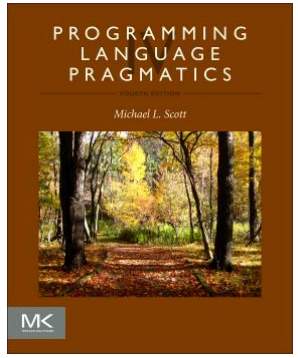
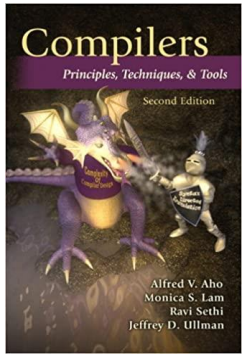
Semantic Analysis





LARK Python Parser

2025.8

[Welcome to Lark's documentation! — Lark documentation]

Lexer for C regx-CFG.lark

```
% import common.WS
% import common.C_COMMENT
% import common.CPP_COMMENT
% ignore WS
% ignore C_COMMENT
% ignore CPP_COMMENT

KEYWORD.2: "int" | "float" | "char" | "doubl
e" | "void" | "return" | "if" | "else"

IDENTIFIER : / [a - zA - Z_][a - zA - Z0 - 9
_] * /
INT : / \d + /
FLOAT: / \d + \.\d + /

PLUS: "+"
MINUS : "-"
MULT : "*"
DIV : "/"

ASSIGN : "="
SEMICOLON : ";"

LPAREN : "("
RPAREN : ")"
LBRACE : "{"
RBRACE : "}"

LT : "<"
GT : ">"
LE : "<="
GE : ">="
EQ : "=="
NEQ : "!="
```

```
start : function

function : KEYWORD IDENTIFIER LPAREN RPAREN block

block : LBRACE stmt_list RBRACE

stmt_list : (stmt)*

stmt : declaration SEMICOLON
| init_declaration SEMICOLON
| assign_stmt SEMICOLON
| expr SEMICOLON
| if_stmt
| return_stmt SEMICOLON
| block

declaration : KEYWORD IDENTIFIER
init_declaration : KEYWORD IDENTIFIER ASSIGN expr

assign_stmt : IDENTIFIER ASSIGN expr

if_stmt : "if" LPAREN rel_expr RPAREN stmt
| "if" LPAREN rel_expr RPAREN stmt "else" stmt

return_stmt : "return" expr

? rel_expr : expr(LT | GT | LE | GE | EQ | NEQ) expr->rel

? expr : expr PLUS term->add
| expr MINUS term->sub
| term

? term : term MULT factor->mul
| term DIV factor->div
| factor

? factor : INT -> int
| FLOAT -> float
| IDENTIFIER->var
| LPAREN expr RPAREN->paren
```

Tokenizing

```
from lark import Lark

# Load the lexer grammar
with open("regx-CFG.lark", "r") as file :
grammar = file.read()

lexer = Lark(grammar, parser = "lalr", lexer = "standard", maybe_placeho
lders = False)

# Sample code to tokenize
code = """
int main() {
int x = 10;
float y = 20.5;
if (x < y) {
x = x + 1;
}
return x;
}
"""
```

```
# Tokenize input
tokens = lexer.parse(code).children

print("Tokens:")
for token in tokens :
print(token)
```

Output

```
Tokens: int main ( ) { int x
= 10 ; float y = 20.5 ; if (
x < y ) { x = x + 1 ; }
return x ; }
```

Parsing

```
from lark import Lark, Transformer

# Load the grammar
with open("regx-CFG.lark", "r") as file :
grammar = file.read()

parser = Lark(grammar, parser = "lalr", start = "start")

# Sample code to parse
code = """
int main() {
    int x;
    x = 10 + 20 * (3 + 2);
    return x;
}
"""

#code = """
#int main() {
    #    int x = 10;
    #    float y = 20.5;
#    if (x < y) {
    #        x = x + 1;
#    }
    #    return x;
#}
#"""

    tree = parser.parse(code)
        print(tree.pretty())
```

Output

```
start
  function
    int
    main
    (
    )
    block
      {
      stmt_list
        stmt
          declaration
            int
            x
          ;
        stmt
          assign_stmt
            x
            =
            add
              int        10
              +
              mul
                int      20
                *
                paren
                  (
                  add
                    int 3
                    +
                    int 2
                  )
          ;
        stmt
          return_stmt
            var x
          ;
      }
```

AST

```
from lark import Lark, Transformer

# Load the grammar
with open("regx-CFG.lark", "r") as file :
grammar = file.read()

parser = Lark(grammar, parser = "lalr", start = "start")


class ASTBuilder(Transformer) :
    def start(self, items) :
    return items[0]

    def function(self, items) :
    # items : return_type, name, block
    return ("function", str(items[0]), str(items[1]), items[-1])

    def block(self, items) :
    return ("block", *items)

    def stmt_list(self, items) :
    return items

    def declaration(self, items) :
    return ("decl", str(items[0]), str(items[1]))

    def init_declaration(self, items) :
    return ("init_decl", str(items[0]), str(items[1]), items[2])

    def assign_stmt(self, items) :
    return ("assign", str(items[0]), items[1])

    def return_stmt(self, items) :
    return ("return", items[0])
```

…

….

```
    def if_stmt(self, items) :
    if len(items) == 2 :
        return ("if", items[0], items[1], None)
    else :
        return ("if", items[0], items[1], items[2])

        # Expressions
        def add(self, items) :
        return ("+", items[0], items[1])


tree = parser.parse(code)

ast = ASTBuilder().transform(tree)

print_ast_unicode(ast)
```

Output

```
function
├── int
├── main
└── block
    ├── decl
    │   ├── int
    │   └── x
    ├── assign
    │   ├── x
    │   └── +
    │       ├── int
    │       │   └── 10
    │       └── int
    │           └── 20
    └── return
        └── var
            └── x
```