to know when you are defining the regular expression for a token.

Sometimes you need to do some extra processing for certain kinds of tokens. For example, for the `INT` token it is nice to convert the matched input string into a Python integer. With PLY you can do this by defining a function for the token. The function must have the regular expression as its documentation string and the body of the function should overwrite in the `value` field of the token. Here's how it would look for the `INT` token. The `\d` regular expression stands for any decimal numeral (0-9).

```
def t_INT(t):
    r'\d+'
    try:
      t.value = int(t.value)
    except ValueError:
      print "integer value too large", t.value
      t.value = 0
    return t
```

In addition to defining regular expressions for each of the tokens, you'll often want to perform special handling of newlines and whitespace. The following is the code for counting newlines and for telling the lexer to ignore whitespace. (Python has complex rules for dealing with whitespace that we'll ignore for now.)

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore  = ' \t'
```

If a portion of the input string is not matched by any of the tokens, then the lexer calls the error function that you provide. The following is an example error function.

```
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)
```

Last but not least, you'll need to instruct PLY to generate the lexer from your specification with the following code.

```
import ply.lex as lex
lex.lex()
```

Figure 1 shows the complete code for an example lexer.

```
tokens = ('PRINT','INT','PLUS')

t_PRINT = r'print'

t_PLUS =   r'\+'

def t_INT(t):
    r'\d+'
    try:
      t.value = int(t.value)
    except ValueError:
      print "integer value too large", t.value
      t.value = 0
    return t

t_ignore  = ' \t'

def t_newline(t):
  r'\n+'
  t.lexer.lineno += t.value.count("\n")

def t_error(t):
  print "Illegal character '%s'" % t.value[0]
  t.lexer.skip(1)

import ply.lex as lex
lex.lex()
```

FIGURE 1. Example lexer implemented using the PLY lexer generator.

EXERCISE 2.1. Write a PLY lexer specification for P_0 and test it on a few input programs, looking at the output list of tokens to see if they make sense.

2.2. Background on CFGs and the P_0 grammar.

A *context-free grammar* (CFG) consists of a set of *rules* (also called productions) that describes how to categorize strings of various forms.

Context-free grammars specify a class of languages known as *context-free languages* (like regular expressions specify regular languages). There are two kinds of categories, *terminals* and *non-terminals* in a context-free grammar. The terminals correspond to the tokens from the lexical analysis. Non-terminals are used to categorize different parts of a language, such as the distinction between statements and expressions in Python and C. The term *symbol* refers to both terminals and non-terminals. A grammar rule has two parts, the left-hand side is a non-terminal and the right-hand side is a sequence of zero or more symbols. The notation `::=` is used to separate the left-hand side from the right-hand side. The following is a rule that could be used to specify the syntax for an addition operator.

(1) `expression ::= expression PLUS expression`

This rule says that if a string can be divided into three parts, where the first part can be categorized as an expression, the second part is the `PLUS` terminal (token), and the third part can be categorized as an expression, then the entire string can be categorized as an expression. The next example rule has the terminal `INT` on the right-hand side and says that a string that is categorized as an integer (by the lexer) can also be categorized as an expression. As is apparent here, a string can be categorized by more than one non-terminal.

(2) `expression ::= INT`

To *parse* a string is to determine how the string can be categorized according to a given grammar. Suppose we have the string "`1 + 3`". Both the `1` and the `3` can be categorized as expressions using rule 2. We can then use rule 1 to categorize the entire string as an expression. A *parse tree* is a good way to visualize the parsing process. (You will be tempted to confuse parse trees and abstract syntax trees. There is a close correspondence, but the excellent students will carefully study the difference to avoid this confusion.) A parse tree for "`1 + 3`" is shown in Figure 2. The best way to start drawing a parse tree is to first list the tokenized string at the bottom of the page. These tokens correspond to terminals and will form the leaves of the parse tree. You can then start to categorize non-terminals, or sequences of non-terminals, using the parsing rules. For example, we can categorize the integer "`1`" as an expression using rule (2), so we create a new node above "`1`", label the node with the left-hand side terminal, in this case `expression`, and draw a line down from the new node down to "`1`". As an optional step, we can record which rule we used in parenthesis after the name of the terminal. We then repeat this process until all of the leaves have been connected into a single tree, or until no more rules apply.

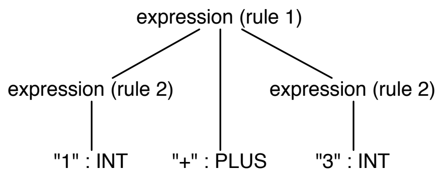


FIGURE 2. The parse tree for "`1 + 3`".

Exhibiting a parse tree for a string validates that it is in the language described by the context-free grammar in question. If there can be more than one parse tree for the same string, then the grammar is *ambiguous*. For example, the string "`1 + 2 + 3`" can be parsed two different ways using rules 1 and 2, as shown in Figure 3. In Section 2.4.2 we'll discuss ways to avoid ambiguity through the use of precedence levels and associativity.

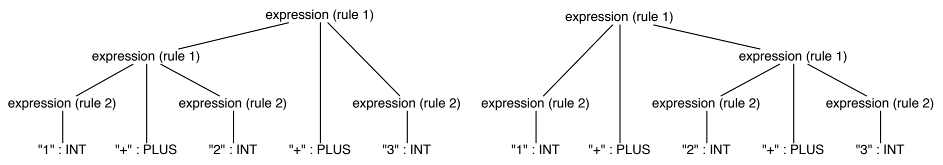


FIGURE 3. Two parse trees for "`1 + 2 + 3`".

The process described above for creating a parse-tree was "bottom-up". We started at the leaves of the tree and then worked back up to the root. An alternative way to build parse-trees is the "top-down" *derivation* approach. This approach is not a practical way to parse a particular string but it is helpful for thinking about all possible strings that are in the language described by the grammar. To perform a derivation, start by drawing a single node labeled with the starting non-terminal for the grammar. This is often the `program` non-terminal, but in our case we simply have `expression`. We then select at random any grammar rule that has `expression` on the left-hand side and add new edges and nodes to the tree according to the right-hand side of the rule. The derivation process then repeats by selecting another non-terminal that does not yet have children. Figure 4 shows the process of building a parse tree by derivation. A *left-most derivation* is one in which the left-most non-terminal is always chosen as the next non-terminal to expand. A `right-most derivation` is one in which the right-most non-terminal is always chosen as the next non-terminal to expand. The derivation in Figure 4 is a right-most derivation.

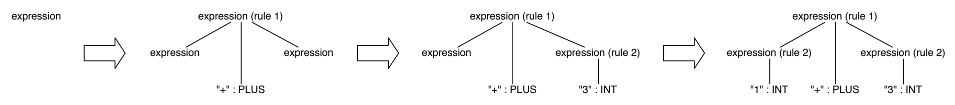


FIGURE 4. Building a parse-tree by derivation.

For each subset of Python in this course, we will specify which language features are in a given subset of Python using context-free grammars. The notation we'll use for grammars is Extended Backus-Naur Form (EBNF). The grammar for P_0 is shown in Figure 5. Any symbol not appearing on the left-hand side of a rule is a terminal (e.g., `name` and `decimalinteger`). For simple terminals consisting of single strings, we simply use the string and avoid giving names to them (e.g., `"+"`). This notation does not correspond exactly to the notation for grammars used by PLY, but it should not be too difficult for the reader to figure out the PLY grammar given the EBNF grammar.

```
program ::= module
module ::= simple_statement+
simple_statement ::= "print" expression
                   | name "=" expression
                   | expression
expression ::= name
             | decimalinteger
             | "-" expression
             | expression "+" expression
             | "(" expression ")"
             | "input" "(" ")"
```

FIGURE 5. Context-free grammar for the P_0 subset of Python.

2.3. Generating parsers with PLY

Figure 6 shows an example use of PLY to generate a parser. The code specifies a grammar and it specifies actions for each rule. For each grammar rule there is a function whose name must begin with `p_`. The document string of the function contains the specification of the grammar rule. PLY uses just a colon `:` instead of the usual `::=`

to separate the left and right-hand sides of a grammar production. The left-hand side symbol for the first function (as it appears in the Python file) is considered the start symbol. The body of these functions contains code that carries out the action for the production.

1+2+3

Typically, what you want to do in the actions is build an abstract syntax tree, as we do here. The parameter `t` of the function contains the results from the actions that were carried out to parse the right-hand side of the production. You can index into `t` to access these results, starting with `t[1]` for the first symbol of the right-hand side. To specify the result of the current action, assign the result into `t[0]`. So, for example, in the production `expression : INT`, we build a `Const` node containing an integer that we obtain from `t[1]`, and we assign the `Const` node to `t[0]`.

```
from compiler.ast import Printnl, Add, Const

def p_print_statement(t):
  'statement : PRINT expression'
  t[0] = Printnl([t[2]], None)

def p_plus_expression(t):
  'expression : expression PLUS expression'
  t[0] = Add((t[1], t[3]))

def p_int_expression(t):
  'expression : INT'
  t[0] = Const(t[1])

def p_error(t):
  print "Syntax error at '%s'" % t.value

import ply.yacc as yacc
yacc.yacc()
```

$\textcircled{1}$ $\textcircled{2}$

statement ::= PRINT expression

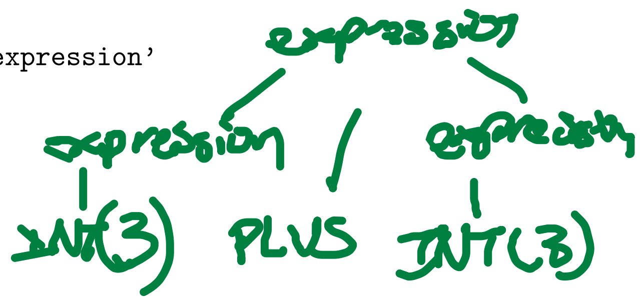


FIGURE 6. First attempt at writing a parser using PLY.

The PLY parser generator takes your grammar and generates a parser that uses the LALR(1) shift-reduce algorithm, which is the most common parsing algorithm in use today. LALR(1) stands for Look Ahead Left-to-right with Rightmost-derivation and 1 token of lookahead. Unfortunately, the LALR(1) algorithm cannot handle all context-free grammars, so sometimes you will get error messages from PLY. To understand these errors and know how to avoid them, you have to know a little bit about the parsing algorithm.

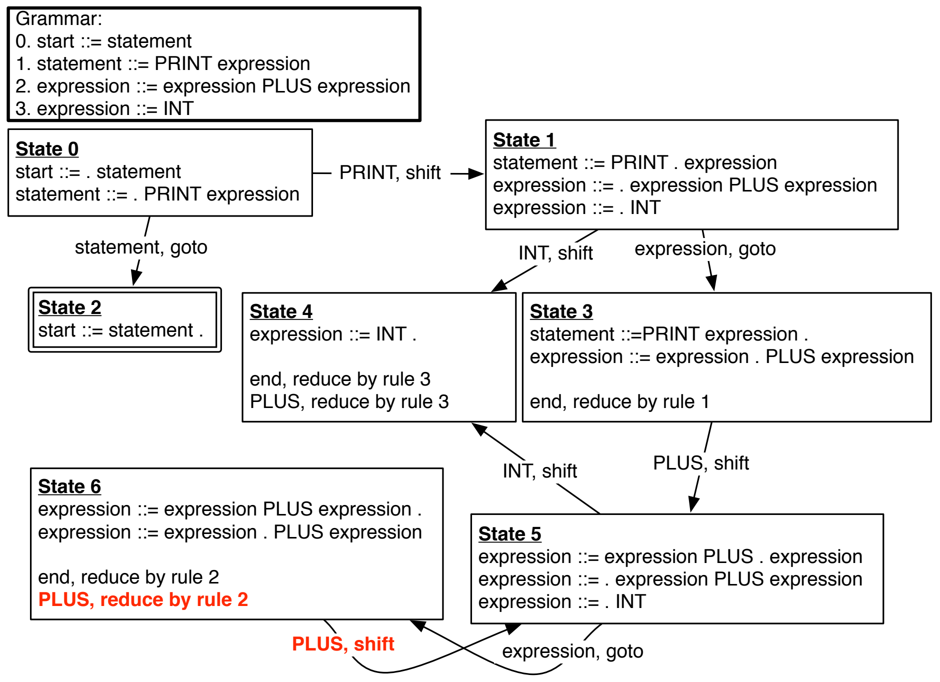
2.4. The LALR(1) algorithm

To understand the error messages of PLY, one needs to understand the underlying parsing algorithm. The LALR(1) algorithm uses a stack and a finite automaton. Each element of the stack is a pair: a state number and a symbol. The symbol characterizes the input that has been parsed so-far and the state number is used to remember how to proceed once the next symbol-worth of input has been parsed. Each state in the finite automaton represents where the parser stands in the parsing process with respect to certain grammar rules. Figure 7 shows an example LALR(1) parse table generated by PLY for the grammar specified in Figure 6. When PLY generates a parse table, it also outputs a textual representation of the parse table to the file `parser.out` which is useful for debugging purposes.

Consider state 1 in Figure 7. The parser has just read in a `PRINT` token, so the top of the stack is `(1,PRINT)`. The parser is part of the way through parsing the input according to grammar rule 1, which is signified by showing rule 1 with a dot after the PRINT token and before the expression non-terminal. A rule with a dot in it is called an *item*. There are several rules that could apply next, both rule 2 and 3, so state 1 also shows those rules with a dot at the beginning of their right-hand sides. The edges between states indicate which transitions the automaton should make depending on the next input token. So, for example, if the next input token is INT then the parser will push INT and the target state 4 on the stack and transition to state 4. Suppose we are now at the end of the input. In state 4 it says we should reduce by rule 3, so we pop from the stack the same number of items as the number of symbols in the right-hand side of the rule, in this case just one. We then momentarily jump to the state at the top of the stack (state 1) and then follow the goto edge that corresponds to the left-hand side of the rule we just reduced by, in this case `expression`, so we arrive at state 3. (A slightly longer example parse is shown in Figure 7.)

In general, the shift-reduce algorithm works as follows. Look at the next input token.

- If there there is a shift edge for the input token, push the edge's target state and the input token on the stack and proceed to the edge's target state.
- If there is a reduce action for the input token, pop k elements from the stack, where k is the number of symbols in the right-hand side of the rule being reduced. Jump to the state at the top of the stack and then follow the goto edge


Grammar:
0. start ::= statement
1. statement ::= PRINT expression
2. expression ::= expression PLUS expression
3. expression ::= INT
State 0
start ::= . statement
statement ::= . PRINT expression
PRINT, shift
State 1
statement ::= PRINT . expression
expression ::= . expression PLUS expression
expression ::= . INT
statement, goto
INT, shift
expression, goto
State 2
start ::= statement .
State 4
expression ::= INT .
end, reduce by rule 3
PLUS, reduce by rule 3
State 3
statement ::=PRINT expression .
expression ::= expression . PLUS expression
end, reduce by rule 1
INT, shift
PLUS, shift
State 6
expression ::= expression PLUS expression .
expression ::= expression . PLUS expression
end, reduce by rule 2
PLUS, reduce by rule 2
State 5
expression ::= expression PLUS . expression
expression ::= . expression PLUS expression
expression ::= . INT
PLUS, shift
expression, goto

Example parse of 'print 1 + 2'

Stack	Input	Action
[]	'print 1 + 2'	shift to state 1
[(1,PRINT)]	'1 + 2'	shift to state 4
[(1,PRINT),(4,INT)]	'+ 2'	reduce by rule 3 to state 1, goto 3
[(1,PRINT),(3,expression)]	'+ 2'	shift to state 5
[(1,PRINT),(3,expression),(5,+)]	'2'	shift to state 4
[(1,PRINT),(3,expression),(5,+),(4,INT)]	''	reduce by rule 3 to state 5, goto 6
[(1,PRINT),(3,expression),(5,+),(6,expression)]	''	reduce by rule 2 to state 1, goto 3
[(1,PRINT),(3,expression)]	''	reduce by rule 1 to state 0, goto 2
[(2,statement)]	''	accept

FIGURE 7. An LALR(1) parse table and a trace of an example run.

for the non-terminal that matches the left-hand side of the rule we're reducing by. Push the edge's target state and the non-terminal on the stack.

Notice that in state 6 of Figure 7 there is both a shift and a reduce action for the token `PLUS`, so the algorithm does not know which action to take in this case. When a state has both a shift and a reduce action for the same token, we say there is a *shift/reduce conflict*. In this case, the conflict will arise, for example, when trying to parse the input `print 1 + 2 + 3`. After having consumed `print 1 + 2` the parser will be in state 6, and it will not know whether to reduce to form an expression of `1 + 2`, or whether it should proceed by shifting the next + from the input.

A similar kind of problem, known as a *reduce/reduce* conflict, arises when there are two reduce actions in a state for the same token. To understand which grammars gives rise to shift/reduce and reduce/reduce conflicts, it helps to know how the parse table is generated from the grammar, which we discuss next.

2.4.1. Parse table generation. The parse table is generated one state at a time. State 0 represents the start of the parser. We add the production for the start symbol to this state with a dot at the beginning of the right-hand side. If the dot appears immediately before another non-terminal, we add all the productions with that non-terminal on the left-hand side. Again, we place a dot at the beginning of the right-hand side of each the new productions. This process called *state closure* is continued until there are no more productions to add. We then examine each item in the current state I. Suppose an item has the form $A ::= \alpha.X\beta$, where A and X are symbols and α and β are sequences of symbols. We create a new state, call it J. If X is a terminal, we create a shift edge from I to J, whereas if X is a non-terminal, we create a goto edge from I to J. We then need to add some items to state J. We start by adding all items from state I that have the form $B ::= \gamma.X\kappa$ (where B is any symbol and γ and κ are arbitrary sequences of symbols), but with the dot moved past the X. We then perform state closure on J. This process repeats until there are no more states or edges to add.

We then mark states as accepting states if they have an item that is the start production with a dot at the end. Also, to add in the reduce actions, we look for any state containing an item with a dot at the end. Let n be the rule number for this item. We then put a reduce n action into that state for every token Y. For example, in Figure 7 state 4 has an item with a dot at the end. We therefore put a reduce by rule 3 action into state 4 for every token. (Figure 7 does not show a reduce rule for INT in state 4 because this grammar does not allow two consecutive INT tokens in the input. We will not go into how this can be figured out, but in any event it does no harm to have a reduce rule for INT in state 4; it just means the input will be rejected at a later point in the parsing process.)

EXERCISE 2.2. On a piece of paper, walk through the parse table generation process for the grammar in Figure 6 and check your results against Figure 7.

2.4.2. Resolving conflicts with precedence declarations. To solve the shift/reduce conflict in state 6, we can add the following precedence rule, which says addition associates to the left and takes precedence over printing. This will cause state 6 to choose reduce over shift.

```
precedence = (
    ('nonassoc','PRINT'),
    ('left','PLUS')
    )
```

In general, the precedence variable should be assigned a tuple of tuples. The first element of each inner tuple should be an associativity (nonassoc, left, or right) and the rest of the elements should be tokens. The tokens that appear in the same inner tuple have the same precedence, whereas tokens that appear in later tuples have a higher precedence. Thus, for the typical precedence for arithmetic operations, we would specify the following:

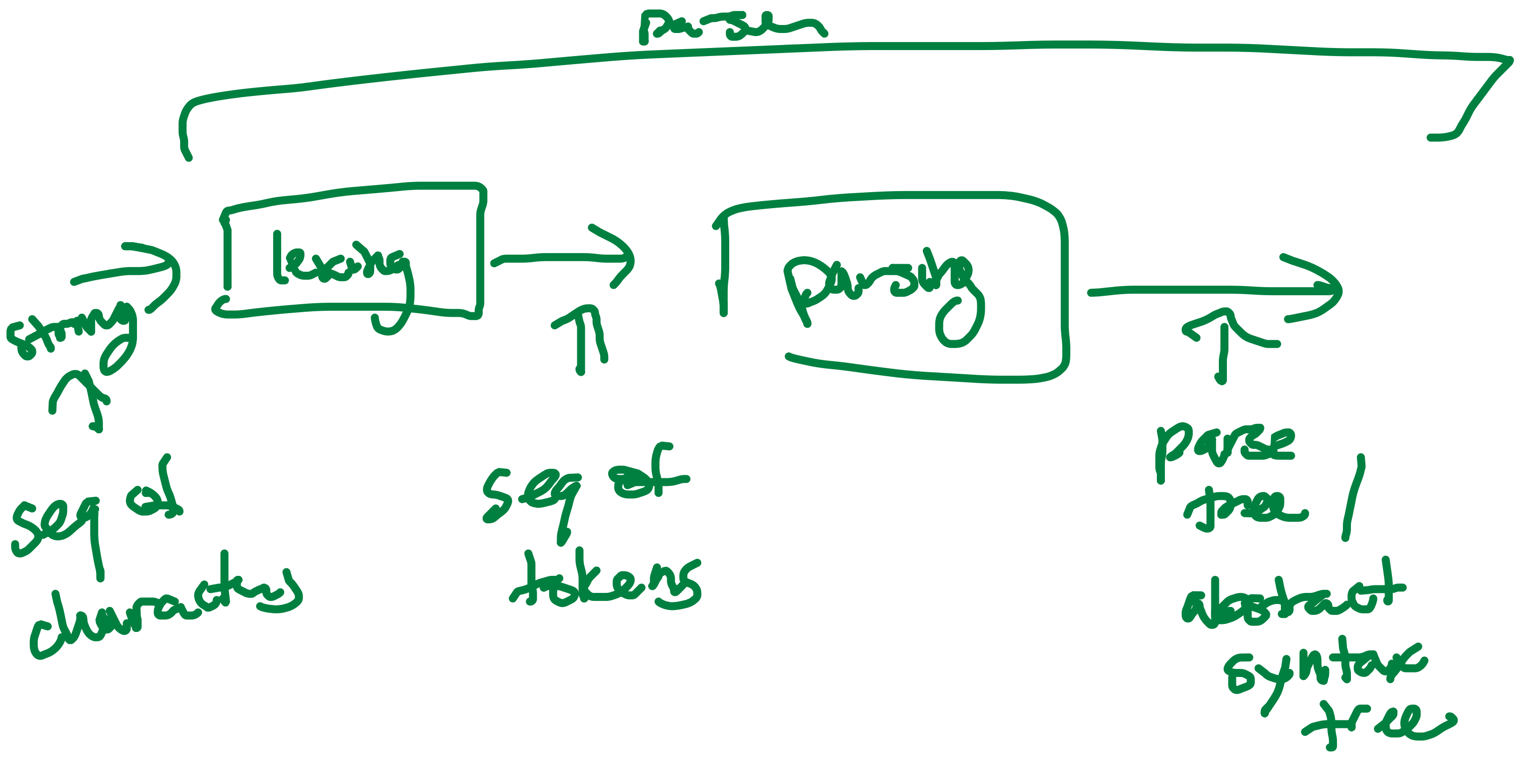
```
precedence = (
    ('left','PLUS','MINUS'),
    ('left','TIMES','DIVIDE')
    )
```

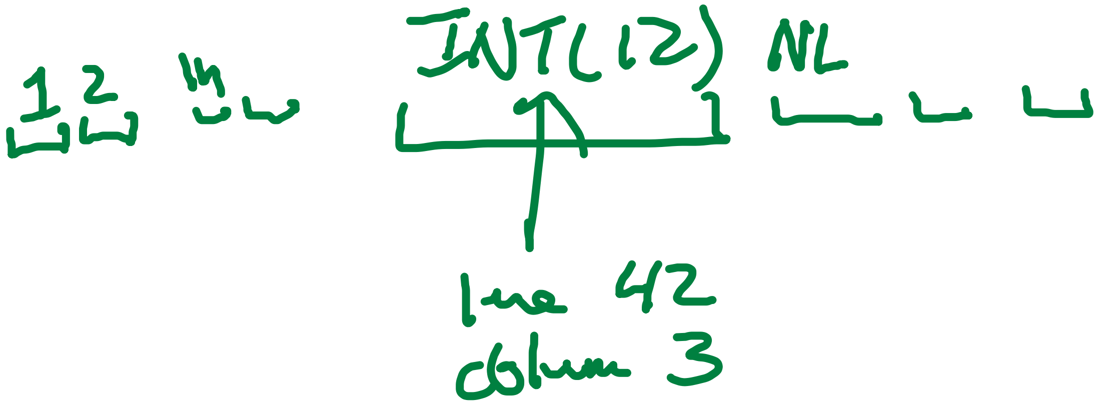
Figure 8 shows the Python code for generating a lexer and parser using PLY.

EXERCISE 2.3. Write a PLY grammar specification for P_0 and update your compiler so that it uses the generated lexer and parser instead of using the parser in the `compiler` module. In addition to handling the grammar in Figure 5, you also need to handle Python-style comments, everything following a # symbol up to the newline should be ignored. Perform regression testing on your compiler to make sure that it still passes all of the tests that you created for P_0.

```
# Lexer
tokens = ('PRINT','INT','PLUS')
t_PRINT = r'print'
t_PLUS = r'\+'
def t_INT(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print "integer value too large", t.value
        t.value = 0
    return t
t_ignore  = ' \t'
def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")
def t_error(t):
    print "Illegal character '%s'" % t.value[0]
    t.lexer.skip(1)
import ply.lex as lex
lex.lex()
# Parser
from compiler.ast import Printnl, Add, Const
precedence = (
    ('nonassoc','PRINT'),
    ('left','PLUS')
    )
def p_print_statement(t):
    'statement : PRINT expression'
    t[0] = Printnl([t[2]], None)
def p_plus_expression(t):
    'expression : expression PLUS expression'
    t[0] = Add((t[1], t[3]))
def p_int_expression(t):
    'expression : INT'
    t[0] = Const(t[1])
def p_error(t):
    print "Syntax error at '%s'" % t.value
import ply.yacc as yacc
yacc.yacc()
```

FIGURE 8. Example parser with precedence declarations to resolve conflicts.

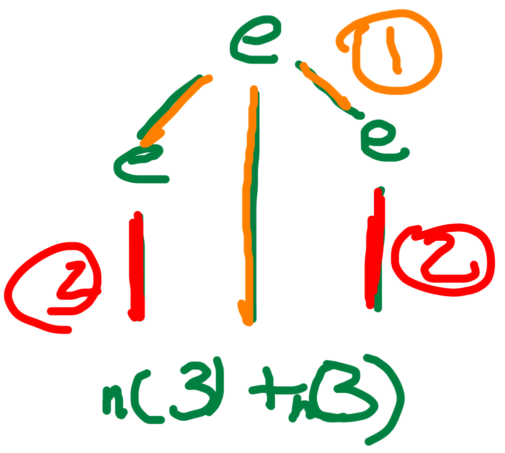


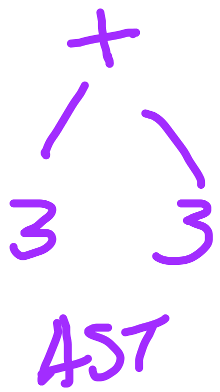


parse tree is defined by the grammar

$e ::= e + e$ ①

$\quad\mid n$ ②





CHAPTER 3

Register allocation

In chapter 1 we simplified the generation of x86 assembly by placing all variables on the stack. We can improve the performance of the generated code considerably if we instead try to place as many variables as possible into registers. The CPU can access a register in a single cycle, whereas accessing the stack can take from several cycles (to go to cache) to hundreds of cycles (to go to main memory). Figure 1 shows a program fragment that we'll use as a running example. The program is almost in x86 assembly but not quite; it still contains variables instead of stack locations or registers.

The goal of register allocation is to fit as many variables into registers as possible. It is often the case that we have more variables than registers, so we can't naively map each variable to a register. Fortunately, it is also common for different variables to be needed during different periods of time, and in such cases the variables can be mapped to the same register. Consider variables `y` and `z` in Figure 1. After the variable `z` is used in `addl z, x` it is no longer needed. Variable `y`, on the other hand, is only used after this point, so `z` and `y` could share the same register.

```
movl $4, z
movl $0, w
movl $1, z
movl w, x
addl z, x
movl w, y
addl x, y
movl y, w
addl x, w
```

FIGURE 1. An example program in pseudo assembly code. The program still uses variables instead of registers and stack locations.

3.1. Liveness analysis

A variable whose current value is needed later on in the program is called *live*.

DEFINITION 3.1. A variable is ***live*** if the variable is used at some later point in the program and there is not an intervening assignment to the variable.

To understand the latter condition, consider variable `z` in Figure 1. It is not live immediately after the instruction `movl $4, z` because the later uses of `z` get their value instead from the instruction `movl $1, z`. The variable `z` is live between `z = 1` and its use in `addl z, x`. We have annotated the program with the set of variables that are live between each instruction.

The live variables can be computed by traversing the instruction sequence back to front (i.e., backwards in execution order). Let $I_1, \ldots, I_n$ be the instruction sequence. We write $L_{\mathsf{after}}(k)$ for the set of live variables after instruction I_k and $L_{\mathsf{before}}(k)$ for the set of live variables before instruction I_k. The live variables after an instruction is always equal to the live variables before the next instruction.

$$L_{\mathsf{after}}(k) = L_{\mathsf{before}}(k+1)$$

To start things off, there are no live variables after the last instruction, so we have

$$L_{\mathsf{after}}(n) = \emptyset$$

We then apply the following rule repeatedly, traversing the instruction sequence back to front.

$$L_{\mathtt{before}}(k) = (L_{\mathtt{after}}(k) - W(k)) \cup R(k),$$

where $W(k)$ are the variables written to by instruction I_k and $R(k)$ are the variables read by instruction I_k. Figure 2 shows the results of live variables analysis for the example program from Figure 1.

Implementing the live variable analysis in Python is straightforward thanks to the built-in support for sets. You can construct a set by first creating a list and then passing it to the `set` function. The following creates an empty set:

```
>>> set([])
set([])
```

You can take the union of two sets with the `|` operator:

```
>>> set([1,2,3]) | set([3,4,5])
set([1, 2, 3, 4, 5])
```

To take the difference of two sets, use the `-` operator:

```
movl $4, z
              {}
movl $0, w
              {w}
movl $1, z
              {w, z}
movl w, x
              {x, w, z}
addl z, x
              {x, w}
movl w, y
              {x, y}
addl x, y
              {x, y}
movl y, w
              {w, x}
addl x, w
              {}
```

FIGURE 2. The example program annotated with the set of live variables between each instruction.

```
>>> set([1,2,3]) - set([3,4,5])
set([1, 2])
```

Also, just like lists, you can use Python's `for` loop to iterate through the elements of a set:

```
>>> for x in set([1,2,2,3]):
...   print x
1
2
3
```

3.2. Building the interference graph

Based on the liveness analysis, we know the program regions where each variable is needed. However, during register allocation, we'll need to answer questions of the specific form: are variables u and v ever live at the same time? (And therefore can't be assigned to the same register.) To make this question easier to answer, we create an explicit data structure, an *interference graph*. An interference graph is an undirected graph that has an edge between two variables if they are live at the same time, that is, if they interfere with each other.

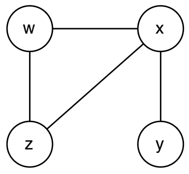


FIGURE 3. Interference graph for the example program.

The most obvious way to compute the interference graph is to look at the set of live variables between each statement in the program, and add an edge to the graph for every pair of variables in the same set. This approach is less than ideal for two reasons. First, it can be rather expensive because it takes $O(n^2)$ time to look at every pair in a set of n live variables. Second, there's a special case in which two variables that are live at the same time don't actually interfere with each other: when they both contain the same value.

A better way to compute the edges of the intereference graph is given by the following rules.

- If instruction I_k is a move: `movl` s, t, then add the edge (t, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = t$ or $v = s$.
- If instruction I_k is not a move but some other arithmetic instruction such as `addl` s, t, then add the edge (t, v) for every $v \in L_{\mathsf{after}}(k)$ unless $v = t$.
- If instruction I_k is of the form `call` $label$, then add an edge (r, v) for every caller-save register r and every variable $v \in L_{\mathsf{after}}(k)$. (The caller-save registers are `eax`, `ecx`, and `edx`.)

Working from the top to bottom of Figure 2, `z` interferes with `w` and `x`, `w` interferes with `x`, and `y` interferes with `x`. In the second to last statement, we see that `w` interferes with `x`, but we already know that. The resulting interference graph is shown in Figure 3.

In Python, a convenient representation for graphs is to use a dictionary that maps nodes to a set of adjacent nodes. So for the interference graph, the dictionary would map variable names to sets of variable names.

Consider the first instruction in Figure 2, `movl $4, z`. The L_{after} set for this instruction is the empty set, in particular, z is not live which means that this `movl` is useless, or in compiler lingo, it's dead. It is tempting to remove dead instructions during the construction of the interference graph, but in general, it's better to keep each pass focused on just one job, making it easier to debug and maintain the code. A more principled approach is to insert a pass after liveness analysis that removes dead instructions. Also, removing dead instructions may cause more instructions to become dead, so one could iterate the liveness analysis and dead-code removal passes until there is no more dead code to remove. However, to really do dead-code elimination right, one should combine it with several other optimizations, such as constant propagation, constant folding, and procedure inlining [**22**]. This combination of optimizations is a good choice of topic for your final project.

3.3. Color the interference graph by playing Sudoku

We now come to the main event, mapping variables to registers (or to stack locations in the event that we run out of registers). We need to make sure not to map two variables to the same register if the two variables interfere with each other. In terms of the interference graph, this means we cannot map adjacent nodes to the same register. If we think of registers as colors, the register allocation problem becomes the widely-studied graph coloring problem [**1, 17**].

The reader may actually be more familar with the graph coloring problem then he or she realizes; the popular game of Sudoku is an instance of graph coloring. The following describes how to build a graph out of a Sudoku board.

- There is one node in the graph for each Sudoku square.
- There is an edge between two nodes if the corresponding squares are in the same row or column, or if the squares are in the same 3×3 region.
- Choose nine colors to correspond to the numbers 1 to 9.
- Based on the initial assignment of numbers to squares in the Sudoku board, assign the corresponding colors to the corresponding nodes in the graph.

If you can color the remaining nodes in the graph with the nine colors, then you've also solved the corresponding game of Sudoku.

Given that Sudoku is graph coloring, one can use Sudoku strategies to come up with an algorithm for allocating registers. For example, one of the basic techniques for Sudoku is Pencil Marks. The idea is that you use a process of elimination to determine what numbers still make sense for a square, and write down those numbers in the square (writing very small). At first, each number might be a possibility, but as the board fills up, more and more of the possibilities are crossed off (or erased). For example, if the number 1 is assigned to a square, then by process of elimination, you can cross off the 1

pencil mark from all the squares in the same row, column, and region. Many Sudoku computer games provide automatic support for Pencil Marks. This heuristic also reduces the degree of branching in the search tree.

The Pencil Marks technique corresponds to the notion of color *saturation* due to Brélaz [**3**]. The saturation of a node, in Sudoku terms, is the number of possibilities that have been crossed off using the process of elimination mentioned above. In graph terminology, we have the following definition:

$$\mathrm{saturation}(u) = |\{c \mid \exists v. v \in \mathrm{Adj}(u) \text{ and } \mathrm{color}(v) = c\}|$$

where $\mathrm{Adj}(u)$ is the set of nodes adjacent to u and the notation $|S|$ stands for the size of the set S.

Using the Pencil Marks technique leads to a simple strategy for filling in numbers: if there is a square with only one possible number left, then write down that number! But what if there aren't any squares with only one possibility left? One brute-force approach is to just make a guess. If that guess ultimately leads to a solution, great. If not, backtrack to the guess and make a different guess. Of course, this is horribly time consuming. One standard way to reduce the amount of backtracking is to use the most-constrained-first heuristic. That is, when making a guess, always choose a square with the fewest possibilities left (the node with the highest saturation). The idea is that choosing highly constrained squares earlier rather than later is better because later there may not be any possibilities left.

In some sense, register allocation is easier than Sudoku because we can always cheat and add more numbers by spilling variables to the stack. Also, we'd like to minimize the time needed to color the graph, and backtracking is expensive. Thus, it makes sense to keep the most-constrained-first heuristic but drop the backtracking in favor of greedy search (guess and just keep going). Figure 4 gives the pseudo-code for this simple greedy algorithm for register allocation based on saturation and the most-constrained-first heuristic, which is roughly equivalent to the DSATUR algorithm of Brélaz [**3**] (also known as saturation degree ordering (SDO) [**7, 15**]). Just as in Sudoku, the algorithm represents colors with integers, with the first k colors corresponding to the k registers in a given machine and the rest of the integers corresponding to stack locations.

3.4. Generate spill code

In this pass we need to adjust the program to take into account our decisions regarding the locations of the local variables. Recall

```
Algorithm: DSATUR
Input: the inference graph $G$
Output: an assignment $\mathrm{color}(v)$ for each node $v \in G$

$W \leftarrow \mathit{vertices}(G)$
while $W \neq \emptyset$ do
  pick a node $u$ from $W$ with the highest saturation,
    breaking ties randomly
  find the lowest color $c$ that is not in $\{\mathrm{color}(v) \mid v \in \mathrm{Adj}(v)\}$
  $\mathrm{color}(u) = c$
  $W \leftarrow W - \{u\}$
```

FIGURE 4. Saturation-based greedy graph coloring algorithm.

that x86 assembly only allows one operand per instruction to be a memory access. For instance, suppose we have a move `movl y, x` where `x` and `y` are assigned to different memory locations on the stack. We need to replace this instruction with two instructions, one that moves the contents of `y` into a register and then another instruction that moves the register's contents into `x`. But what register? We could reserve a register for this purpose, and use the same register for every move between two stack locations. However, that would decrease the number of registers available for other uses, sometimes requiring the allocator to spill more variables.

Instead, we recommend creating a new temporary variable (not yet assigned to a register or stack location) and rerunning the register allocator on the new program, where `movl y, x` is replaced by

```
movl y, tmp0
movl tmp0, x
```

The `tmp0` variable will have a very short live range, so it does not make the overall graph coloring problem harder to solve. However, to prevent `tmp0` from being spilled and then needing yet another temporary, we recommend marking `tmp0` as "unspillable" and changing the graph coloring algorithm with respect to how it picks the next node. Instead of breaking ties randomly between nodes with equal saturation, give preference to nodes marked as unspillable.

If you did not need to introduce any new temporaries, then register allocation is complete. Otherwise, you need to go back and do another iteration of live variable analysis, graph building, graph coloring, and generating spill code. When you start the next iteration, do not start from scratch; keep the spill decisions, that is, which variables are spilled and their assigned stack locations, but redo the allocation for the new temporaries and the variables that were assigned to registers.

3.5. Assign homes and remove trivial moves

Once the register allocation algorithm has settled on a coloring, update the program by replacing variables with their homes: registers or stack locations. In addition, delete trivial moves. That is, wherever you have a move between two variables, such as

```
movl y, x
```

where x and y have been assigned to the same location (register or stack location), delete the move instruction.

EXERCISE 3.1. Update your compiler to perform register allocation. Test your updated compiler on your suite of test cases to make sure you haven't introduced bugs. The suggested organization of your compiler is shown in Figure 7. What is the time complexity of your register allocation algorithm? If it is greater than $O(n \log n)$, find a way to make it $O(n \log n)$, where n is the number of variables in the program.

3.6. Read more about register allocation

The general graph coloring problem is NP-complete [**6**], so finding an optimal coloring (fewest colors) takes exponential time (for example, by using a backtracking algorithm). However, there are many algorithms for finding good colorings and the search for even better algorithms is an ongoing area of research. The most widely used coloring algorithm for register allocation is the classic algorithm of Chaitin [**5**]. Briggs describes several improvements to the classic algorithm [**4**]. Numerous others have also made refinements and proposed alternative algorithms. The interested reader can google "register allocation".

More recently, researchers have noticed that the interference graphs that arise in compilers using static single-assignment form have a special property, they are chordal graphs. This property allows a simple algorithm to find optimal colorings [**8**]. Furthermore, even if the compiler does not use static single-assignment form, many interference graphs are either chordal or nearly chordal [**16**].

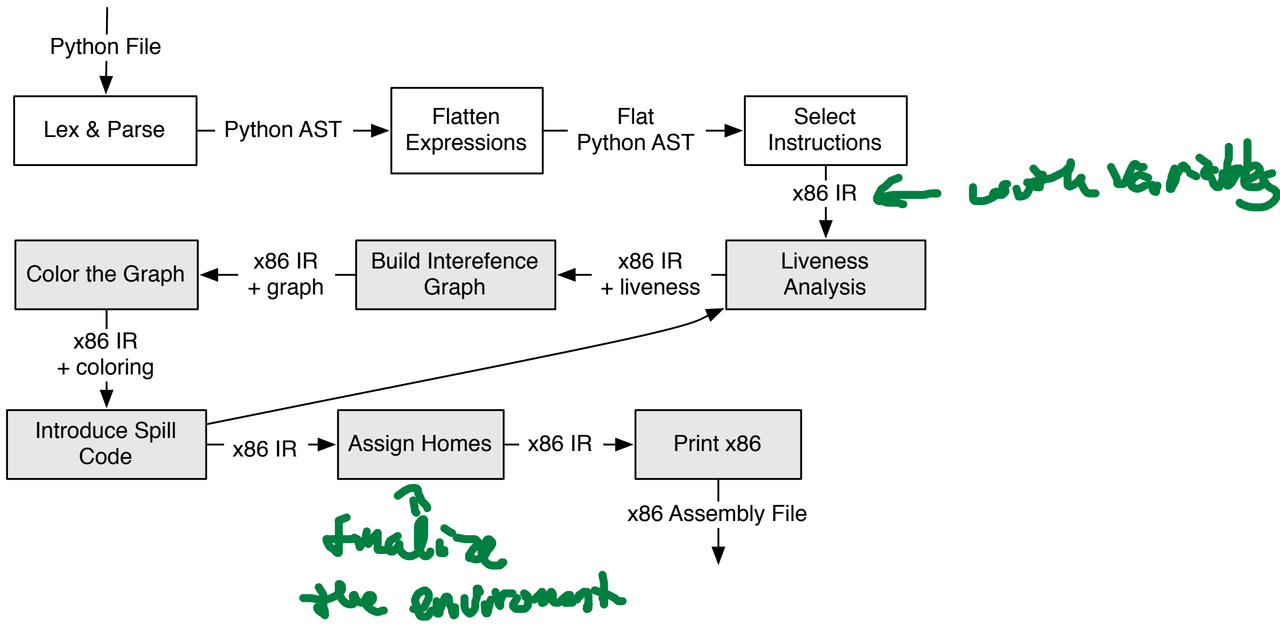


FIGURE 5. Suggested organization of the compiler.

The chordal graph coloring algorithm consists of putting two standard algorithms together. The first algorithm orders the nodes so that that the next node in the sequence is always the node that is adjacent to the most nodes further back in the sequence. This algorithm is called the maximum cardinality search algorithm (MCS) [**18**]. The second algorithm is the greedy coloring algorithm, which simply goes through the sequence of nodes produced by MCS and assigns a color to each node. The ordering produced by the MCS is similar to the most-constrained-first heuristic: if you've already colored many of the neighbors of a node, then that node likely does not have many possibilities left. The saturation based algorithm presented in Section 3.3 takes this idea a bit further, basing the choice of the next vertex on how many colors have been ruled out for each vertex.