How to Create a Programming Language

L10: INTERPRETATION

1

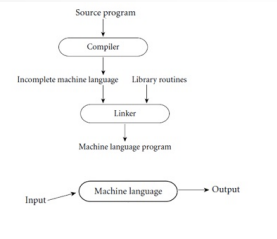
TOPICS

- DIFFERENCES BETWEEN INTERPRETERS AND COMPILERS
- TYPICAL COMPONENTS IN AN INTERPRETER

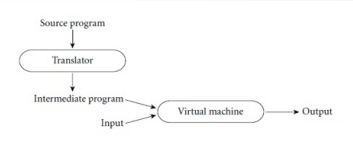
2

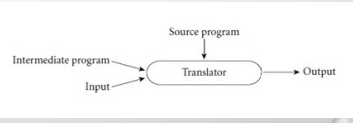
DIFFERENCES BETWEEN INTERPRETERS AND COMPILERS

Compiling



Interpreting





3

DIFFERENCES BETWEEN INTERPRETERS AND COMPILERS

Compiled		Interpreted	
Pros	**Cons**	**Pros**	**Cons**
Ready to Run	Not Cross Platform	Cross-platform	Interpreter required
Often faster	Inflexible	Simpler to test	Often slower
Source code private	Extra step	Easier to debug	Source code public

4

TYPICAL COMPONENTS

- LEXER
- PARSER / AST GENERATION – ABSTRACT SYNTAX TREE
- BYTECODE COMPILER
- BYTECODE INTERPRETER
- RUNTIME

5

OPTIONAL ALTERNATIVE

- LEXER
- PARSER / AST GENERATION – ABSTRACT SYNTAX TREE
- AST (WALKER) – EXECUTES ACTIONS IN CODE
- ~~BYTECODE COMPILER~~
- ~~BYTECODE INTERPRETER~~
- RUNTIME

6

LEXER

- SAME FUNCTION AS FOR COMPILER
- CONVERT STREAM OF CHARACTERS INTO STREAM OF TOKENS

7

PARSER / AST

- READS TOKEN STREAM
- GENERATES AN ABSTRCT SYNTAX TREE TO DESCRIBE ACTIONS

8

BYTECODE COMPILER

- AST IS 'WALKED' IN ORDER TO GENERATE INTERMEDIATE CODE (BYTECODE)
- GENERATING BYTECODE MAY INCLUDE CREATING STORAGE FOR
  - CONSTANTS
  - VARIABLES

9

BYTECODE INTERPRETER

- STACK BASED CODE
- OPERATIONS INCLUDE:
  - PUSH (LOAD) CONSTANT
  - PUSH (LOAD) VARIABLE
  - PERFORM OPERATION (+, -, *, /, ETC)
  - POP TOP OF STACK (RESULT)
  - STORE VARIABLE VALUEE
- THIS NOW EXECUTES THE BYTECODES MAKING UPDATES TO VARIABLES IN STORAGE

10

RUNTIME

- THIS IS SIMPLY VARIABLE STORAGE AND THE EXECUTION STACK

11

OPTIONAL ALTERNATIVE

- LEXER
- PARSER / AST GENERATION – ABSTRACT SYNTAX TREE
- AST (WALKER) – EXECUTES ACTIONS IN CODE
- RUNTIME

12

EXAMPLE – SIMPLE EXPRESSION GRAMMAR

```
assign => NAME EQUALS expr
expr => expr PLUS term
expr => expr MINUS term
expr => term
term => term TIMES factor
term => term DIVIDE factor
term => factor
factor => NUMBER
factor => NAME
factor => ( expr )
```


13

ARCHITECTURE

- CODE WILL USE PLY FOR LEXER AND PARSER
- BOTTOM UP (LR(1)) PARSER
- PARSER GENERATES AST
- AST COMPILED INTO BYTECODES
- BYTECODE IS INTERPRETED TO RUN THE PROGRAM

14

LEXER CODE

```
import ply.lex as lex

# List of token names.   This is always required
tokens = (
   'PLUS',
   'MINUS',
   'TIMES',
   'DIVIDE',
   'LPAREN',
   'RPAREN',
    'NAME',
    'EQUALS',
    'NUMBER'
)

# Regular expression rules for simple tokens
t_PLUS    = r'\+'
t_MINUS   = r'-'
t_TIMES   = r'\*'
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_EQUALS  = r'='

# A regular expression rule with some action code
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    print("Number!")
    return t

def t_NAME(t):
    r'[a-zA-Z0-9_][a-zA-Z0-9_]*'
    print("Name!")
    return t
```


15

LEXER CODE

```
# Define a rule so we can track line numbers
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# A string containing ignored characters (spaces
and tabs)
t_ignore  = ' \t'

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

# Build the lexer
lexer = lex.lex()
```


16

PARSER CODE

```
import ply.yacc as yacc
import exprLexer
import exprLang as ast
import context
Import sys

tokens = exprLexer.tokens

start = 'assign'

vars = {};

def p_assign(p):
   '''assign : NAME EQUALS expr'''
   vars[p[1]] = p[3]
   print ('REDUCE : assign : NAME EQUALS expr')
   print (vars[p[1]])
   p[0] = ast.Assign(p[1], p[3])
```

```
def p_expr_plus(p):
   '''expr : expr PLUS term'''
   print ('REDUCE : expr : expr PLUS term')
   p[0] = ast.Expr('+', p[1], p[3])

def p_expr_minus(p):
   '''expr : expr MINUS term'''
   print ('REDUCE : expr : expr MINUS term')
   p[0] = ast.Expr('-', p[1], p[3])

def p_expr_term(p):
   '''expr : term'''
   print ('REDUCE : expr : term')
   p[0] = p[1]
```


17

PARSER CODE

```
def p_term_times(p):
   '''term : term TIMES factor'''
   print ('REDUCE : term : term TIMES factor')
   p[0] = ast.Expr('*', p[1], p[3])

def p_term_divide(p):
   '''term : term DIVIDE factor'''
   print ('REDUCE : term : term DIVIDE factor')
   p[0] = ast.Expr('/', p[1], p[3])

def p_term_factor(p):
   '''term : factor'''
   print ('REDUCE : term : factor')
   p[0] = p[1]

def p_factor(p):
   '''factor : NUMBER'''
   print ('REDUCE : factor : NUMBER')
   p[0] = ast.Number(p[1])

def p_factor_name(p):
   '''factor : NAME'''
   print ('REDUCE : factor : NAME')
   p[0] = ast.Name(p[1])
```

```
def p_factor_expr(p):
   '''factor : LPAREN expr RPAREN'''
   print ('REDUCE : factor : LPAREN expr RPAREN')
   p[0] = p[2]
def p_error(p):
   print ("Error parsing")

def print_program(theProgram):
   for line in theProgram:
      print(str(line))

yacc.yacc()
data = "a=10+14*20-b"
result = yacc.parse(data)
print(result.toStringNode(0))

ctx = context.Context()

result.compile(ctx)

theProgram = ctx.programCode()
print_program(theProgram)
print(str(ctx.constants))
print(str(ctx.variables))
```


18

PARSER CODE

```
yacc.yacc()

sourceFile = ''
if len(sys.argv) < 2:
   print("No source!")
else:
   sourceFile = sys.argv[1]
print ('Source: ' + sourceFile)

progfile = open(sourceFile, 'r')
data = progfile.readline()
progfile.close()
result = yacc.parse(data)
print(result.toStringNode(0))
ctx = context.Context()

result.compile(ctx)

theProgram = ctx.programCode()
print("Program code:")
print_program(theProgram)
print(str(ctx.constants))
print(str(ctx.variables))
```


19

EXPRLANG CODE
CLASSES TO BUILD AND WALK AST

```
spcString = "                "

class Name():
   def __init__(self, value):
      self.type = "Name"
      self.nameText = value

   def toStringNode(self, lvl):
      return "Name: " + self.nameText

   def compile(self, ctx):
      ctx.emit2("LOAD_VAR ", ctx.new_var(self.nameText))

class Number():
   def __init__(self, value):
      self.type = "Number"
      self.numberValue = value

   def toStringNode(self, lvl):
       return "Val: " + str(self.numberValue)

   def compile(self, ctx):
      ctx.emit2("LOAD_CONST ", ctx.new_const(self.numberValue))
```


20

EXPRLANG CODE
CLASSES TO BUILD AND WALK AST

```
class Assign():
    def __init__(self, var, expr):
        self.type = "Assign"
        self.var = var
        self.expr = expr

    def toStringNode(self, lvl):
        return "Assign: " + "\n" + spcString[:lvl+1] + self.var + "\n" + spcString[:lvl+1] +  self.exp\
r.toStringNode(lvl+1)

    def compile(self, ctx):
        self.expr.compile(ctx)
        ctx.new_var(self.var)
        ctx.emit1("POP_TOP ")
        ctx.emit2("STORE ", self.var)
        pass
```


21

EXPRLANG CODE
CLASSES TO BUILD AND WALK AST

```
class Expr():
    def __init__(self, op, arg1, arg2):
        self.type = "Expr"
        self.op = op
        self.arg1 = arg1
        self.arg2 = arg2

    def toStringNode(self, lvl):
        return "Expr: " + self.op + "\n" + spcString[:lvl] + str(self.arg1.toStringNode(lvl+1)) + "\n"\
 + spcString[:lvl] + str(self.arg2.toStringNode(lvl+1))

    def compile(self,ctx):
        self.arg1.compile(ctx)
        self.arg2.compile(ctx)
        if (self.op == '+'):
            ctx.emit1("BIN_OP_ADD ")
        elif (self.op == '-'):
            ctx.emit1("BIN_OP_SUB ")
        elif (self.op == '*'):
            ctx.emit1("BIN_OP_MUL ")
        else:
            ctx.emit1("BIN_OP_DIV ")
```


22

CONTEXT
FOR GENERATING BYTECODE FROM AST

```
class Context:
    def __init__(self):
        self.constants = []
        self.variables = dict([])
        self.program = []

    def new_var(self, name):
        if name not in self.variables:
            self.variables[name] = 0

    def new_const(self, number):
        self.constants.append(number)
        return len(self.constants)-1

    def emit1(self, op):
        self.program.append((op))

    def emit2(self, op, arg):
        self.program.append((op, arg))

    def programCode(self):
        return self.program
```


23

SAMPLE RUN

test1.el:
a=10+14*20-b

python3 exprParser.py test1.el

```
Source: test1.el
Name!
Number!
REDUCE : factor : NUMBER
REDUCE : term : factor
REDUCE : expr : term
Number!
REDUCE : factor : NUMBER
REDUCE : term : factor
Number!
REDUCE : factor : NUMBER
REDUCE : term : term TIMES factor
REDUCE : expr : expr PLUS term
Name!
REDUCE : factor : NAME
REDUCE : term : factor
REDUCE : expr : expr MINUS term
REDUCE : assign : NAME EQUALS expr
<exprLang.Expr object at 0x1033c4190>
```

```
Assign:
a
Expr: -
Expr: +
 Val: 10
 Expr: *
  Val: 14
  Val: 20
 Name: b
Program code:
('LOAD_CONST ', 0)
('LOAD_CONST ', 1)
('LOAD_CONST ', 2)
BIN_OP_MUL
BIN_OP_ADD
('LOAD_VAR ', None)
BIN_OP_SUB
POP_TOP
('STORE ', 'a')
[10, 14, 20]
{'b': 0, 'a': 0}
```


24


QUESTIONS?
Sejong - How to Create a Programming Language - Tony Mione / 2020
25

25