Chapter 2

The Syntax of Syntax

Machine Code

Once upon a time programmers crafted their programs by wiring circuits to perform logic operations. This was laborious. The stored program computer[1] made it easier to reconfigure a machine for a new computation–and introduced a machine code. Thus was introduced the new tedium of machine code programming wherein the binary numbers corresponding to machine operations are calculated by hand.

Assembly Language

A mere year later[2] an enterprising chap got a computer to assemble machine code for him. The simple assembly language he created mapped letters of the alphabet to machine operations. Subsequent assemblers allow a bit more flexibility but maintain the direct correspondence of assembly statements to machine instructions.

Low Level Language

The programmer attempting to express formulas, procedures and algorithms directly in assembly language soon found him or her self saddled with unweildy programs that were difficult to debug, integrate, and reason about due to the low conceptual and semantic level of the assembly languages. To make matters worse, the programs were seldom portable from one machine architecture to another.

[1] Williams, Kilburn (1948). Electronic Digital Computers. Nature, Vol 162, p. 487

[2] Campbell-Kelly (1998). Programming the EDSAC. Annals of the History of Computing, IEEE, Vol. 20 Issue 4 p. 46-67

High Level Language

To compensate, software engineers built compilers[3] and interpreters[4] to further abstract away from the machine and to allow for portable programs that would enable the programmer to move beyond simple instructions to higher level concepts such as procedures, functions, objects, modules, first-class and higher-order 'things' of every kind.[5]

Domain Specific Language

Not only did the variety of general purpose languages multiply as new machines and new techniques of program organization arrived, many programmers found that designing a specific language[6] for a problem or domain made their task easier.

Computer scientists learned (and are still learning) quite a bit from all those languages about how to design, specify, and implement a computer language. This chapter will use just a tiny fraction of the accumulated knowledge in order to bootstrap a general computer language implementation system.

2.1 Automatically generate language compilers from BNF-style syntax + semantics. 1/20/2013 ✓

Goals, continued.

In order to accomplish the goal, this chapter is organized as follows:

Sub-projects (sections) for Chapter 2: The Syntax of Syntax.

[3] J.W. Backus, H. Herrick and I. Ziller. (1954) Preliminary Report : Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN. Programming Research Group, Applied Science Division, International Business Machines Corporation

[4] J. McCarthy. (1959) Recursive Functions of Symbolic Expressions and Their Computation by Machine. Memo 8, Artificial Intelligence Project, RLE and MIT Computation Center

[5] Many of which we introduced in the previous chapter.

[6] J. Bentley. (1986) Programming Pearls: Little Languages. Communications of the ACM, Vol. 29, No. 8, pp. 711-721

Just as with the previous chapter, this is a literate program[7] wherein regular descriptive text (such as this paragraph) is mixed with computer evaluated program text in a `monospace font`.

2.1 Syntax and the Backus-Naur Form

Early on language designers realized they needed a formal way of describing what was a validly declared (as opposed to correct or bug[8]-free) program in a particular language and what was not.

John Backus (and Peter Naur) gave us that formal method of describing[9] computer language syntax. The following example shows how Backus-Naur Form describes a syntax with simple textual grammar made of lowercase letters, numbers and keyboard-accessible symbols:

BNF, Syntax

```
<us phone number> ::= ( <three digits> ) <three digits> - <four digits>
<three digits>    ::=  <digit> <digit> <digit>
<four digits>     ::=  <digit> <digit> <digit> <digit>
<digit>           ::==  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

`(111)222-3333` satisfies the above grammer, while `(44)5555-666` and `(777) 888 - 9999` (note the spaces) do not.

This chapter will use a variant of BNF called `ibnf`, formally defined in section 2.4.

In the over-arching project of which this chapter is a part, syntax definitions (starting with an ibnf description of commonly-used character sequences) will be placed in the `indp` branch of the tree:[10]

```
`cap,env,lang,indp ~ ( [ sntx :~  ( [ common-chr :~( [ ibnf :~ $ ]; $ )]; $ )]; $);
```

[7] Donald E. Knuth, Literate Programming, Stanford, California: Center for the Study of Language and Information, 1992, CSLI Lecture Notes, no. 27

[8] Hopper, (1947) U.S. Naval Historical Center Online Library Photograph NH 96566-KN

[9] J. W. Backus, (1959) The syntax and semantics of the proposed international algebraic language of the Zuerich ACM-GRAMM conference, ICIP Paris

[10] This line of i code declares a location where our syntax definitions may reside. See chapter 1 for an introduction to i code.

2.2 Common ibnf Rules

rules An `ibnf` syntax consists of a sequence of rules. Many complete syntax definitions (often for different languages) may share a common set of rules. These rules can be collected and defined just once, then composed in a modular[11] manner. This section will define some commonly-used character sequences applicable to many grammars.

2.2 Collect common ibnf rules.	12/18/2010 ✓

Goals, continued.

The program text in this section, highligted in blue, is declared to be stored contiguously as `sntx,common-chr,ibnf` but when extracted from this document may named `common-chr.ibnf`.

digits, alternatives rule Common definitions such as identifying digits in `ibnf` are substantially the same as they would be in BNF, with alternatives separated by a vertical bar (`'|'`) and with the `'?'` indicating that the this is an alternatives rule, in which the semantic result is the semantic result of the successful alternative. The semantic result of matching a character is the character itself.

```
dgt ? '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' ;
```


uppercase The uppercase alphabet is declared the same way.

```
upr ? 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|
      'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z' ;
```

lowercase Likewise lowercase letters in the English alphabet:

```
lwr ? 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|
      'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z' ;
```

alphabetic Alphabetics are uppercase or lowercase.

[11] D.L. Parnas, (1972). On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM Vol. 15, No. 12 pp.1053 - 1058

```
alp   ? upr | lwr ;
```

Alphanumerics are uppercase, lowercase, or digits. alphanumeric

```
aln   ? upr | lwr | dgt ;
```

A hex digit is a letter A to F or a digit. hex digit

```
hex   ? dgt |'A'|'B'|'C'|'D'|'E'|'F'|'a'|'b'|'c'|'d'|'e'|'f' ;
```

There is a category for conventional character symbols. character symbols

```
smb ? '-'|'_'|'+'|'='|'`'|'~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|
      '|'|'/'|':'|';'|'*'|'('|')'|'['|']'|'{'|'}'|','|'.'|'<'|'>'|'?' ;
```

Three specific symbols are always preceded by a backslash when included in typical strings. Two more may be. Each specific symbol is described as the accumulation of the rule for matching a backslash and for matching that symbol. Accumulation rules are identified by a forward slash. escaped symbols, accumulation rule


```
sps ? bsl | btk | bqt | bnl | btb ;
bsl / '\\' '\\' ;
btk / '\\' '\'' ;
bqt / '\\' '\"' ;
bnl / '\\' 'n' ;
btb / '\\' 't' ;
```

Whitespace characters need definition too. The code in this chapter implements a scannerless parser[12] and whitespace is explicitly coded in ibnf grammars. whitespace, scannerless parser

```
wsc  ? ' ' | '\t' | '\n' ;
```

Whitespace is often made of spaces in sequence. The period in this next optional rule, recursive rule, sequences

[12] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. SIGPLAN 89, pp 170-178. ACM Press, 1989.

rule indicates an optional item in the sequence, and in this case the item refered to is the rule itself and so the rule is recursive[13], allowing a sequence of one or more spaces to be met by the rule.

```
s  /  sp .s ;
```

tabs

Spaces can be made of space characters or tab characters.

```
sp   ? ' ' | '\t' ;
```

characters in strings

A character suitable for inclusion in a string is a number, a lowercase letter, an uppercase letter, a common symbol, or a special symbol, or whitespace.

```
sch ? dgt | upr | lwr | smb | wsc | sps ;
```

strings of characters

Strings in ibnf are made of one or more suitible characters.

```
chs / sch .chs ;
```

positive integers

Positive integers are made from one or more digits in sequence. In `ibnf` the period indicates an optional element of the sequence, and rules may be recursive.


```
pnt / dgt .pnt ;
```

alphanumeric symbols

Alphanumeric 'symbols' are made of uppercase, lowercase, and numeric characters.

```
als / aln .als ;
```

[13] Wirth, Niklaus (1976). Algorithms + Data Structures = Programs. Prentice-Hall p. 126.

2.3 A Calculator Example

A calculator that can compute simple math expressions will have a syntax for those expressions. This calculator example therefore contains additional syntax definitions.

2.2 Implement a Calculator Example.	1/28/2013 ✓

Goals, continued.

The calculator example syntax is defined in `sntx,calc-example,ibnf` and is exported as `calc-example.ibnf`):.

```
`cap,env,lang,indp,sntx ~ ( [ calc-example :~( [ ibnf :~ $ ];
                                                [ six-py  :~ $ ];
                                                [ input   :~ $ ]; $ )]; $ );
```

In this calculator example the syntax is a sequence of newline-terminated lines:

```
    syntax  = line .s '\n' .syntax ;
```


Semantic actions

But calculator does not just parse expressions... It must also return results. Returning the results of what was expressed in the parsed syntax make up the semantic actions[14], of the calculator.

In this chapter another set of definitions declare what semantic action is associated with the succesful parsing of a syntax item. In this calculator example the semantics are defined in `sntx,calc-example,six-py` and are exported as `calc-example.six-py`). For convenience the semantic actions are shown in a different color. Here is the semantic declaration for the above syntax rule:

```
    syntax  ^ .1.1 + .4.1
```

The above semantic action for the syntax rule merely says that the semantic results of the first component of the syntax rule (e.g. the result of the line rule) is concatenated with the fourth component of the syntax rule (e.g. the result of

[14] Dijkstra, Edsger W. (1976.) The characterization of semantics (A Discipline of Programming, Chapter 3) Prentice-Hall.

recursively applying the syntax rule.)

For the syntax of a line, one may include a variable assignment but will include a negatable expression:

```
line  = .var nexpr ;
```

```
line  : registers[ .1.1.1 ] = int( .2.1 )
      |  --> ` .2.1 `
```

For the semantics of a line, a register will be assigned the value of the negatable expression and then a textual string containing the value of the negatable expression will be returned to be accumulated by the syntax rule.

Implementation Language

The semantic expressions for these syntax rules are a pidgin[15] of 'semantics for ibnf expressions' mixed with the native syntax of another programming language used for implementation such as (in this case) python. In the above examples the references (e.g. .1.1 or .4.1) are references to sematic products of the 'six' parsing engine, while the funcitons, assignments, and operations (e.g. int(), =, +) are borrowed from the implementation language.

Parsing Engine, Syntax Tree

The ibnf/six system is a regularly constructed parsing engine that consumes a source text under the direction of an ibnf grammer text and produces a result. Each time a syntax rule successfully parses all of its sub rules, the collection of the sub rule results is passed as a tuple to the semantic action of the rule. The result of the semantic action is returned from the rule to be accumulated in a tuple at the next level up. The natural intermediate representation of this behaivor is a syntax tree.

For convenience there are three modes of semantic production, each seen once above. The caret indicates that a referenced item or the result of an operation (which may be on referenced items) should be returned. The colon indicates that a computation will occur before a result is returned. A sequence of lines with vertical bars declare quoted text to be returned, with back-ticks unescaping expressions embedded in the quoted text.

Continuing the example:

negatable

A negatable expression is one which may be negated, and the (optional) negation is identified with a '-' sign.

[15] Holm, John (2000), An Introduction to Pidgins and Creoles, Cambridge Univ. Press.

```
nexpr   = .s .neg expr ;
neg     ? '-' ;
```

```
nexpr   ^ (str(0 - int(.3.1)) if(len(.2) > 1 and .2.1 == "-") else .3.1 )
```

A variable assignment is just a lowercase letter followed by the equals sign:

```
var     = .s lwr .s '=' ;
```

```
var     ^ .2
```

An expression may be an addition, a subtraction, an multiplication, a division, or an item all by itself. expression

```
expr    ? addexpr | subexpr | mulexpr | divexpr | itmexpr ;
```

The addition expression (with optional white space) is two items joined by a '+' sign. Likewise the subtraction, multiplication, and division. addition subtraction multiplication division

```
addexpr = .s itm .s '+' .s itm ;
subexpr = .s itm .s '-' .s itm ;
mulexpr = .s itm .s '*' .s itm ;
divexpr = .s itm .s '/' .s itm ;
```

```
addexpr ^ str(int(.2.1) + int(.6.1))
subexpr ^ str(int(.2.1) - int(.6.1))
mulexpr ^ str(int(.2.1) * int(.6.1))
divexpr ^ str(int(.2.1) / int(.6.1))
```

The item expression is just an item. An item may be a variable identifier, a positive integer or a parenthetical negatable expression. item expression, item, variable

```
itmexpr = .s itm  ;
itm     ? vbl | pnt | parens ;
```

```
itmexpr ^ .2.1
```

variable

A variable is just a lowercase letter, but when the sytnax is matched in an expression the semantic action is to retrieve the previous value recorded for that variable.

```
vbl     = lwr ;
```

```
vbl     ^ str( registers[ .1.1 ] )
```

parentheses

Parentheses surround a negatable expression. Nesting is thus allowed.

```
parens  = .s '(' nexpr .s ')' ;
```

```
parens ^ .3.1
```

A calculator needs input:

```
a=1+34
b=a + 40
c = 56-6
 d = b - c
x = 10 / 2
v = x - 12
y = - ( 5 - 3 )
z = -80
e = (3 + 5 ) / ( 2 + 2 )
f = - 1
```


To build the example, first run the semantic rule parser on the calculator example semantic rules:

```
python sixparser.py calc-example.six-py blank.txt calc-example.smtx
```

Combine the grammars and then run the syntax rule parser on the calculator example syntax rules and include the generated semantics as a parameter:

```
cat ./common-chr.ibnf ./calc-example.ibnf > ./calc-example-full.ibnf
python ibnfmeta.py calc-example-full.ibnf calc-example.smtx calc-example.py
```

Run the calculator on the input and view the output:

```
python calc-example.py calc-example.input blank.txt calc.out
cat calc.out

 --> 35
 --> 75
 --> 50
 --> 25
 --> 5
 --> -7
 --> -2
 --> -80
 --> 2
 --> -1
```

2.4 Declaring ibnf in ibnf/six

Annotating rules with productions (as shown in the calculator example) enables the system to automatically construct an interpreter or even a compiler for input texts written in the described syntax. When the described syntax is a system for describing syntax (e.g. `ibnf.ibnf`) and when the semantics are those of semantics-generation (as will be seen in `ibnf.six-py`, `six.ibnf` and in `six.six-py`) then the result is a parser-generator or a compiler-compiler[16]. compiler-compiler

If the compiler-compiler thus described is sufficiently versatile to reproduce its own executable when provided with its own syntax and semantics definitions then the result is a metacompiler[17]. `ibnf/six` is a metacompiler. meta-compiler

2.3 Declare ibnf in ibnf/six.	1/15/2013 ✓

Goals, continued.

`ibnf` will use the common definitions introduced in the first section and introduces more defined in `sntx,ibnf,ibnf` and exported as `ibnf.ibnf`. The semantic rules for parsing ibnf are defined in `sntx,ibnf,six-py` and are exported as `ibnf.six-py`.

```
`cap,env,lang,indp,sntx ~ ( [ ibnf :~( [ ibnf :~ $ ];
                                       [ six-py  :~ $ ]; $ )]; $ );
```

[16] Stephen C. Johnson. YACC: Yet Another Compiler-Compiler. Unix Programmer's Manual Vol 2b, 1979.

[17] Schorre (1964). A Syntax-Oriented Complier Writing Language. Proceedings of the 1964 19th ACM National Conference, ACM Press, New York, NY, 41.301-41.3011

PEG, packrat

The `ibnf/six` system implements a parsing expression grammer, or PEG[18]. A PEG imposes an order of selection on what might otherwise be an ambiguous choice in a BNF-style grammer– in a PEG the first option is selected when there are two possible matches. In addition the `ibnf/six` system implements a packrat[19]. parser, using memoization of previously visited states as an optimization for efficiency and for handling recursive rules.

The result of processing the syntax and semantics rules in this section will be an `ibnf` syntax parser, which is one half of the `ibnf/six` system. Crafting a `six` syntax parser with its own semantic rules will be the subject of the section following this one.

To begin with, a syntax is made of rules.

```
syntax = rules ;
```

```
syntax   ^ prologue + .1.1 + semantics + epilogue
```

The semantic action of the syntax rule is to produce a program of four parts: a prologue that sets up the parsing engine initial conditions, a set of function definitions (each corresponding to a defined rule in the syntax being parsed,) the semantic actions for the syntax being parsed and an epilogue that performs the final actions of the parsing engine before it exits. In this implementation of `ibnf/six` the result is a python program.


```
rules = rule .rules ;
```

```
rules   ^ .1.1 + .2.1
```

The function definitions that result from successfuly parsed rules are simply concatenated into a sequence of function definitions.

A rule may incorporate other rules, may be a choice amongst alternatives or may just be a blank line:

[18] Ford, Bryan (2004). Parsing Expression Grammars: A Recognition Based Syntactic Foundation. Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

[19] Warth, Douglass, Millstein (2008). Packrat Parsers Can Support Left Recursion. ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation

```
rule ? incorp | altern | blankline ;
```

Blank lines have optional whitespace and end in a newline character. Blank lines provide nothing to the parsing program that is being assembled.

```
blankline = .s '\n' ;
```

```
blankline  ^ ""
```

Each rule type has a name and a body. The alternating rule is signified by a question mark.

```
altern =  .s name .s '?' albody  .s ';' .s '\n'  ;
```

The semantic action of matching an 'altern' rule is to compose a new function that takes an input text and indicates a match was found if that input text starts with one of the alternatives listed in the body of the 'altern' rule.

```
altern  |def `.2.1`_p( s, c):
        |  if been("`.2.1`",s): return was( c, "`.2.1`",s)
        |  else:
        |    mark("`.2.1`",s,(F,T,s,0,c,("","")));met = F
        |`#.5#`
        |    if not met:
        |      return mark("`.2.1`",s,(F,T,s,0,c,("","")))
        |    else:
        |      return mark("`.2.1`",s,(met,mem,s,tl,tc,ta))
.albody ^ #.1.2#  + #.1.3.1#
.cmatch ^ #.1#
.cm     |    if not met: (met,mem,ts,tl,tc,ta)=cm('`.1`',s,c)
.btb    |    if not met: (met,mem,ts,tl,tc,ta)=cm('`.1`',s,c)
.bnl    |    if not met: (met,mem,ts,tl,tc,ta)=cm('`.1`',s,c)
.name   |    if not met: (met,mem,ts,tl,tc,ta)=`.1`_p(s,c)
```

The first thing the produced function does is check to see if the same position in the source text has already been evaluated for matching that syntax rule. If so the function returns the success or failure previously determined.

If that location in the source text has not been evaluated for matching the syntax rule before, the generated function marks the location as failing the

match and sets a flag (for having met at least one rule) to False.

In the process of creating the generated function, the 'altern' rule next recurses on the 5th component of the rule, which is the 'albody'. Upon recursion one of six things will be included in the genrated function: a call to another named rule, a call to matching a newline, a call to matching a tab, a call to matching a character, a call to a superior rule to matching characters, or the results of composing recursive calls to the 'albody' rule.

Finally in the genrated function if nothing was matched then failure is returned, otherwise success is both memoized and returned.

The incorporating rule is signified by either a forwards slash or an equals sign:

```
incorp =  .s name .s iflag inbody  .s ';' .s '\n'  ;
iflag  ? '/' | '=' ;
```

The semantic action of matching an 'incorp' rule is to compose a new function that takes an input text and indicates if matches were found sequentially in the input text for all of the items listed in the body of the 'incorp' rule.

```
incorp  :smfnc="_s(a,m,s,e,c,n): return(T,T,s,e-s,c,(n,fi[s:e]))"
        |def `.2.1`_p( s, c):
        |  if been("`.2.1`",s): return was( c, "`.2.1`",s)
        |  else:
        |    mark("`.2.1`",s,(F,T,s,0,c,("","")))
        |    ok=True; ts=s; tl=0; a={0: ("","")}
        |    mem={0:True}; tc=c; n=0
        |`#.5#`
        |    if ok:
        |      rv=`.2.1`_s(a,andmemo(mem),s,ts+tl,tc,"`.2.1`")
        |      return mark("`.2.1`",s,rv)
        |    return mark("`.2.1`",s,(F,T,s,0,c,("","")))
        |`("def "+.2.1+smfnc if .4.1 == "/" else "")`
.inbody |    if ok:
        |      n=n+1; ( `(  "n" if .1.2.0 == "pnit" else "" )`ok,mem[n],ts,tl,tc,a[n])=\
        |      `#.1.2.1 if .1.2.0=="pnit" else .1.2#`
        |`#.1.3#`
.pnit   ^ "n"
.cmatch ^ #.1#
.name   ^ .1 +"_p ( (ts+tl), tc)"
.cm     ^ "cm(\'" + .1 + "\',(ts+tl), tc)"
.bsl    ^ " cm(chr(92) ,(ts+tl), tc)"
.btk    ^ " cm(chr(39) ,(ts+tl), tc)"
.bqt    ^ " cm(chr(34) ,(ts+tl), tc)"
.bnl    ^ " cm(chr(10) ,(ts+tl), tc)"
```


As with the previous rule, the first thing the produced function does is check to see if the same position in the source text has already been evaluated for matching that syntax rule. If so the function returns the success or failure previously determined.

If this is a new evaluation, the generated function marks the location as failing the match and sets a flag (for not having failed a rule yet) to True.

In the process of creating the generated function, the 'incorp' rule next recurses on the 5th component of the rule, which is the 'inbody'. Upon recursion one of nine things will be included in the genrated function: a call to another named rule, a call to matching a newline, a call to matching a tab, a call to matching a single quote, a call to matching a backslash, a call to matching a character, a call to a superior rule to matching characters, a call to matching a period (optional) named item, or the results of composing recursive calls to the 'inbody' rule.

Finally in the genrated function if everything was matched then the semantic function is called, after which success is both memoized and returned. Otherwise

in the generated function failure is memoized and returned.

If the rule being generated was marked with a slash then the incorp rule will generate an additional semantic function for the rule that simply returns the extent of the source text matched as the result of that rule's semantic action.

A name is made up of one or more lowercase letters. The semantic action of matching the name rule is to return the letters making up the name.

```
name / lwr .name ;
```

Within the rule `albody` rule-named-items are separated by the vertical bar symbol. More alternatives may be found on the next line if the last thing on the albody line is a vertical bar symbol. For a semantic action, the albody rule returns the complete tuple of values available to it. The almore semantic action however is to return the fourth item in the tuple of values available to it (e.g. the result of its 'albody'.

```
albody = .s nit .almore ;
almore = .s '|' .alnewline albody ;
alnewline / .s '\n' ;
```

```
albody  ^ ..
almore  ^ .4
```


Within the rule `inbody` rule there may be multiple possibly period prefixed named items. Like albody, inbody passes upwards everything it gets.

```
inbody = .s onit .inbody ;
onit ? pnit | nit ;
```

```
inbody ^ ..
```

pnits are:

```
pnit = '.' nit ;
```

```
pnit ^ .2
```

A nit is a name or a character match.

```
nit ? name | cmatch ;
```

A character match is a suitable character (or escaped character) as defined in common-chr surrounded by single quotes.

```
cmatch = '\'' sch '\'' ;
```

```
cmatch ^ .2
```

To build the ibnf parser, first run the semantic rule parser on the ibnf semantic rules:

```
# python sixparser.py ibnf.six-py blank.txt ibnf-smtx.py
```

Combine the grammars and then run the syntax rule parser on the ibnf syntax rules and include the generated semantics as a parameter:

```
# cat ./common-chr.ibnf ./ibnf.ibnf > ./ibnf-full.ibnf
# python ibnfmeta.py ibnf-full.ibnf ibnf-smtx.py ibnf.py
```


Run the generated parser on the very input that produced it, then do it again:

```
# python ibnf.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta2.py
# python ibnfmeta2.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta3.py
```

2.5 Implementing six in ibnf/six

The previous section implemented a parser of syntax rules that allowed for the inclusion of externally generated semantic productions for those rules. This section implements a parser and generator for those semantic rules. The separation of syntax from semantics allows syntax to be re-used and semantics to be re-targeted.

2.4 Declare the syntax and semantics of six.	1/20/2013 ✓

Goals, continued.

`six` will use the common definitions introduced in the first section and introduces more defined in `sntx,six,ibnf` and exported as `six.ibnf`. The semantic rules for parsing six are defined in `sntx,six,six-py` and are exported as `six.six-py`.

```
`cap,env,lang,indp,sntx ~ ( [ six :~( [ ibnf :~ $ ];
                                      [ six-py  :~ $ ]; $ )]; $ );
```

The result of processing the rules of sytax for six will be an six parser.

```
syntax = srules end ;
```

```
syntax ^ .1.1 + .2.1
```

The six parser consumes semantic rules and (as its own semantic function) emits the results of processing those rules into semantic functions, followed by some predefined values that may be used by semantic functions.

The srules rule is defined recursively:


```
srules = srule .srules ;
```

```
srules ^ .1.1 + .2.1
```

The values of the rules (as semantic functions for inclusion in an ibnf parser) are returned concatenated together by the six parser.

A srule is a blank line or a basic semantic definition.

```
srule ? blankline | base ;
```

Blank lines have optional whitespace and end in a newline character.

```
blankline = .s '\n' ;
```

```
blankline  ^   ""
```

A basic semantic definition has a name followed by some optional setup code, a body of program code that is produced when the syntax for that named syntax component is matched by the parser, and then an optional sequence of recursive definitions that may be refrenced by that body of code.

```
base   =  .s name .setup body .recr;
body   ?  qlineset | cline  ;
setup  = .s ':' .s code .s '\n' .setup ;
recr   = .s '.' name .rsetup body .recr ;
rsetup = .s ':' .s rcode .s '\n' .rsetup ;
rcode  = ritm .rcode ;


ritm ?     string |  rcr | lwr | dpathw | dhas | pnt | '>' | '<' |
       '{' | '}' | ':' | '%' | '(' | ',' | ')' | '_' | '[' | ']' |
                   ';' | '+' | '-' | '*' | '/' | '=' | '!' | ' ' ;
```


The body of program text may be a set of 'quoted' text lines to be deposited as the resulting function's return value or it may be a singe code line, and the result of executing that code line is deposited as the resulting function's return value.

The semantic implementation of the basic semantic rule declares a semantic function which conditionally includes setup code for a rule being parsed by that rule, will include 'body' code for returning a result for a rule being parsed by that rule, and which conditionaly is followed by a recursive function definition for a semantic rule being parsed by that rule:

```
base  :rbody="  o = \"\"\n  rx="+.2.1+"_r\n  if a != \"\":\n"+.5.1+"  return (o)\n"
      |def `.2.1`_s(a,m,s,e,c,n):
      |`("  rx="+.2.1+"_r "  if .5.1 != "" else "")`
      |`("  "+.3.1 if .3.1 != "" else "")`
      |  return (T,T,s,e-s,c,( "`.2.1`", `.4.1` ))
      |`("def "+.2.1+"_r(a,m,s,e,c,n):\n"+rbody if .5.1 != "" else "")`

setup  ^ ( ( .4.1 + "\n  " + .7.1) if .7.1 != "" else .4.1 )

rsetup ^ ( ( "      "+.4.1 + "      " + .7.1+"\n") if .7.1 != "" else "      "+.4.1+"\n" )

recr   ^ "    if a[0] ==\"" + .3.1 + "\":\n" + .4.1 + "      o=o+" + .5.1 +"\n"+ .6.1
```

A code line starts with a caret:

```
cline    =  .s '^' .s code .s '\n' ;
```

```
cline    ^ .4.1
```

A set of quoted lines have a vertical bar at the start of each:

```
qlineset =  qlines ;
qlines   =  .s qlsep qline .qlines ;
qlsep    =  '|' ;
qline    =  qchs '\n' ;
```

```
qlineset ^ "\"" + .1.1 + "\""
qlines   ^ .2.1 + .3.1 + .4.1
qlsep    ^ "\\n\" + \\\n\""
qline    ^ .1.1
```

A code line may be escaped with back-tics. The result of code within the back-tics is inserted in-line with the quoted text:

```
qchs   = .qch .qchs ;
qch    ? aln | qq | qt | qs | qsmb | ' ' | qcode  ;
qq     = '\"' ;
qt     = '\'' ;
qs     = '\\' ;


qcode = '`' .s code .s '`' ;
```

```
qchs   ^ .1.1 + .2.1
qq     ^    "\\\""
qt     ^    "\\\'"
qs     ^    "\\\\"
qcode  ^ "\" + " + .3.1 + " + \""
```

A name is composed of lowercase letters:

```
name / lwr .name ;
```

Quoted symbols may not include the back-tick, because that is used to escape the quoted text:

```
qsmb   ? '-'|'_'|'+'|'='|'~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|'!'|'|'|'/'|
           ':'|';'|'*'|'('|')'|'['|']'|'{'|'}'|','|'.'|'<'|'>'|'?' ;
```


Strings may be interrupted by newlines and continued on the next line after a continuation character:

```
string  =  '\"' .strcs '\"' ;
strcs   =  sch .strcs ;
```

```
string  ^  "\"" + .2.1 + "\""
strcs   ^  .1.1 + .2.1
```

six-py is a pidgin of six syntax for data reference paths and recursive calls, and of python code. The pidgin is constructed of code item(s) and may span multiple lines when the newline is followed by a code continuation character:

```
code = citm .code ;
citm ? string | cnl | rcr | lwr | dpathw | dhas | pnt | '>' | '<' |
       '{' | '}' | ':' | '%' | '(' | ',' | ')' | '_' | '[' | ']' |
                   ';' | '+' | '-' | '*' | '/' | '=' | '!' | ' ' ;
cnl = '\n' .s '^' ;
```

```
code     ^    .1.1  + .2.1
cnl      ^    "\\\n"
```

Data reference paths are turned into python array dereferences:

```
dpathw   =    dpath ;
dpath    =    '.' pnt .dpath ;
dhas     =    '.' '.' ;
```

```
dpathw   ^    "a" + .1.1
dpath    ^    "[" + .2.1 + "]" + .3.1
dhas     ^    "a"
```

Recursive calls are made between hash marks:

```
rcr   ?    rca | rcb ;
rca   =    '#' .s name .s ':' .s code .s '#' ;
rcb   =    '#' .s code .s '#' ;
```



```
rca   ^    .3.1 +"_s(" + .7.1 + ",m,s,e,c,n)[5][1]"
rcb   ^    "rx(" + .3.1 + ",m,s,e,c,n)"
```

The parsing engine assembled by the ibnf/six system requires some predefined functionality, which is encapsulatd in six in definitions for a prologue and an epilogue and which is placed in the semantic output after all of the semantic rules. The syntax rule for 'end' optionally matches whitespace, which will always succeed.

```
end   =  .s  ;
```

The prologue and epilogue are included as simple global variable assignments that may be referred to in other sematic productions (such as the production for 'syntax' in ibnf.)

```
end |prologue="""import sys
    |from binascii import *
    |fi = file(sys.argv[1]).read()
    |semantics = file(sys.argv[2]).read()
    |fo = open(sys.argv[3], "w+")
    |
    |h={}; registers={}; context={}; mseq=0; dseq=1; T=True; F=False
    |
    |def n2z( a ):
    |  return ( '0' if a=='' else a )
    |
    |def be2le( a ):
    |  return a[6:8]+a[4:6]+a[2:4]+a[0:2]
    |
    |def mark( p, s, t ):
    |  ( v, m, ss, l, c, a ) = t
    |  if t[1]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
    |  else:
    |    if not t[0]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
    |  return t
    |
    |def been(p, s):
    | if h.has_key( p +"-" + str(s) ): return h[p +"-" + str(s)][1]
    | else:  return False
    |
    |def was(c,p,s): (v,m,l,a) = h[p+"-"+str(s)]; return (v,m,s,l,c,a)
    |
    |def cm( ch, s, c ):
    |  if s < len(fi):
    |    if fi[s] == ch:    return ( T, T, s, 1,  c, ( "cm", fi[s] ) )
    |  return ( False, True, s, 0, c, ( "cm", "") )
    |
    |def andmemo( m ):
    |  r = True
    |  for i in m:
    |    if not m[i]: r = False
    |  return r
    |
    |outdata = ""
    |
    |def output( s ):
    |  global outdata
    |  outdata = outdata + str(s)
    |
    |"""; epilogue="""
    |
    |(v,m,s,l,c,a) = syntax_p( 0, ({},'<1>','<0>') )
    |if v:
    |  print "Parsed "+a[0]+" OK"
    |else: print "Failed to Parse"
    |print >> fo, a[1]
    |fo.close()
    |"""
```

To build the six parser, run the semantic rule parser on the six semantic rules:

```
# python sixparser.py six.six-py blank.txt six-smtx.py
```

Combine the grammars and then run the syntax rule parser on the six syntax rules and include the generated semantics as a parameter:

```
# cat ./common-chr.ibnf ./six.ibnf > ./six-full.ibnf
# python ibnfmeta.py six-full.ibnf six-smtx.py sixmeta.py
```

Run the generated parser on the very input that produced it, then regenerate the semantics parser:

```
# python sixmeta.py six.six-py blank.txt six-smtx2.py
# python ibnfmeta.py six-full.ibnf six-smtx2.py sixmeta2.py
```

Rebuild the ibnf parser using the new semantic rule parser:

```
# python sixmeta2.py ibnf.six-py blank.txt ibnf-smtx.py
```

Combine the grammars and then run the syntax rule parser on the ibnf syntax rules and include the generated semantics as a parameter:


```
# cat ./common-chr.ibnf ./ibnf.ibnf > ./ibnf-full.ibnf
# python ibnfmeta.py ibnf-full.ibnf ibnf-smtx.py ibnf.py
```

Run the generated parser on the very input that produced it, then do it again:

```
# python ibnf.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta2.py
# python ibnfmeta2.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta3.py
```

2.6 A toy compiler in ibnf/six

The ibnf/six system can be used to create a compiler that produces machine code from source code in a defined language syntax.

2.6 Craft a toy compiler in ibnf/six.	2/8/2013 ✓

Goals, continued.

The toy syntax needs a place to reside.

```
`cap,env,lang,indp,sntx ~ ( [ toy :~( [ ibnf :~ $ ];
                                      [ six-py :~ $ ]; $ )]; $ );
```

The syntax of a toy program will consist of a series of statement sequence lines.

```
  syntax  =  program end ;
  program =  lines ;
  lines   = .seq .s '\n' .lines ;
  seq     = .s stmt .more ;
  more    = .s ';' seq ;
```

```
  syntax ^ header + code + literals

 program : # sizecode   : .1.1 #
         : # countvars  : .1.1 #
         : # makelit    : .1.1 #
         : # emitcode   : .1.1 #
         : print .1.1
         | ok

  lines  ^ ( "seq", .1.1 , .4.1 )
  seq    ^ ( "seq", .2.1 , .3.1 )
  more   ^ .3.1
```


The only statements for now are print, var, and exit.

```
  stmt    ?  lsprint | vsprint | vdef | exit ;
  lsprint   = 'p' 'r' 'i' 'n' 't' .s lstring ;
  vsprint   = 'p' 'r' 'i' 'n' 't' .s vname ;
  vdef     = 'v' 'a' 'r' s vname .s '=' .s varable ;
  vname   / lwr .vname ;
  exit    = 'e' 'x' 'i' 't' .s exitvalue ;
```

```
  lsprint  ^ ( "lsprint", .7.1 )
  vsprint  ^ ( "vsprint", .7.1 )
  vdef     ^ ( "vdef", .5.1, .9.1 )
  exit     ^ ( "exit",  .6.1 )
```

Only literal strings are printable right now, and one exits with an exit value.

```
  lstring = '\"' .strchs '\"' ;
  strchs  = strch .strchs ;
  strch   ? dgt | upr | lwr | smb | ' ' | '\t' | nsl | ntk | nqt | nnl | ntb ;
  nsl     = bsl ;
  ntk     = btk ;
  nqt     = bqt ;
  nnl     = bnl ;
  ntb     = btb ;
  varable   = '\"' .chs '\"' ;
  exitvalue / pnt ;
```

```
  strchs  ^ .1.1 + .2.1
  nsl     ^ "\\"
  ntk     ^ "\'"
  nqt     ^ "\""
  nnl     ^ "\n"
  ntb     ^ "\t"
  lstring  ^ ( "lstring", .2.1, str(s), str(len(.2.1)) )
  varable    ^ ( "lstring", .2.1, str(s), str(len(.2.1)) )
```



```
  end  = .s ;
```

Before we do anything else we need to know how big the executable portion of the program is:

```
sizecode : global lstart
         : lstart = int(# .. #)+232+4096+8
         | ok
 .seq    ^ str(0 + int(n2z(#.1#)) + int(n2z(#.2#)))
 .exit   ^ str(len(unhexlify(# exitx : 0 # )))
 .lsprint  ^ str(len(unhexlify( # lsprintx : (0,0) # )))
 .vdef   ^ str(len(unhexlify( # vdefx : (0,0,0) # )))
 .vsprint ^ str(len(unhexlify( # vsprintx : 0 # )))
```

Making the literal pool:

```
 makelit : global literals, llist, lend; llist = {}; lend = 0
         : literals = # .. #
         | ok
.seq     ^ #.1# + #.2#
.exit    ^ ""
.lsprint   ^ #.1#
.lstring  : global llist, lend; llist[.2] = lend; lend = lend + len(.1)
         ^ .1
.vdef    ^ #.2#
```

Counting the local variables:

```
countvars : global vlist, vend; vlist = {}; vend = 0
          : literals = # .. #
          | ok
 .seq     ^ #.1# + #.2#
 .vdef    : global vlist, vend; vlist[.1] = vend; vend = vend + 1
          | ok
```

Emitting the code:

```
emitcode  : global lstart, code, vend
          : code = unhexlify(# mkloclx : vend #) + # .. #
          | ok
 .seq     : global lstart, llist
          ^  #.1# + #.2#
 .exit    ^ unhexlify(# exitx : int(.1) # )
 .lsprint ^ unhexlify( # lsprintx : ( int(.1.3),lstart+llist[.1.2]) # )
 .vdef    ^ unhexlify( # vdefx : (lstart+llist[.2.2],vlist[.1],int(.2.3)) # )
 .vsprint ^ unhexlify( # vsprintx : vlist[.1] # )
```


'Assembly language' for each language construct, starting with making space for local variables:

```
 mkloclx  ^ # decsp   : .. * 8  # +
          ^ # esptoebp: .. #
```

To make an 'exit' system call, put 1 in eax and push the exit value on the

stack then call interrupt 0x80.

```
exitx    ^ # oneeax  : .. # +
         ^ # pushc   : ..  # +
         ^ # decsp   : 4  # +
         ^ # int     : 128  #
```

To make an 'write' system call with a string constant, put 4 in eax, push the length of the string, the location of the string and some padding on the stack, then call interrupt 0x80. When the system call returns, remove the values from the stack.

```
lsprintx ^ # pushc   : .0  # +
         ^ # pushc   : .1  # +
         ^ # pushc   : 1  # +
         ^ # toeax   : 4  # +
         ^ # decsp   : 4  # +
         ^ # int     : 128  # +
         ^ # incsp   : 16  #
```

To initialize a local variable to a string constant value, place the address of the string and the length of the string at the local variable's offset in the stack frame.


```
vdefx    ^ # toeax   : .0 # +
         ^ # ebpeax  :  .1 * 8 # +
         ^ # toeax   :  .2   # +
         ^ # ebpeax  : ( .1 * 8 ) + 4 #
```

To make an 'write' system call with a string variable, retrieve and push the length of the string, retrieve and push the location of the string, put 4 in eax and some padding on the stack, then call interrupt 0x80. When the system call returns, remove the values from the stack.

```
vsprintx ^ # eaxebp  : ( .. *8)+4 # +
         ^ # pusheax : .. # +
         ^ # eaxebp  : ( .. *8) # +
         ^ # pusheax : .. # +
         ^ # pushc   : 1  # +
         ^ # toeax   : 4  # +
         ^ # decsp   : 4  # +
         ^ # int     : 128  # +
         ^ # incsp   : 16  #
```

Machine code for the 'Assembly language':

```
oneeax   ^ "31c040"
esptoebp ^ "89e5"
incsp    ^ "81c4" + be2le("%08X" % (..))
decsp    ^ "81ec" + be2le("%08X" % (..))
toeax    ^ "b8"   + be2le("%08X" % (..))
eaxebp   ^ "8b45" + "%02X" % (..)
ebpeax   ^ "8945" + "%02X" % (..)
pushc    ^ "68"   + be2le("%08X" % (..))
pusheax  ^ "50"
int      ^ "cd"   + "%02X" % (..)
```

Building the mach-o program header:

```
end   : global header, code, literals
      : hdr =       unhexlify("cefaedfe070000000300000002000000020000000CC000000")
      : c1  =       unhexlify("00000000010000007c0000005f5f54455854000000000000")
      : c1  = c1  + unhexlify("000000000010000000100000000000000")
      : fsz =       unhexlify("00000000")
      : c2  =       unhexlify("07000000050000000100000000000000005f5f7465787400000000")
      : c2  = c2  + unhexlify("00000000000005f5f544558540000000000000000000E8100000")
      : bsz =       unhexlify("00000000")
      : c3  =       unhexlify("E8000000020000000000000000000000000000000")
      : c3  = c3  + unhexlify("00000000000000000500000050000000")
      : c3  = c3  + unhexlify("01000000100000000000000000000000")
      : c3  = c3  + unhexlify("00000000000000000000000000000000")
      : c3  = c3  + unhexlify("00000000000000000000000000000000")
      : c3  = c3  + unhexlify("E810000000000000000000000000000")
      : c3  = c3  + unhexlify("0000000000000000")
      : fsz = unhexlify( be2le( "%08X" % (232 + len(code) + len(literals)) ))
      : bsz = unhexlify( be2le( "%08X" % (len( code ) + len( literals )) ))
      : header = hdr + c1 + fsz + c2 + bsz + c3
      | ok
```

A toy compiler needs source code to compile:

```
print "This is a test of the emergency broadcast sytsem. This is only a test.\n"
var foo = "bar"
var baz = "quux"
print baz; print foo
print "This is another print statement.\n"
exit 42
```


Build the toy compiler semantics with sixparser then build the toy compiler with ibnfmeta:

```
python sixparser.py toy.six-py blank.txt toy-smtx.py
cat ./common-chr.ibnf ./toy.ibnf > ./toy-full.ibnf
python ibnfmeta.py toy-full.ibnf toy-smtx.py toy-compile.py
```

Compile the toy program source code with the toy compiler, then execute the toy executable:

```
python toy-compile.py test.toy blank.txt toy.out
chmod 755 toy.out
./toy.out
```