Implementing a Simple Interpreter

Ray Song

May 27, 2012

Flow sheet

- Lexical analysis
- Grammatical analysis

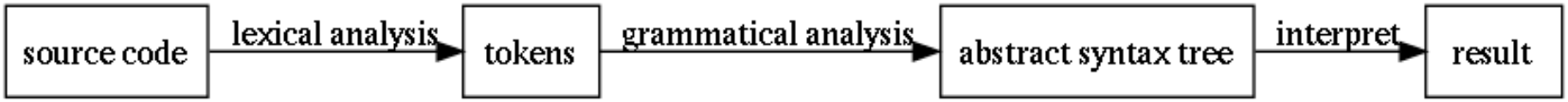

Figure : Flow

Flow sheet

- Lexical analysis
- Grammatical analysis

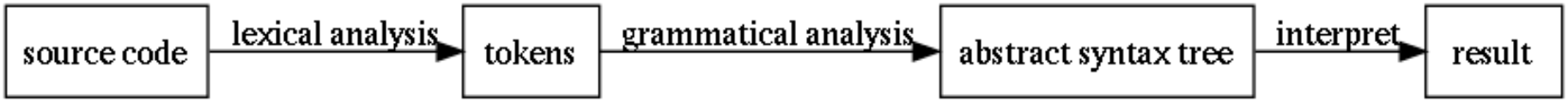

Figure : Flow

Lexical Analysis

- manual
- flex

Lexical Analysis

- manual
- flex

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Tokens

- identifier
- integer
- string
- while
- if
- else
- print
- NEWLINE
- NO WHITESPACE

Off-side rule

def hello():

print 'world'

- indentation sensitive
- Ex: ISWIM(1966), occam(1983), Miranda(1985), Haskell(1990), Python(1991)

Off-side rule

def hello():

print 'world'

- indentation sensitive
- Ex: ISWIM(1966), occam(1983), Miranda(1985), Haskell(1990), Python(1991)

Off-side rule - Cont.

- represented as virtual tokens
- INDENT
- DEDENT

Off-side rule - Cont.

- represented as virtual tokens
- INDENT
- DEDENT

Off-side rule - Cont.

- represented as virtual tokens
- INDENT
- DEDENT

Example

if a > 2:

    print 5

print a

- IF a > 2 : NEWLINE INDENT PRINT 5 NEWLINE DEDENT PRINT a NEWLINE

Grammatical analysis

- Cocke–Younger–Kasami algorithm
- Earley's algorithm
- LR parser
- Recursive-descent parser

Grammatical analysis

- Cocke–Younger–Kasami algorithm
- Earley's algorithm
- LR parser
- Recursive-descent parser

Grammatical analysis

- Cocke–Younger–Kasami algorithm
- Earley's algorithm
- LR parser
- Recursive-descent parser

Grammatical analysis

- Cocke–Younger–Kasami algorithm
- Earley's algorithm
- LR parser
- Recursive-descent parser

Tools

- Bison
- Can generate C, C++ and Java codes
- ANTLR

Tools

- Bison
- Can generate C, C++ and Java codes
- ANTLR

Tools

- Bison
- Can generate C, C++ and Java codes
- ANTLR

Expression

- Precedence climbing method
- Shunting-yard algorithm
- Parsing expressions in infix notation
- Output in Reverse Polish notation (RPN)
- Output in Abstract syntax tree (AST)
- Operator precedence parser

Expression

- Precedence climbing method
- Shunting-yard algorithm
- Parsing expressions in infix notation
- Output in Reverse Polish notation (RPN)
- Output in Abstract syntax tree (AST)
- Operator precedence parser

Expression

- Precedence climbing method
- Shunting-yard algorithm
- Parsing expressions in infix notation
- Output in Reverse Polish notation (RPN)
- Output in Abstract syntax tree (AST)
- Operator precedence parser

Expression

- Precedence climbing method
- Shunting-yard algorithm
- Parsing expressions in infix notation
- Output in Reverse Polish notation (RPN)
- Output in Abstract syntax tree (AST)
- Operator precedence parser

Expression

- Precedence climbing method
- Shunting-yard algorithm
- Parsing expressions in infix notation
- Output in Reverse Polish notation (RPN)
- Output in Abstract syntax tree (AST)
- Operator precedence parser

Expression

- Precedence climbing method
- Shunting-yard algorithm
- Parsing expressions in infix notation
- Output in Reverse Polish notation (RPN)
- Output in Abstract syntax tree (AST)
- Operator precedence parser

Parser combinator

- Straightforward to construct
- Readability
- Parsec

Parser combinator

- Straightforward to construct
- Readability
- Parsec

Parser combinator

- Straightforward to construct
- Readability
- Parsec

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl

- Two top-level nonterminals: STMT and EXPR
- STMT: SIMPLE_STMT '\n' | COMPOUND
- SIMPLE_STMT: EXPR
- SIMPLE_STMT: IDENT '=' EXPR
- SIMPLE_STMT: BREAK
- SIMPLE_STMT: print EXPR
- COMPOUND: if EXPR ':' SUITE OPT_ELSE
- COMPOUND: while EXPR ':' SUITE
- SUITE: many1('\n') INDENT many1(STMT) DEDENT
- SUITE: SIMPLE_STMT '\n'
- OPT_ELSE: ELSE ':' SUITE
- OPT_ELSE: /* empty */

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Impl - Cont.

- EXPR: EXPR '==' TERM
- EXPR: EXPR '!=' TERM
- EXPR: TERM
- TERM: TERM '+' FACTOR
- TERM: FACTOR
- FACTOR: FACTOR '*' ATOM
- FACTOR: ATOM
- ATOM: identifier
- ATOM: literal_integer
- ATOM: literal_string
- ATOM: '(' EXPR ')'

Example

```
if a > 2:
    print 5
print a
```

Example - Cont.

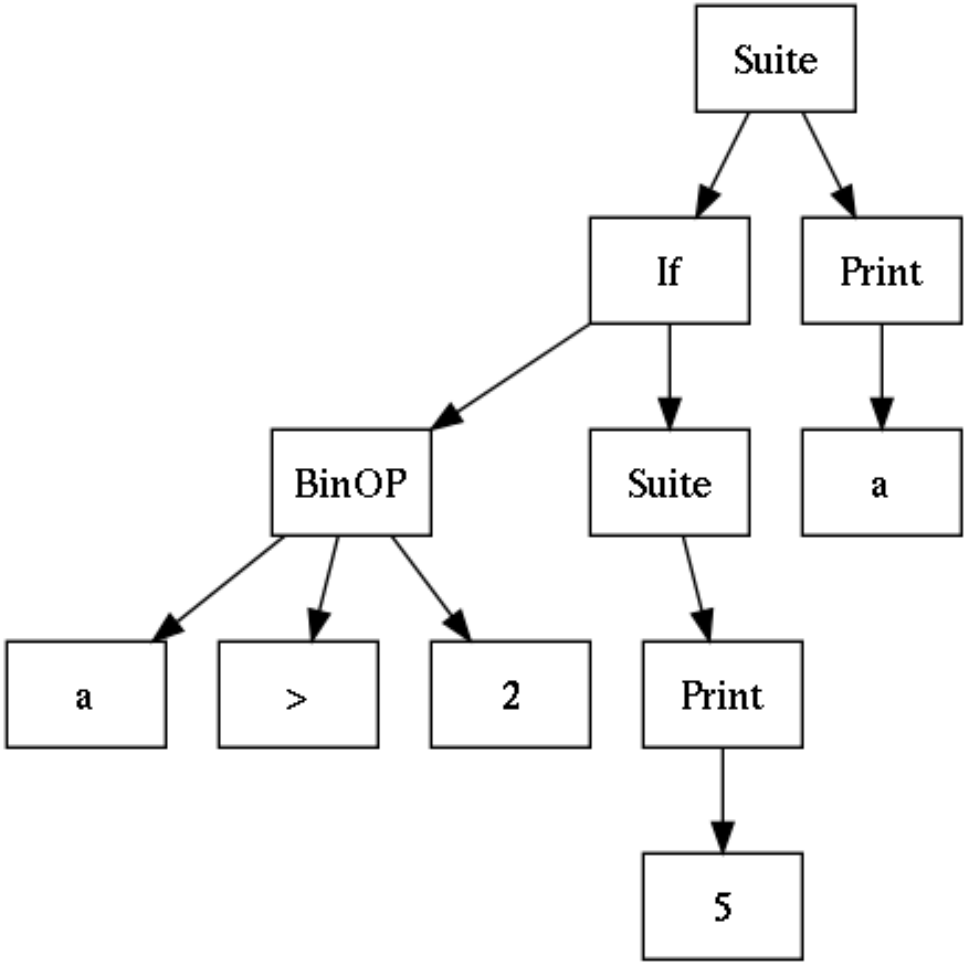

Figure : Parse

Interpreting

- Abstract syntax tree (AST).
- Define semantics for each class of nodes
- object(atom): trivial
- binary operator BinOP(operator, lhs, rhs, RESULT) :- obj1 = eval(lhs), obj2 = eval(rhs), calc(op, obj1, obj2, RESULT).
- Object & BinOP inherit from Expr

Interpreting

- Abstract syntax tree (AST).
- Define semantics for each class of nodes
- object(atom): trivial
- binary operator BinOP(operator, lhs, rhs, RESULT) :- obj1 = eval(lhs), obj2 = eval(rhs), calc(op, obj1, obj2, RESULT).
- Object & BinOP inherit from Expr

Interpreting

- Abstract syntax tree (AST).
- Define semantics for each class of nodes
- object(atom): trivial
- binary operator BinOP(operator, lhs, rhs, RESULT) :- obj1 = eval(lhs), obj2 = eval(rhs), calc(op, obj1, obj2, RESULT).
- Object & BinOP inherit from Expr

Interpreting

- Abstract syntax tree (AST).
- Define semantics for each class of nodes
- object(atom): trivial
- binary operator BinOP(operator, lhs, rhs, RESULT) :- obj1 = eval(lhs), obj2 = eval(rhs), calc(op, obj1, obj2, RESULT).
- Object & BinOP inherit from Expr

Interpreting

- Abstract syntax tree (AST).
- Define semantics for each class of nodes
- object(atom): trivial
- binary operator BinOP(operator, lhs, rhs, RESULT) :- obj1 = eval(lhs), obj2 = eval(rhs), calc(op, obj1, obj2, RESULT).
- Object & BinOP inherit from Expr

Subclasses of Stmt - Cont.

- Assign
- eval() need a parameter: Binding (which variable holds which object)
- ExprStmt
- Print
- Continue (throwing an exception)

Subclasses of Stmt - Cont.

- Assign
- eval() need a parameter: Binding (which variable holds which object)
- ExprStmt
- Print
- Continue (throwing an exception)

Subclasses of Stmt - Cont.

- Assign
- eval() need a parameter: Binding (which variable holds which object)
- ExprStmt
- Print
- Continue (throwing an exception)

Subclasses of Stmt - Cont.

- Assign
- eval() need a parameter: Binding (which variable holds which object)
- ExprStmt
- Print
- Continue (throwing an exception)

Subclasses of Stmt - Cont.

- Assign
- eval() need a parameter: Binding (which variable holds which object)
- ExprStmt
- Print
- Continue (throwing an exception)