Introduzione ai Compilatori in Python

Enrico Franchi

Outline

- Introduzione
- Brevi cenni sui formalismi alla base della costruzione di un compilatore
- Descrizione di un semplice analizzatore lessicale con ply.lex
- Descrizione di un semplice analizzatore sintattico con ply.yacc
- Byteplay e generazione di codice per la macchina virtuale Python

Scrivere compilatori?

- Probabilmente un compilatore per un linguaggio general purpose no, ma...
  - File di configurazione
    - vi
  - Formati di file
    - YAML
  - Linguaggi per descrivere/automatizzare processi
    - Make
  - Acceptance tests

Scrivere compilatori?

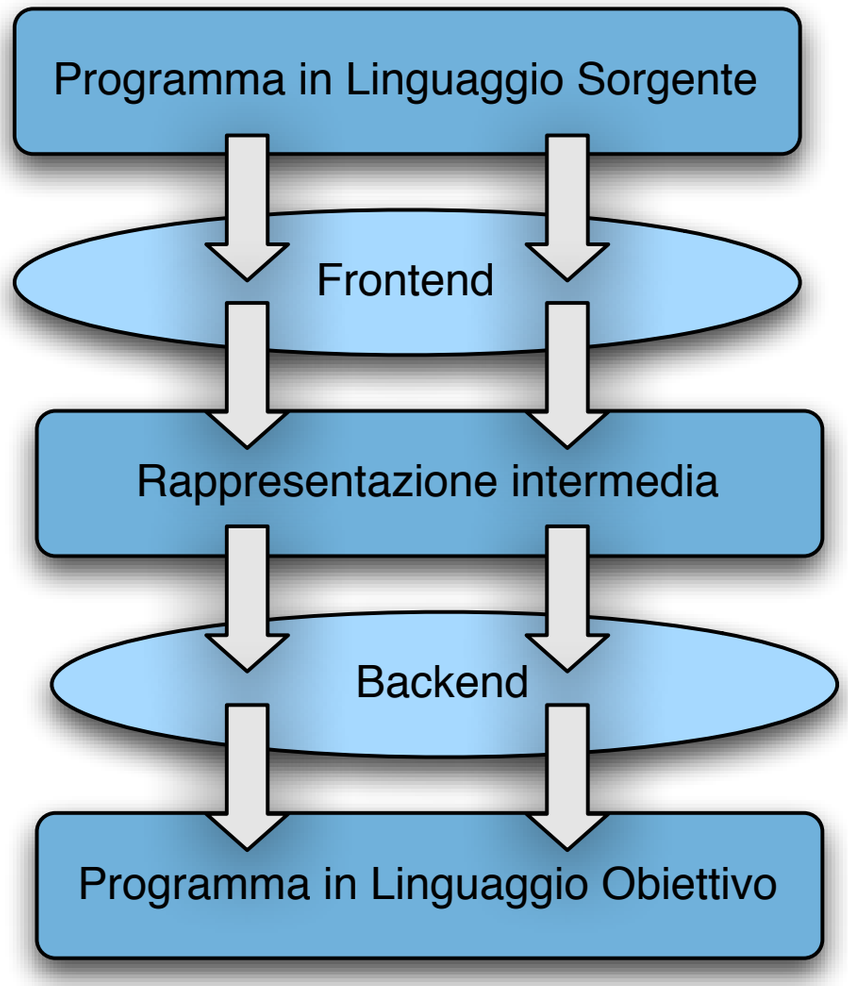
Organizing a program as a language processor encourages regularity of syntax (which is the user interface), and structures the implementation. It also helps to ensure that new features will mesh smoothly with existing ones. “Languages” are certainly not limited to conventional programming languages.

The Unix Programming Environment,

B. Kernighan - R. Pike

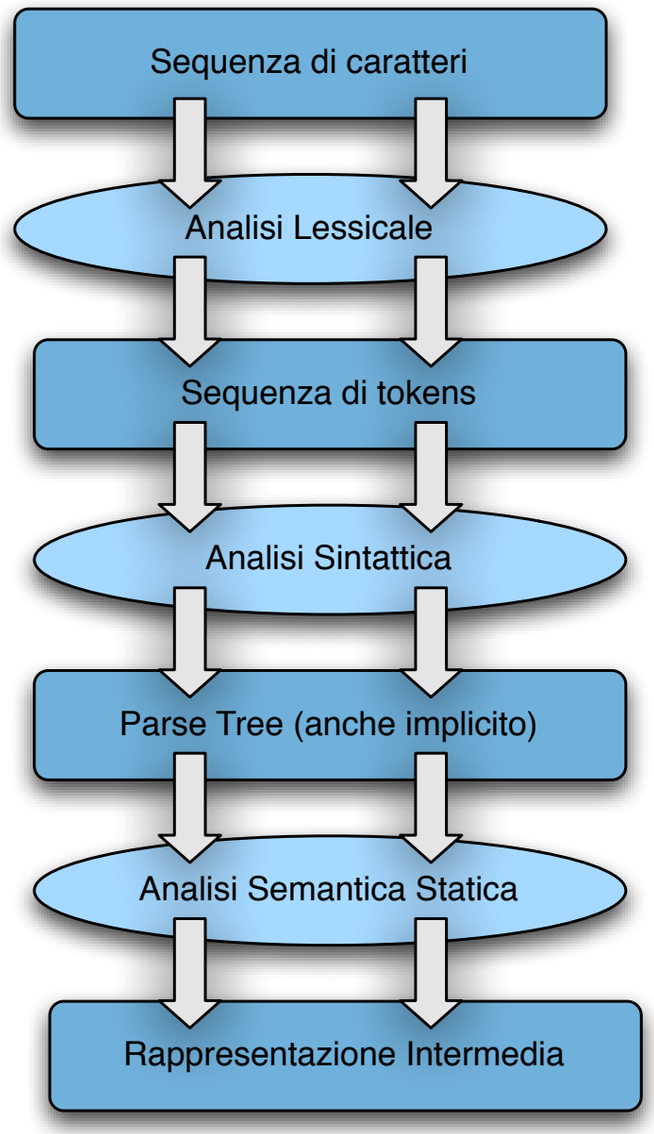
Struttura

- Tradizionalmente dividiamo il compilatore in front-end e backend
- Idealmente il front-end dipende solo dal linguaggio sorgente
- Il back-end prende il codice intermedio, eventualmente lo ottimizza e genera il codice finale



Front-end

- Trasformiamo la sequenza di caratteri in una sequenza di elementi lessicali (tokens)
- Una grammatica context free descrive in modo naturale la struttura della maggior parte dei costrutti di un linguaggio di programmazione
- I vincoli non context-free sono controllati nella fase di analisi di semantica statica
  - Tipi, identificatori, argomenti



Esempio Completo

- Supponiamo di volere scrivere un programma che riceve dall'utente una stringa del tipo

```
post.title contains 'Foo' and len post.body >= 120
```

e filtra una lista di posts in base al criterio

- Analizziamo la stringa e generiamo il codice per calcolare se un dato post verifica la condizione
- Creiamo una funzione con quel codice
- Iteriamo sulla lista e ritorniamo i valori per i quali la condizione è verificata

Esempio Semplice

- Per chiarirci le idee, partiamo da un sottoinsieme del problema originario: valutare delle espressioni aritmetiche
  - Valutare un'espressione aritmetica significa "capire" espressioni come "4 + (3 * 2)"
- In seguito aggiungeremo operazioni relazionali, and, or, operatori su stringhe etc.
- Cominciamo scrivendo l'analizzatore lessicale

Roadmap

- Introduzione
- Brevi cenni sui formalismi alla base della costruzione di un compilatore
- Descrizione di un semplice analizzatore lessicale con ply.lex
- Descrizione di un semplice analizzatore sintattico con ply.yacc
- Byteplay e code generation
- Completiamo l’esempio

Analisi Lessicale

- Un token è una coppia costituita da un nome e da un valore e viene riconosciuto tramite un pattern
- Un pattern viene tipicamente specificato tramite un'espressione regolare
- Un lexema è la sequenza di caratteri nel programma originale che è matchato da un pattern e diventa un'istanza del token corrispondente
- Convenzionalmente indichiamo i tokens con lettere maiuscole o fra singoli apici (se di un solo carattere)

Tokens Semplici

- Utilizziamo ply.lex
- tokens è una lista di stringhe con i nomi dei tokens
  - analogo a %token di yacc
- t_TOKEN contiene l'espressione regolare che deve matchare il token "TOKEN"

```
tokens = [
    'ADD', 'MUL', 'MINUS',
    'DIV', 'MOD',
    'LPAREN', 'RPAREN',
    'INTEGER'
]

t_ADD    = r'\+'
t_MUL    = r'\*'
t_MINUS  = r'-'
t_DIV    = r'/'
t_MOD    = r'%'
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

Tokens con Valore

- Per i token visti fin ora importa solo il nome
- Per i numeri è importante il valore, i token hanno per valore il lexema
- Convertiamo la stringa nel numero che rappresenta
- Ritorniamo il token modificato

```
def t_INTEGER(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        raise SyntaxError
    return t
```

Analisi Lessicale (miscellanea)

- Vogliamo ignorare spazi e tabulazione
- Ignoriamo i commenti
  - Una regola che ritorna None “ignora” il token
- Dobbiamo gestire gli eventuali errori
  - Tipicamente ignorarli non è una buona idea

```
t_ignore = ’ \t’

def t_COMMENT(t):
    r'\#.*’
    pass

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

```
if __name__ == '__main__':
    import ply.lex as lex
    lexer = lex.lex()
    try:
        while True:
            buffer = raw_input('> ')
            lexer.input(buffer)
            while True:
                tok = lexer.token()
                if not tok: break
                print "%-12s %s" % (tok.type, tok.value)
    except EOFError: print
```

```
% python tok.py
> 4 + (3 * 2)
INTEGER      4
ADD          +
LPAREN       (
INTEGER      3
MUL          *
INTEGER      2
RPAREN       )
```

Roadmap

- Introduzione
- Brevi cenni sui formalismi alla base della costruzione di un compilatore
- Descrizione di un semplice analizzatore lessicale con ply.lex
- Descrizione di un semplice analizzatore sintattico con ply.yacc
- Byteplay e code generation
- Completiamo l’esempio

Grammatiche Context-Free

- Un insieme di simboli terminali (tokens)
- Un insieme di simboli detti non terminali
  - Definiscono la struttura gerarchica del linguaggio
  - Ognuno determina un insieme di stringhe
- Un insieme di produzioni costituite da:
  - Un non terminale detto testa della produzione
  - Alcuni terminali e non terminali che costituiscono il corpo
  - Il corpo descrive un metodo in cui le stringhe della testa possono essere costruite
- Un non-terminale detto simbolo iniziale

Espressioni Aritmetiche

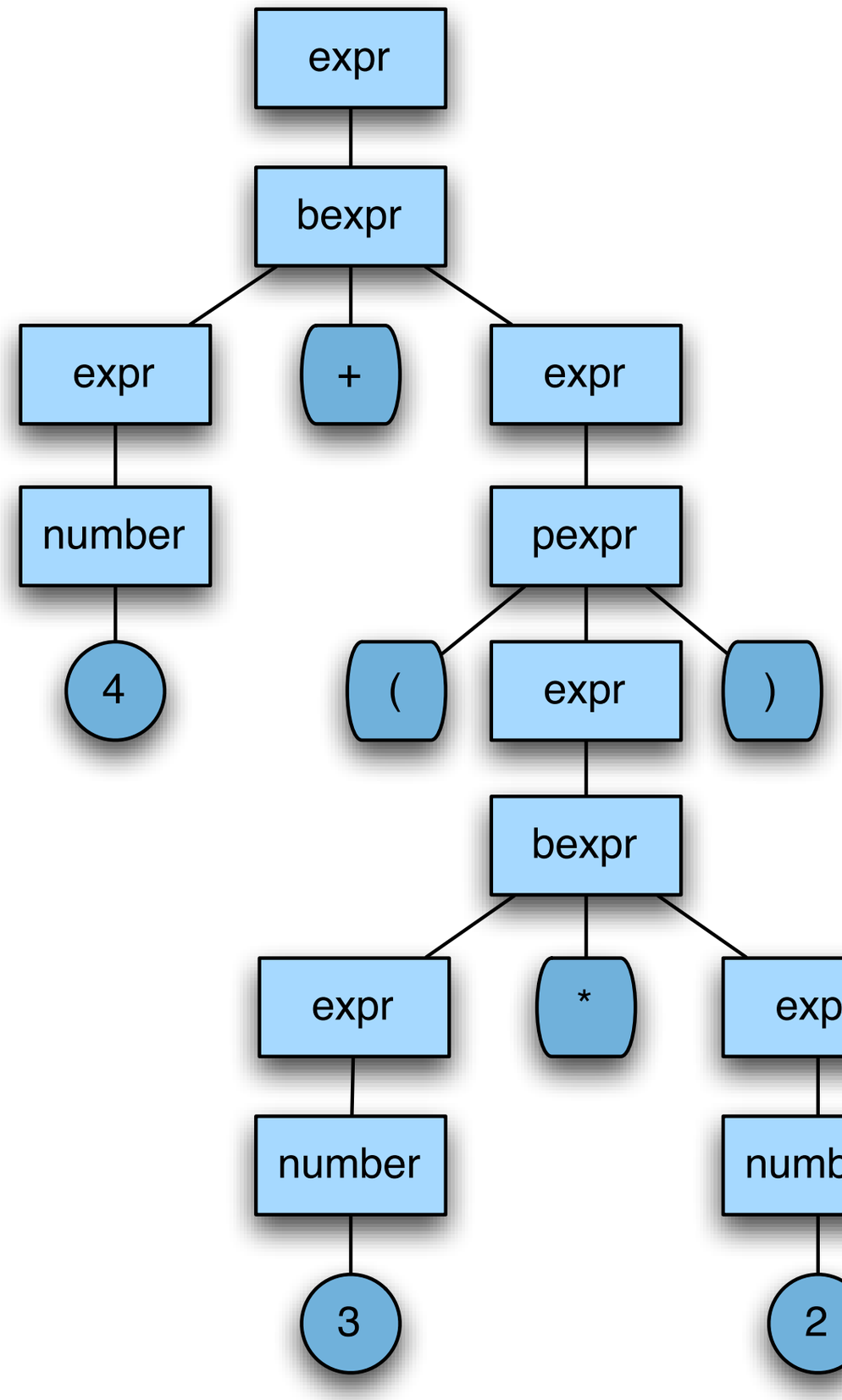

```
expr   : bexpr
       | uexpr
       | pexpr
       | number

bexpr  : expr ‘+’ expr
       | expr ‘*’ expr
       | expr ‘-’ expr
       | expr ‘/’ expr

uexpr  : ‘-’ expr

pexpr  : ‘(‘ expr ‘)’

number : INTEGER
```

- La grammatica descrive la struttura di un’espressione aritmetica
- Le espressioni aritmetiche valide sono solo quelle derivabili dalla grammatica

Esempio

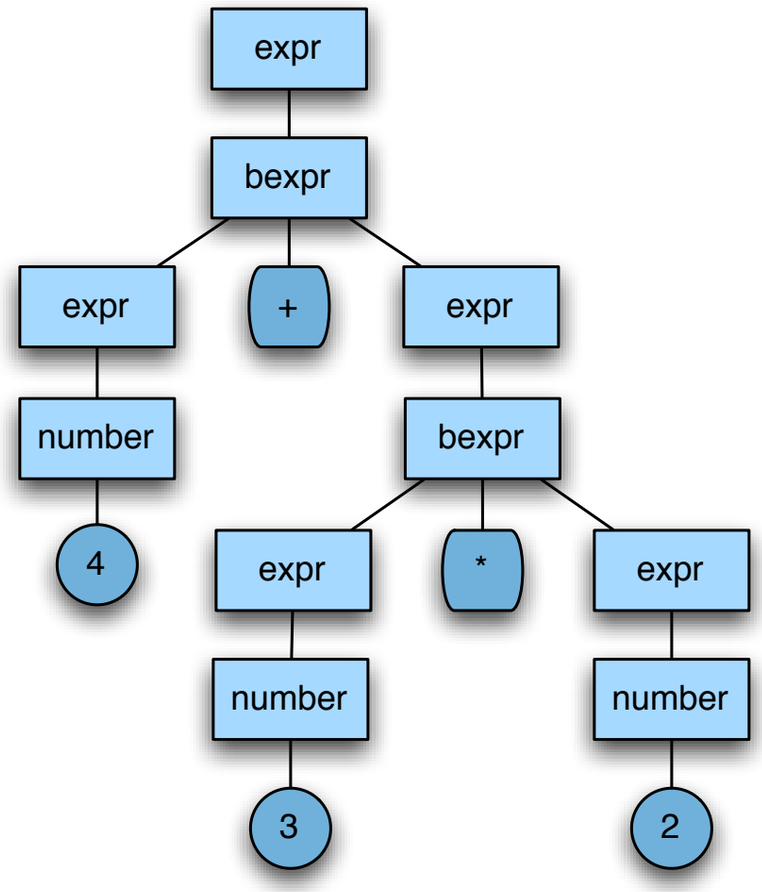
Albero di Parse per l'espressione

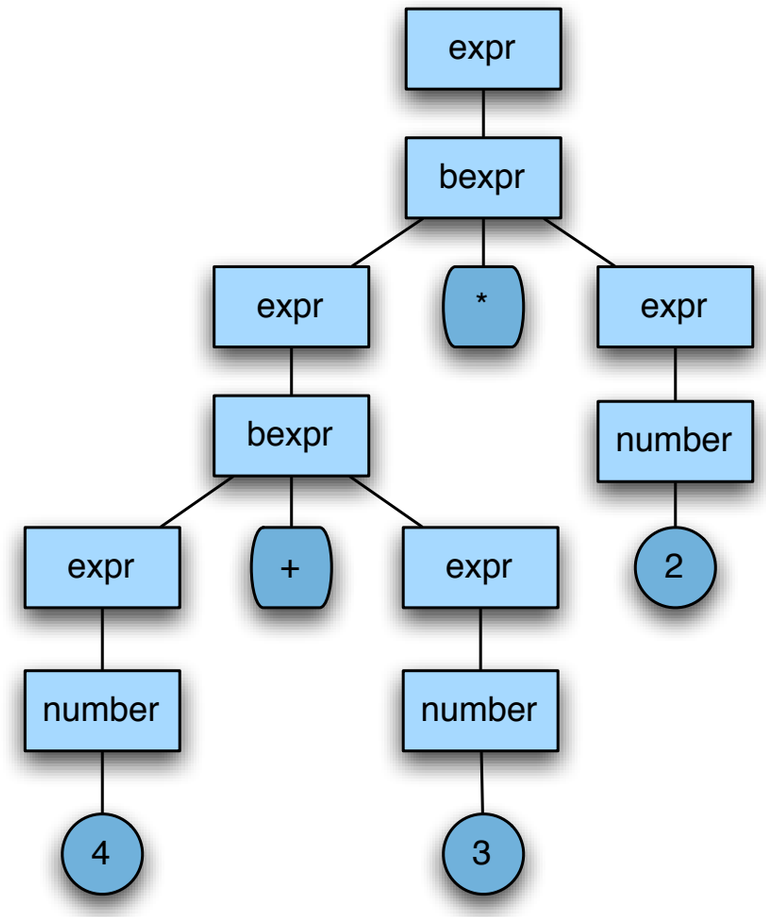
4 + (3 * 2)



Ambiguità

- Questa grammatica è ambigua!
- Consideriamo “4 + 3 * 2”





Syntax Directed Translation

- Agganciando azioni alle produzioni di una grammatica, la sintassi guida il processo
  - Controllo dei tipi / altri vincoli non CF
  - Generazione di codice / Interpretazione
- Associamo “attributi” ai costrutti del programma (ovvero ai simboli che li rappresentano)
- Le computazioni usano questi attributi per passare informazione ai vari livelli della “traduzione”

Analisi Sintattica (precedenza)

- Il parser ha bisogno della definizione della lista tokens
- Dobbiamo risolvere l'ambiguità della grammatica.
- Dichiariamo:
  - associatività
  - precedenza

```
from scanner import tokens

precedence = [
    ('left', 'ADD', 'MINUS'),
    ('left', 'MUL', 'DIV',
             'MOD'),
    ('right', 'UMINUS')
]
```

Analisi Sintattica (regole)

- Nella docstring mettiamo la produzione della grammatica
- Nel corpo della funzione definiamo azioni semantiche da compiere quando “usiamo” la regola
  - p[0] è la testa della produzione, p[*i*] l’*i*-esima componente nel corpo
  - p[0] è calcolato in funzione dei p[1], ..., p[n]

```
def p_expr(p):
    r'''expr : bexpr
             | pexpr
             | uexpr
             | number'''
    p[0] = p[1]

def p_number(p):
    r'number : INTEGER'
    p[0] = p[1]

def p_pexpr(p):
    r'''pexpr : LPAREN expr RPAREN'''
    p[0] = p[2]
```

Operazioni Binarie

- L'azione semantica associata alla regola "calcola" il valore
  - Approccio interpretativo
  - Spesso le costanti sono comunque valutate
- La struttura sintattica e anche semantica delle operazioni è simile

```
import operator

def p_bexpr(p):
    r'''bexpr : expr ADD   expr
              | expr MUL   expr
              | expr MINUS expr
              | expr DIV   expr
              | expr MOD   expr
              '''
    p[0] = {
        '+'   : operator.add,
        '-'   : operator.sub,
        '*'   : operator.mul,
        '/'   : operator.div,
        '%'   : operator.mod,
    }[p[2]](p[1], p[3])
```

Operazioni Unarie

- Il token ‘-’ (MINUS) compare sia come operatore binario che come operatore unario
  - Ha diversa precedenza
  - Ha diversa associatività
  - Specifichiamo nella regola che quando ‘-’ matcha come meno unario di usare una differente precedenza

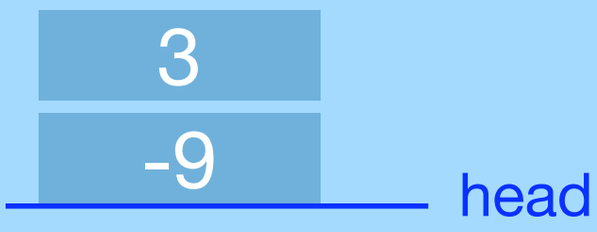
```
def p_uexpr(p):
    r'uexpr : MINUS expr %prec UMINUS'
    p[0] = -p[2]
```

Roadmap

- Introduzione
- Brevi cenni sui formalismi alla base della costruzione di un compilatore
- Descrizione di un semplice analizzatore lessicale con ply.lex
- Descrizione di un semplice analizzatore sintattico con ply.yacc
- Byteplay e code generation
- Completiamo l’esempio

Python Virtual Machine


```
LOAD_CONST 6
LOAD_CONST 2
UNARY_NEGATIVE
BINARY_MULTIPLY
LOAD_CONST 3
BINARY_ADD
```



- La macchina virtuale Python è una macchina a stack con supporto nativo per gli oggetti
- Le operazioni lavorano esclusivamente sui primi elementi dello stack
- Introdurremo gli opcodes mano a mano che ci servono

Byteplay

- Byteplay definisce i nomi dei vari opcodes
- Code è la classe principale del modulo
  - Una sua istanza può essere costruita a partire da un oggetto codice di Python
  - Può generare un oggetto codice Python a partire da una lista di opcodes e alcune informazioni aggiuntive
  - Rappresenta il programma come una lista di opcodes

Analizzare Oggetti Codice

- Utilizziamo byteplay per esaminare oggetti codice creati automaticamente da Python

```
>>> from byteplay import *
>>> def f(a, b): return a + b
...
>>> code = Code.from_code(f.func_code)
>>> print code.code

  1           1 LOAD_FAST                a
              2 LOAD_FAST                b
              3 BINARY_ADD
              4 RETURN_VALUE
```

Modificare Oggetti Codice

- Modifichiamo il codice on the fly

```
>>> code.code.insert(2, (LOAD_CONST, 2))
>>> code.code.insert(3, (BINARY_MULTIPLY, None))
>>> print code.code

  1           1 LOAD_FAST                a
              2 LOAD_CONST               2
              3 BINARY_MULTIPLY
              4 LOAD_FAST                b
              5 BINARY_ADD
              6 RETURN_VALUE
>>> f.func_code = code.to_code()
>>> f(3, 1)
7
```

Generare Oggetti Codice

- Costruttore dell'oggetto Code

```
Code(
   code,         # iterabile con gli opcodes
   freevars,     # lista dei nomi negli outer scopes
   args,         # lista dei nomi di variabile
   varargs,      # true se c'è *args
   varkwargs,    # true se c'è **kargs
   newlocals,    # true se crea un nuovo scope
   name,         # nome dell'oggetto
   filename,     # nome del file di definizione
   firstlineno, # prima linea di definizione
   docstring     # indovinare...
)
```

```
code_list = [
    (LOAD_FAST, 'a'),
    (LOAD_FAST, 'b'),
    (BINARY_ADD, None),
    (LOAD_FAST, 'm'),
    (BINARY_MODULO, None),
    (RETURN_VALUE, None)
]

c = Code(
    code         = code_list,
    freevars     = [],
    args         = ['a', 'b', 'm'],
    varargs      = False,
    varkwargs    = False,
    newlocals    = True,
    name         = 'add_mod',
    filename     = '<stdin>',
    firstlineno = 1,
    docstring    = 'Addition modulo m'
)

add_mod = new.function(c.to_code(), {})
```

Code Generation (1)


```
def emit_constant(value):
    return [(LOAD_CONST, value), ]

def emit_binary_operation(cf, cr, op):
    opcodes = {
        '+'  : (BINARY_ADD, None),
        # ...
        '%'  : (BINARY_MODULO, None),
    }
    code = cf + cr
    code.append(opcodes[op])
    return code

def emit_unary_operation(code, op):
    code.append((UNARY_NEGATIVE, None))
    return code
```

Code Generation (2)

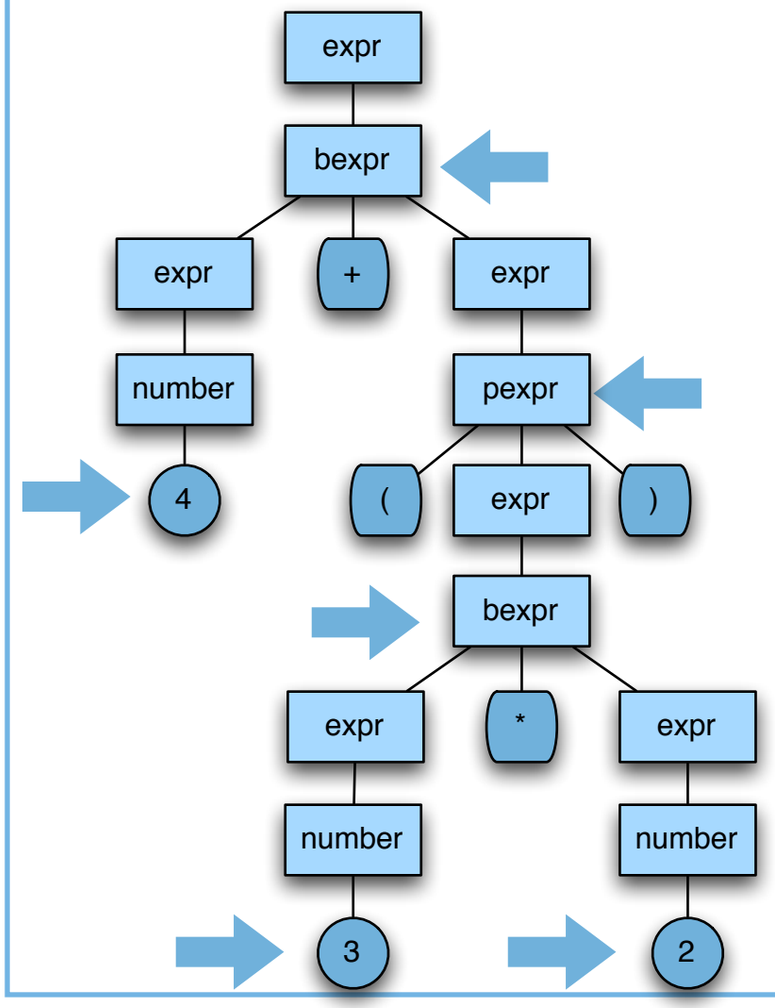

```
def build_cobj(code):
    code.extend([
        (DUP_TOP, None),
        (PRINT_ITEM, None),
        (PRINT_NEWLINE, None),
        (RETURN_VALUE, None)
    ])
    c = Code(
        code = code, freevars = [], args = [],
        varargs = False, varkwargs = False,
        newlocals = False, name = '',
        filename = '', firstlineno = 0,
        docstring = ''
    )
    return c.to_code()
```

Code Generation (3)


```
def p_expr(p):  # come prima
def p_pexpr(p): # come prima

def p_number(p):
    r'number : INTEGER'
    p[0] = emit_constant(p[1])

def p_bexpr(p):
    r'''bexpr : expr ADD   expr
              # cut...'''
    p[0] = emit_binary_operation(p[1], p[3], p[2])

def p_uexpr(p):
    r'uexpr : MINUS expr %prec UMINUS'
    p[0] = emit_unary_operation(p[2], p[1])
```

4 + (3 * 2)

```
LOAD_CONST 4
LOAD_CONST 3
LOAD_CONST 2
BINARY_MULTIPLY
BINARY_ADD
```




```
calc> 4 + (3 * 2)
p_number called: 4
p_expr called: 4
p_number called: 3
p_expr called: 3
p_number called: 2
p_expr called: 2
p_bexpr called: 3 * 2
p_expr called: 3 * 2
p_pexpr called: (3 * 2)
p_expr called: (3 * 2)
p_bexpr called: 4 + (3 * 2)
p_expr called: 4 + (3 * 2)
```

Eseguiamo il codice


```
def p_calc(p):
  r'calc : expr'
  p[0] = build_cobj(p[1])

# … rest of the code …

def interpreter():
  try:
    lex.lex(debug=1)
    yacc.yacc()
    while True:
      s = raw_input('calc> ')
      cobj = yacc.parse(s)
      print 'answ>',
      eval(cobj, {})
  except EOFError: print
```

- Aggiungiamo in testa una produzione per “incapsulare” la costante
- Creiamo una funzione per ottenere un semplice interprete di linea
  - Parserizziamo una linea per volta
  - Eseguiamo l’oggetto codice ritornato

Keywords e Identificatori

- In genere le parole chiave sono riconosciute dalla regola per gli identificatori
- Tuttavia una parola chiave non è un identificatore
- Definiamo un dizionario avente per chiavi le keywords e per valori i nomi dei tokens
- Scriviamo una regola che gestisce ambo i casi:

```
def t_ID(t):
    r'[A-Za-z_][0-9A-Za-z_]*'
    if t.value in keywords:
        t.type = keywords[t.value]
    return t
```

Roadmap

- Introduzione
- Brevi cenni sui formalismi alla base della costruzione di un compilatore
- Descrizione di un semplice analizzatore lessicale con ply.lex
- Descrizione di un semplice analizzatore sintattico con ply.yacc
- Byteplay e code generation
- Completiamo l’esempio

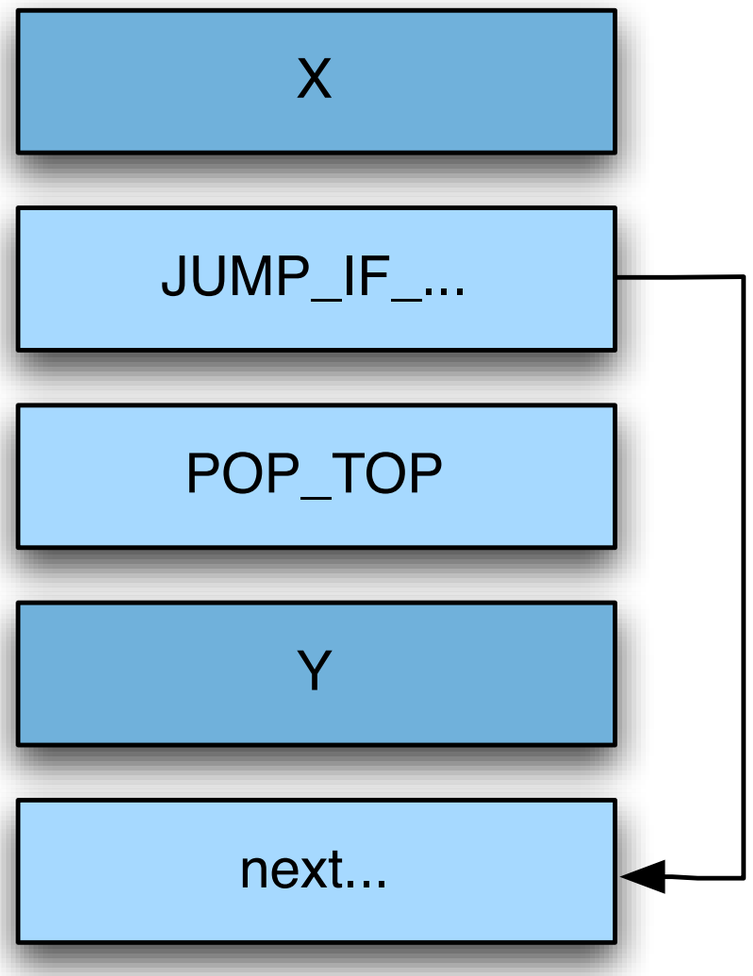
Operatori relazionali


```
precedence = [
    ('left', 'OR'),
    ('left', 'AND'),
    ('nonassoc', 'LE', 'GE', 'LT', 'GT'),
    ('nonassoc', 'EQ', 'NE'), # …
]

def p_rexpr(p):
    r''' rexpr : expr EQ expr
               | # …'''
    p[0] = emit_comparison(p[1], p[3], p[2])

def emit_comparison(cf, cr, op):
    code = cf + cr
    code.append((COMPARE_OP, op))
    return code
```

Operatori Corto-circuitati

- **or** e **and** sono “corto-circuitati”
- x **or** y : y è valutato se e solo se x è falso
  - in questo caso l’espressione vale y, altrimenti vale x

```
def emit_or(cls, cf, cr):
    next = Label()
    code = cf[:]
    code.append((JUMP_IF_TRUE, next))
    code.append((POP_TOP, None))
    code.extend(cr)
    code.append((next, None))
    return code
```



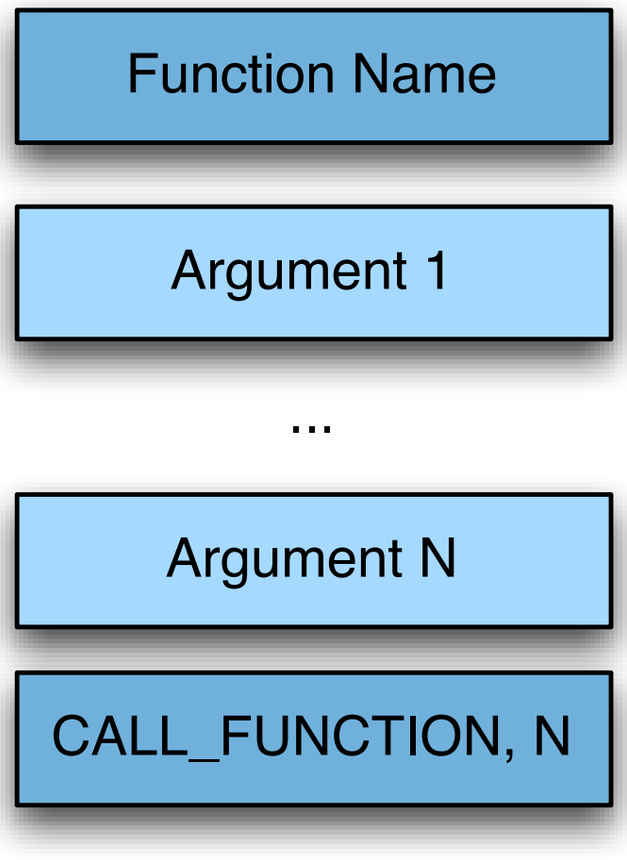
Gestione di Stringhe

- Aggiungiamo supporto alle stringhe
  - Le stringhe sono gestite direttamente dalla macchina virtuale
  - Insegnamo al tokenizzatore a riconoscere costanti letterali stringa
  - Aggiungiamo alcuni operatori per le stringhe
    - len s: restituisce la lunghezza di s
    - s contains p: vero se “p in s”
    - s startswith t : vero se s.startswith(t)
    - concatenazione di stringhe (con +)

```
def emit_len(expr_code):
    code = [(LOAD_GLOBAL, 'len'), ]
    code.extend(expr_code)
    code.append((CALL_FUNCTION, 1))
    return code
```

```
def emit_string_test(cf, cr, test):
    code = cf[:] # test ~ startswith
    code.append((LOAD_ATTR, test))
    code.extend(cr)
    code.append((CALL_FUNCTION, 1))
    return code
```

```
def emit_string_uop(code, op):
    code = code[:] # op ~ upper/lower
    code.append((LOAD_ATTR, op))
    code.append((CALL_FUNCTION, 0))
    return code

def emit_contains(cf, cr):
    code = cr + cf
    code.append((COMPARE_OP, 'in'))
    return code
```



Identificatori

- Aggiungiamo DOT ai token e assegnamogli precedenza massima
- Aggiungiamo le regole

```
def p_identifier(p):
    r'''identifier : simple_identifier
                   | attributed_identifier'''
    p[0] = p[1]

def p_simple_identifier(p):
    r'simple_identifier : ID'
    p[0] = emitter.simple_identifier(p[1])

def p_attributed_identifier(p):
    r'attributed_identifier : identifier DOT ID'
    p[0] = emitter.load_attribute(p[1], p[3])
```

Identificatori

- Trasformiamo l'emitter in una classe
- Teniamo traccia degli identificatori usati

```
def simple_identifier(self, name):
    if not name in self.symbols:
        self.symbols.append(name)
    return [(LOAD_FAST, name), ]

@staticmethod
def load_attribute(obj, attribute):
    code = obj[:]
    code.append((LOAD_ATTR, attribute))
    return code
```

Bibliografia

- *Compilers: Principles, Techniques, & Tools* - A. Aho, M. Lam, R. Sethi, J. Ullman - Addison-Wesley
- *Computer Organization and Design: The Hardware/ Software Interface* - D. Patterson, J. Hennessy - Morgan Kaufman Ed.
- *Lex & Yacc, Second Edition* - J. Levine, T. Mason, D. Brown - O. Reilly
- http://www.dabeaz.com/ply/ply.html
- http://wiki.python.org/moin/ByteplayDoc
- http://docs.python.org/lib/bytecodes.html

Conclusioni

- Abbiamo visto come scrivere il front-end per un semplice linguaggio
- Abbiamo mostrato come creare un interprete o un compilatore per quel linguaggio
- Usando come target di compilazione la macchina virtuale Python, possiamo eseguire al volo il codice generato
- Grazie!