Entwicklung eines Compilers

Von einer Grammatik zur Code-Ausführung

Von René Hosch, Arne Köller, Vanessa Schieben,
Fabian Scholz und Benedikt Sielaff

Compilerbau
Wintersemester 2017/2018

Gliederung

- Einleitung und Grammatik
- Grobstruktur Compiler
- Scanner/Lexer
- Recursive-descent Parser
- Codeerzeugung
- Virtuelle Maschine
- Ausblick und Fazit
- Demo

Einleitung

- Sprachenname: “magic” (File-Ext.: .magic)
- Compiler komplett in Python geschrieben
- Syntax-Highlighting für SublimeText-Editor
- Un-typisierte Variableninitialisierung
- Unterstützung aller einfachen Datentypen
- Kommentarfunktion
- Funktionsaufrufe mit Rekursion möglich
- Sehr performant (z.B. fac(10.000) in 1 sek.)
- Auf Github zu finden unter: https://github.com/koeller21/comp

Grammatik - Grundlagen

- Erstellung einer Grammatik für eine kontextfreie Sprache
- Vermeidung von Linksrekursionen
- Spezifikation und Erstellung von Grammatikstrukturen zu:
  - Programmstatements
  - Variablendeklaration
  - Arithmetischen Ausdrücken
  - Logischen Ausdrücken
  - Funktionsdefinitionen
  - Funktionsaufrufen
  - Konditionalen Ausdrücken
  - Rückkehraufrufen

Grammatik - Grundlagen

Geschützte Wörter:

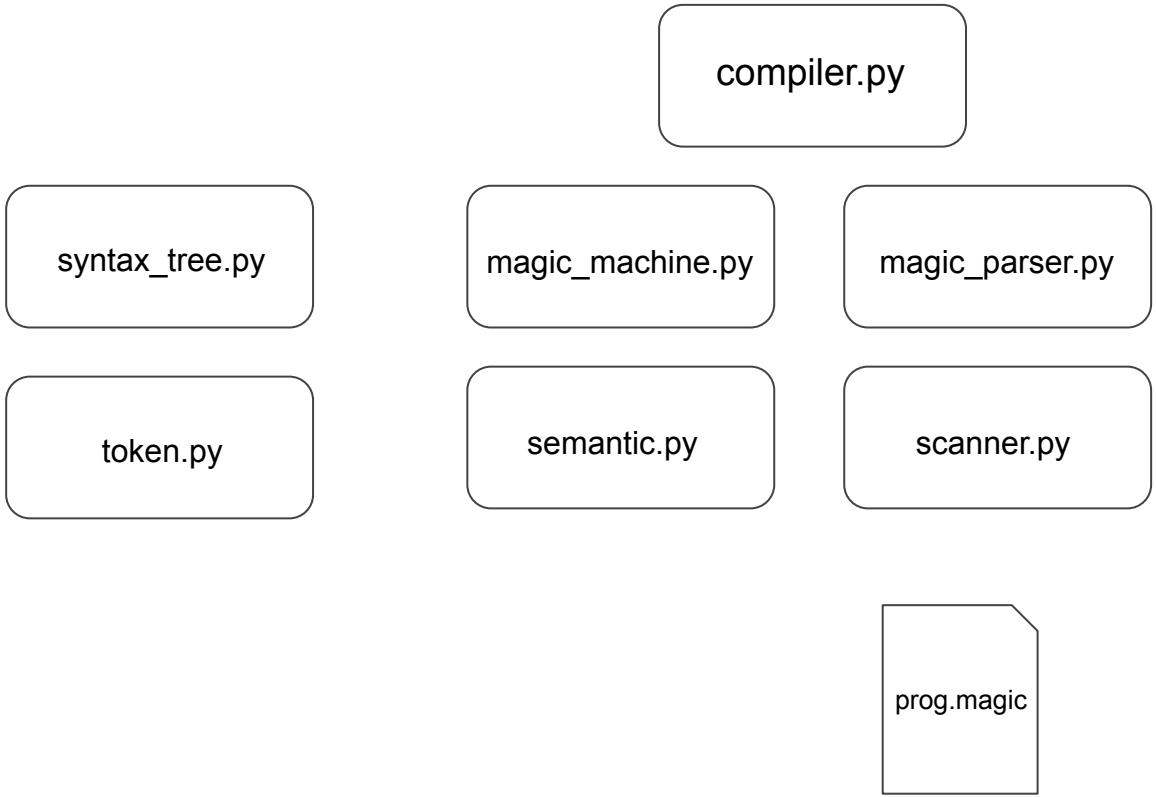
- magic für einen Funktionsdeklaration
- ; für ein Zeilenende
- return für Funktionsrückgabe
- if- und else-Konditionen
- + , - , * , / , % , = , { , } , (,)
- Logische Operatoren <, >, ==, !=, &&, ||

Grammatik - Beispiel

```
MagicCode -> Statement
Statement -> AssignmentStatement; Statement
          -> ApplicationStatement; Statement
          -> FunctionDefinition; Statement
          -> conditional; Statement
          -> ReturnStatement
          -> ε

AssignmentStatement -> identifier '=' expression
                    -> identifier '=' identifier
                    -> identifier '=' var_string
                    -> identifier '=' bool_expression
                    -> identifier '=' ApplicationStatement
```

```
FunctionDefinition -> 'magic' identifier '(' parameter ')' '{' statement '}'
parameter          -> identifier R
                   -> ε
R                  -> ',' parameter R
                   -> ε
```

Grobstruktur des Compilers



Scanner - Grundlagen

- Liest den Quellcode aus einer Datei ein
- Entfernt Kommentare und Escapezeichen wie “\n”, “\t”
- Tokenized den Quellcode in Operatoren, Identifier und Keywords
- Ein Token besteht aus (token_type, data)
- Reguläre Ausdrücke statt expliziter DEA
- Speichert die Token in ein Array

Scanner - Beispiel

```
try:
    with open(prog,"r") as content:
        magic_programm = ""
        for line in content:

            line = line.replace(" ","")
            line = line.replace("\n","")
            line = line.replace("\t","")

            if not line.startswith("#"):
                magic_programm = magic_programm + line

        tokens = self.tokenize(magic_programm)
        return tokens
except IOError as e:
    print(e)
    return False
```

```
def getWordToToken(word):
    token_t = ""
    if word == ";":
        token_t = SEMICOLON
    elif word == ",":
        token_t = COMMA
    elif word == "(":
        token_t = OPEN_PARA
    elif word == ")":
        token_t = CLOSED_PARA
    elif word == "=":
        token_t = EQUALS
    elif word == "+":
        token_t = PLUS
    elif word == "-":
        token_t = MINUS
    elif word == "*":
        token_t = MUL
    elif word == "/":
        token_t = DIV
    elif word == "%":
        token_t = MOD
```

Parser

- Baut den Syntaxtree auf
- Prüft auf Syntaktische Korrektheit
- Bricht den Compilerlauf bei fehlerhafter Syntax ab
- Gibt bei entsprechendem Fehler entspreche Rückgabe zum Beispiel: “Geschlossene Klammer vergessen”

Parser

```
# expression -> term rightExpression
def expression(self, syntax_tree):
    return self.term(syntax_tree.insertSubtree(token.TERM)) and self.rightExpression(syntax_tree.insertSubtree(token.RIGHTEXPRESSION))
```

Syntaxbaum für die Grammatik “expression → term rightExpression”

```
if self.cond_else(syntax_tree.insertSubtree(token.COND_ELSE)):
    return True
else:
    print("Else-Zweig falsch!")
    return False
```

Syntaktische Korrektheit wird geprüft

Syntaxbaum

```
STATEMENT ---> None
        ASSIGNMENT ---> None
                IDENTIFIER ---> a
                EQUALS ---> =
                EXPRESSION ---> None
                        TERM ---> None
                                OPERATOR ---> None
                                        VAR_NUM ---> 4
                                RIGHTTERM ---> None
                                        MUL ---> *
                                        OPERATOR ---> None
                                                VAR_NUM ---> 3
                                        RIGHTTERM ---> None
                                                EPSILON ---> None
                        RIGHTEXPRESSION ---> None
                                EPSILON ---> None
        SEMICOLON ---> ;
        STATEMENT ---> None
                EPSILON ---> None
```

Syntaxbaum für das Beispiel: a = 4 * 3;

Codeerzeugung

- VM-Maschinenbefehle werden definiert
- Nutzung des Syntaxtrees des Parsers
  - Baumdurchlauf
  - Insbesondere Non-Terminal-Token bekommen VM-Code zugewiesen
- Sukzessiver Aufbau des VM-Programmcodes
- Beachtung der Postfix-Notation für VM-Stack
- Liefert der Stackmaschine die VM-Maschinenbefehle

Codeerzeugung

- Startpunkt für jeden Durchlauf ist die Funktion generate()
- Durchläuft den Syntaxbaum
- Anschließend wird mit der Funktion semantic_statement() geprüft welche Regeln weiter verfolgt werden
- Durchlauf geht immer weiter bis zum Ende des Syntaxbaums

```
def generate(self):
    if self.st.get_token() == "MAGICCODE":
        return str(self.semantic_statement(self.st.children[0], ""))
    else:
        return None

def semantic_statement(self, top, n):

    if len(top.children) == 3:
        nxt_stmt = top.children[0]
        semicolon = top.children[1]
        stmt = top.children[2]


        if nxt_stmt.get_token() == "ASSIGNMENT":
            return n + self.semantic_statement(stmt, self.assignment_statement(nxt_stmt))
        if nxt_stmt.get_token() == "APPLICATION":
            return n + self.semantic_statement(stmt, self.semantic_application_statement(nxt_stmt))

    elif len(top.children) == 2:
        nxt_stmt = top.children[0]
        stmt = top.children[1]

        if nxt_stmt.get_token() == "CONDITIONAL":
            return n + self.semantic_statement(stmt, self.semantic_condition(nxt_stmt))
        if nxt_stmt.get_token() == "FUNCTIONDEF":
            return n + self.semantic_statement(stmt, self.semantic_function_definition(nxt_stmt))
        if nxt_stmt.get_token() == "RETURN":
            return n + self.semantic_return(nxt_stmt)
    elif len(top.children) == 1:
        nxt_stmt = top.children[0]
        if nxt_stmt.get_token() == "EPSILON":
            return n
```

Beispiel: a = 4 * 3;
;PUSH CONSTANT 4;PUSH CONSTANT 3;MUL;POP VARIABLE a

Codeerzeugung - VM-Code-Beispiel

```
magic simple_add(a, b)
{
  c = a + b;
  return c;
}


a = 10;
b = 20;


x = simple_add(a, b);
```

```
FUNCTION simple_add
POP VARIABLE a
POP VARIABLE b
PUSH VARIABLE a
PUSH VARIABLE b
ADD
POP VARIABLE c
PUSH VARIABLE c
RETURN
PUSH CONSTANT 10
POP VARIABLE a
PUSH CONSTANT 20
POP VARIABLE b
PUSH VARIABLE b
PUSH VARIABLE a
CALL simple_add
POP VARIABLE x
```

Virtuelle Maschine - Grundlagen

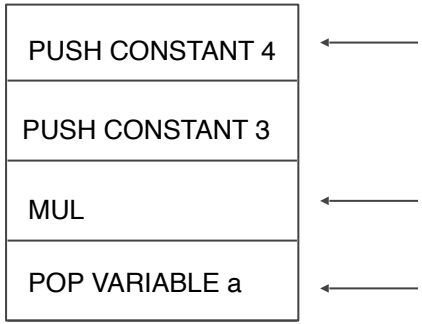
- Eng geknüpft an Codeerzeugung
- Arbeitet den VM-Code Befehl für Befehl ab
- VM besteht aus vielen Komponenten:
  - Code-Memory
  - Instruction Pointer
  - Intelligente Symbol-Tabelle für Variablen
  - Stack arithmetische und logische Operationen
  - Funktionenstack für Aufrufverschachtelung

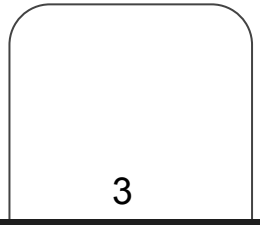
```
class magic_machine():


    def __init__(self, vm_code, debug_mode):

        self.debug_mode = debug_mode

        self.ip = 0 #instruction pointer
        self.code_memory = vm_code.split(";")[1:]

        if debug_mode:
            print("Code Memory : " + str(self.code_memory))

        self.label_table = []
        self.build_label_table() #build initial label table

        self.stack = []

        self.symbol_table = []
        self.create_symbol_table_for_function_call() # build initial symbol table

        self.function_table = []
        self.build_function_table() # build initial function table

        self.function_stack = []
```

Virtuelle Maschine - Ablauf (Sequenziell)

Magic-Code: **a = 4 * 3;**

VM-Code: **;PUSH CONSTANT 4;PUSH CONSTANT 3;MUL;POP VARIABLE a**






```
def run(self):

    while self.ip < len(self.code_memory):

        if self.debug_mode: # execute slowly in debug mode
            time.sleep(1)

        self.interpret(self.code_memory[self.ip])

        if self.debug_mode: # print states of vm
            print("Command : " + str(self.code_memory[self.ip]) + " || Code Memory Position : " + str(self.ip))
            print("Stack : " + str(self.get_stack()))
            print("Symbol Tabellen :" + str(self.get_symbol_table()))
            print("Function Stack : " + str(self.get_function_stack()))
            print("============")

        self.ip = self.ip + 1
```

Virtuelle Maschine - Ablauf (Konditional)

```
if( (var < 3) || (var == 3) ){
        return 1;
}else{
        return 0;
}
```

```
PUSH VARIABLE var
PUSH CONSTANT 3
ST
PUSH VARIABLE var
PUSH CONSTANT 3
EQ
OR
GOFALSE L0
PUSH CONSTANT 1
RETURN
LABEL L0
PUSH VARIABLE var
PUSH CONSTANT 3
ST
PUSH VARIABLE var
PUSH CONSTANT 3
EQ
OR
GOTRUE L1
PUSH CONSTANT 0
RETURN
LABEL L1
```

1. **Logische Ausdrücke über Stack ausgewertet**
2. **Konditional-Statements durch Label umgesetzt**
3. **Um Jumps zu ermöglichen muss eine Label-Table im Pre-Compiling aufgebaut werden (Label x -> Addr)**
4. **Jump = Setzen von IP auf Addr**

Virtuelle Maschine - Ablauf (Funktionen)

Schwierigkeiten:

1. Wie merken, welche Funktion welche Funktion aufgerufen hat (*Verschachtelungsproblem*)?

Lösung: Funktionen-Stack! Merke die Addr bei CALL, POP bei RETURN

2. Wie Variablenüberschreibung für Rekursionen vermeiden?
   -> Normaler Algorithmus ist nämlich: Falls Variable neu, trage sie mit Wert in Symbol-Tabelle ein
   Falls Variable vorhanden, überschreibe Wert in Sym.-Tabelle

Lösung:

- **Aufbau von “Frames” in der Symbol-Tabelle.**
- **Jeder Funktionsaufruf hat eigenes Frame.**

```
Stack : []
Symbol Tabellen :[[['n', 7]], [['n', 8], ['a', 7]], [['n', 9], ['a', 8]], [['n', '10'], ['a', 9]], [['i', '10']]]
Function Stack : [18, 18, 18, 30]
```

Ausblick und Fazit

- Features ergänzen
  - Komplexe Datentypen (Arrays, Strings, Klassen)
    - Benötigt Heap-Struktur mit z.B. malloc-Befehl
  - SYS-Calls für I/O
  - Import-Funktion für Magic-Bibliotheken (wie z.B. math.magic)
- Mehr Exceptions einfügen
- Floating-Point-Unterstützung

- Magic ist eine imperativ-prozeduale Programmiersprache mit einem in Python geschriebenen Compiler
- Magic erlaubt Funktionen-Rekursion durch Stack-Frames
- Magic ist leicht erweiterbar (OO, ca. 1300 Zeilen Compiler-Code)

Demo

https://github.com/koeller21/comp