Введение

1 Аналитический раздел

1.1 Постановка задачи

1.2 Метод такой-то

1.3 Метод сякой-то

Заключение

2 Конструкторский раздел

2.1Грамматика

Описание грамматики:

— единственный тип данных — int  
— все переменные — глобальные. Всего есть 26 переменных (a-z)  
— из математических операций поддерживаются только "+" и "-"  
— единственный оператор сравнения — это "<"  
— из конструкций языка — условные операторы if, if/else, while, do/while

Полученная грамматика:

<program> ::= <statement>

<statement> ::= "if" <paren-expr> <statement> |

"if" <paren-expr> <statement> "else" <statement> |

"while" <paren-expr> <statement> |

"do" <statement> "while" <paren-expr> |

"{" { <statement> } "}" |

<expr> ";" |

";"

<paren-expr> ::= "(" <expr> ")"

<expr> ::= <test> | <id> "=" <expr>

<test> ::= <sum> | <sum> "<" <sum>

<sum> ::= <term> | <sum> "+" <term> | <sum> "-" <term>

<term> ::= <id> | <int> | <paren-expr>

<id> ::= "a" | "b" | ... | "z"

<int> ::= <digit>, { <digit> }

<digit> ::= "0" | "1" | ... | "9"

Это запись грамматики в форме EBNF.   
  
Программа — это один оператор (statement).  
Операторы бывают условными (if..else...), циклическими (while...) и просто операторами (напр., «a=2+3»).  
Условные и циклические содержат в себе выражение-условие и тело (в виде оператора). Обычные операторы заканчиваются точкой с запятой. Можно группировать операторы в фигурных скобках, тогда получим составной оператор.  
Выражения — это либо сумма, либо присваивание переменной значения.  
Здесь сумма — это последовательность сложений-вычитаний термов.   
Терм — это число, переменная или выражение в скобках.  
Переменные — это символы от a до z. Числа — это набор цифр от 0 до 9.

2.2 Лексический анализатор

На вход компилятору поступает текстовый файл (исходник). [Лексический анализатор](http://en.wikipedia.org/wiki/Lexical_analysis) нужен для того, чтобы выделить в этом файле токены. Т.е. строчку «a = 3 + 42;» лексический анализатор должен представить в виде «идентификатор: a», «оператор =», «число 3», «оператор +», «число 42», «символ ;». Лексический анализатор работает только с последовательностью букв, т.е. для него строчка «a b if =» тоже имеет смысл и является абсолютно корректной.  
  
Итак, наш лексический анализатор должен узнавать следующие токены:  
  
— числа  
— идентификаторы-переменные  
— ключевые слова: if, else, while, do  
— символы +, -, <, =, {, }, (, ),;  
— конец файла  
  
Вот как выглядит его исходный код:

class Lexer:

NUM, ID, IF, ELSE, WHILE, DO, LBRA, RBRA, LPAR, RPAR, PLUS, MINUS, LESS, \

EQUAL, SEMICOLON, EOF = range(16)

# специальные символы языка

SYMBOLS = { '{': LBRA, '}': RBRA, '=': EQUAL, ';': SEMICOLON, '(': LPAR,

')': RPAR, '+': PLUS, '-': MINUS, '<': LESS }

# ключевые слова

WORDS = { 'if': IF, 'else': ELSE, 'do': DO, 'while': WHILE }

# текущий символ, считанный из исходника

ch = ' ' # допустим, первый символ - это пробел

def error(self, msg):

print 'Lexer error: ', msg

sys.exit(1)

def getc(self):

self.ch = sys.stdin.read(1)

def next\_tok(self):

self.value = None

self.sym = None

while self.sym == None:

if len(self.ch) == 0:

self.sym = Lexer.EOF

elif self.ch.isspace():

self.getc()

elif self.ch in Lexer.SYMBOLS:

self.sym = Lexer.SYMBOLS[self.ch]

self.getc()

elif self.ch.isdigit():

intval = 0

while self.ch.isdigit():

intval = intval \* 10 + int(self.ch)

self.getc()

self.value = intval

self.sym = Lexer.NUM

elif self.ch.isalpha():

ident = ''

while self.ch.isalpha():

ident = ident + self.ch.lower()

self.getc()

if ident in Lexer.WORDS:

self.sym = Lexer.WORDS[ident]

elif len(ident) == 1:

self.sym = Lexer.ID

self.value = ord(ident) - ord('a')

else:

self.error('Unknown identifier: ' + ident)

else:

self.error('Unexpected symbol: ' + self.ch)

В методе next\_tok() анализатор получает следующий токен. Тип токена можно получить из атрибута sym, а его значение (если это переменная или число) — из атрибута value.   
  
Анализатор игнорирует пробелы, проверяет, является ли текущий символ специальным символом языка, если нет — проверяет, является ли он числом или идентификатором. Т.е. встретив цифру 1, анализатор продолжит вычитывать символы, пока не встретит не-числовой символ. Аналогично проверяются идентификаторы

2.3 Парсер

Задача парсера, используя токены, полученные от лексического анализатора, сформировать своего рода дерево, в котором иерархия и связи отображают структуру кода. Например:

"if (a < 0) a = 5;"

IF

+-LESS

| +-VAR(a)

| +-NUM(0)

+-SET

+-VAR(a)

+-NUM(5)

Дерево, которое строится парсером, состоит из узлов. У узла есть тип (IF, LESS, SET, VAR...), значение (если это число или переменная) и несколько дочерних узлов-операндов (в коде — op1, op2, op3). Для if в них хранятся условие и ветки then/else, для циклов — условие и тело цикла.  
  
Для описания узлов введем класс Node. Вот код класса парсера и класса Node:

class Node:

def \_\_init\_\_(self, kind, value = None, op1 = None, op2 = None, op3 = None):

self.kind = kind

self.value = value

self.op1 = op1

self.op2 = op2

self.op3 = op3

class Parser:

VAR, CONST, ADD, SUB, LT, SET, IF1, IF2, WHILE, DO, EMPTY, SEQ, EXPR, PROG = range(14)

def \_\_init\_\_(self, lexer):

self.lexer = lexer

def error(self, msg):

print 'Parser error:', msg

sys.exit(1)

def term(self):

if self.lexer.sym == Lexer.ID:

n = Node(Parser.VAR, self.lexer.value)

self.lexer.next\_tok()

return n

elif self.lexer.sym == Lexer.NUM:

n = Node(Parser.CONST, self.lexer.value)

self.lexer.next\_tok()

return n

else:

return self.paren\_expr()

def summa(self):

n = self.term()

while self.lexer.sym == Lexer.PLUS or self.lexer.sym == Lexer.MINUS:

if self.lexer.sym == Lexer.PLUS:

kind = Parser.ADD

else:

kind = Parser.SUB

self.lexer.next\_tok()

n = Node(kind, op1 = n, op2 = self.term())

return n

def test(self):

n = self.summa()

if self.lexer.sym == Lexer.LESS:

self.lexer.next\_tok()

n = Node(Parser.LT, op1 = n, op2 = self.summa())

return n

def expr(self):

if self.lexer.sym != Lexer.ID:

return self.test()

n = self.test()

if n.kind == Parser.VAR and self.lexer.sym == Lexer.EQUAL:

self.lexer.next\_tok()

n = Node(Parser.SET, op1 = n, op2 = self.expr())

return n

def paren\_expr(self):

if self.lexer.sym != Lexer.LPAR:

self.error('"(" expected')

self.lexer.next\_tok()

n = self.expr()

if self.lexer.sym != Lexer.RPAR:

self.error('")" expected')

self.lexer.next\_tok()

return n

def statement(self):

if self.lexer.sym == Lexer.IF:

n = Node(Parser.IF1)

self.lexer.next\_tok()

n.op1 = self.paren\_expr()

n.op2 = self.statement()

if self.lexer.sym == Lexer.ELSE:

n.kind = Parser.IF2

self.lexer.next\_tok()

n.op3 = self.statement()

elif self.lexer.sym == Lexer.WHILE:

n = Node(Parser.WHILE)

self.lexer.next\_tok()

n.op1 = self.paren\_expr()

n.op2 = self.statement();

elif self.lexer.sym == Lexer.DO:

n = Node(Parser.DO)

self.lexer.next\_tok()

n.op1 = self.statement()

if self.lexer.sym != Lexer.WHILE:

self.error('"while" expected')

self.lexer.next\_tok()

n.op2 = self.paren\_expr()

if self.lexer.sym != Lexer.SEMICOLON:

self.error('";" expected')

elif self.lexer.sym == Lexer.SEMICOLON:

n = Node(Parser.EMPTY)

self.lexer.next\_tok()

elif self.lexer.sym == Lexer.LBRA:

n = Node(Parser.EMPTY)

self.lexer.next\_tok()

while self.lexer.sym != Lexer.RBRA:

n = Node(Parser.SEQ, op1 = n, op2 = self.statement())

self.lexer.next\_tok()

else:

n = Node(Parser.EXPR, op1 = self.expr())

if self.lexer.sym != Lexer.SEMICOLON:

self.error('";" expected')

self.lexer.next\_tok()

return n

def parse(self):

self.lexer.next\_tok()

node = Node(Parser.PROG, op1 = self.statement())

if (self.lexer.sym != Lexer.EOF):

self.error("Invalid statement syntax")

return node

Парсер работает рекурсивно, начиная с метода parse().  
Вначале он создает узел «Программа», дочерним узлом которого становится главный оператор программы.  
  
Операторы формируются в методе statement(). В этом методе проверяется первый токен и в зависимости от него формируется if, if/else, while, do/while, составной оператор (если начинается с фигурной скобки) или просто одиночный оператор, завершающийся точкой с запятой.  
  
При построении операторов используются методы expr() — получить выражение и paren\_expr() — получить выражение в скобках.  
  
Выражения строятся из проверок, которые строятся из сумм, которые состоят из термов. А термы в свою очередь состоят из чисел, переменных и выражений в скобках. В доме, который построил Джек.

На выходе метода parse() получаем объект класса Node, который содержит дерево нашей программы. Это дерево теперь можно преобразовывать в машинный код.

Заключение

3 Технологический раздел

3.1 Выбор среды разработки

3.2 Описание программного комплекса

3.3 Тестирование программного комплекса

[Тут](https://gist.github.com/1415759) лежит полный исходник компилятора. Я использовал скриптик для запуска и проверки (у меня Lexer читал из stdin):

echo " i = 3;" | ./cc.py

echo " { a=3; b=5; }" | ./cc.py

echo " { a = 1; b = 2; c = a + b; }" | ./cc.py

echo " { a = 5; b = 2; c = a - b; }" | ./cc.py

echo " { a = 5; b = 2; c = b < a; }" | ./cc.py

echo " { a = 5; if (a < 10) a = 33; }" | ./cc.py

echo " { a = 5; if (10 < a) a = 33; else { a = 1; b = 2; } }" | ./cc.py

echo " { a = 10; do { a = a - 2;} while (3 < a); }" | ./cc.py

echo " { a = 1; b = 5; while (a < b) { a = a + 3; }}" | ./cc.py

Заключение

Список литературы