Введение

1 Аналитический раздел

1.1 Постановка задачи

1.2 Абстрактное синтаксическое дерево

Абстрактное синтаксическое дерево (АСД) — это конечное, помеченное, ориентированное дерево, в котором внутренние вершины сопоставлены с (помечены) операторами языка программирования, а листья — с соответствующими операндами. Таким образом листья являются пустыми операторами и представляют только переменные и константы. Синтаксические деревья используются в парсерах для промежуточного представления программы между деревом разбора (конкретным синтаксическим деревом) и структурой данных, которая затем используется в качестве внутреннего представления компилятора или интерпретатора компьютерной программы для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом.

Абстрактное синтаксическое дерево отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Классическим примером такого отсутствия являются группирующие скобки, так как в АСД группировка операндов явно задаётся структурой дерева. Для языка, который описывается контекстно-свободной грамматикой, какими являются почти все языки программирования, создание абстрактного дерева в синтаксическом анализаторе является тривиальной задачей. Большинство правил в грамматике создают новую вершину, а символы в правиле становятся рёбрами. Правила, которые ничего не привносят в АСД такие, например, как группирующие правила, просто заменяются в вершине одним из своих символов. Кроме того, анализатор может создать полное дерево разбора и затем пройти по нему, удаляя узлы и рёбра, которые не используются в абстрактном синтаксисе чтобы получить АСД.

1.3 Лексический анализ

лексический анализ — процесс аналитического разбора входной последовательности символов с целью получения на выходе последовательности символов, называемых «токенами» (подобно группировке букв в словах).  Группа символов входной последовательности, идентифицируемая на выходе процесса как токен, называется лексемой. В процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.

Как правило, лексический анализ производится с точки зрения определённого формального языка или набора языков. Язык, а точнее его грамматика, задаёт определённый набор лексем, которые могут встретиться на входе процесса.

Традиционно принято организовывать процесс лексического анализа, рассматривая входную последовательность символов как поток символов. При такой организации процесс самостоятельно управляет выборкой отдельных символов из входного потока.

Распознавание лексем в контексте грамматики обычно производится путём их идентификации (или классификации) согласно идентификаторам (или классам) токенов, определяемых грамматикой языка. При этом любая последовательность символов входного потока (лексема), которая согласно грамматике не может быть идентифицирована как токен языка, обычно рассматривается как специальный токен-ошибка.

Каждый токен можно представить в виде структуры, содержащей идентификатор токена (или идентификатор класса токена) и, если нужно, последовательность символов лексемы, выделенной из входного потока (строку, число и т. д.).

Лексический анализатор обычно работает в две стадии: сканирование и оценка.

На первой стадии, сканировании, лексический анализатор обычно реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах . Например, токен «целое число» может содержать любую последовательность десятичных цифр. Во многих случаях первый непробельный символ может использоваться для определения типа следующего токена, после чего входные символы обрабатываются один за другим пока не встретится символ, не входящий во множество допустимых символов для данного токена. В некоторых языках правила разбора лексем несколько более сложные и требуют возвратов назад по читаемой последовательности.

Полученный таким образом токен содержит необработанный исходный текст (строку). Для того чтобы получить токен со значением, соответствующим типу (напр. целое или дробное число), выполняется оценка этой строки — проход по символам и вычисление значения.

Токен с типом и соответственно подготовленным значением передаётся на вход синтаксического анализатора.

На рисунке 1 изображена схема лексического анализатора.

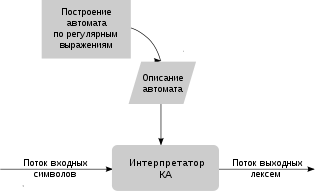


Рисунок 1. Схема лексического анализатора

1.4 Расширенная форма Бэкуса — Наура

Расширенная форма Бэкуса — Наура (расширенная Бэкус — Наурова форма (РБНФ)) — формальная система определения синтаксиса, в которой одни синтаксические категории последовательно определяются через другие. Используется для описания контекстно-свободных формальных грамматик. Предложена Никлаусом Виртом.

1.4.1Терминалы и нетерминалы

Как и в БНФ, описание грамматики в РБНФ представляет собой набор правил, определяющих отношения между терминальными символами (терминалами) и нетерминальными символами (нетерминалами).

Терминальные символы — это минимальные элементы грамматики, не имеющие собственной грамматической структуры. В РБНФ терминальные символы — это либо предопределённые идентификаторы (имена, считающиеся заданными для данного описания грамматики), либо цепочки — последовательности символов в кавычках или апострофах.

Нетерминальные символы — это элементы грамматики, имеющие собственные имена и структуру. Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов, сочетание которых определяется правилами грамматики. В РБНФ каждый нетерминальный символ имеет имя, которое представляет собой строку символов.

1.4.2Правила

Правило в РБНФ имеет вид:

идентификатор = выражение.

где идентификатор — имя нетерминального символа, а выражение — соответствующая правилам РБНФ комбинация терминальных и нетерминальных символов и специальных знаков. Точка в конце — специальный символ, указывающий на завершение правила.

Семантика правила РБНФ — нетерминальный символ, заданный идентификатором слева от знака «равно», представляет собой определяемую выражениемкомбинацию терминальных и нетерминальных символов.

Полное описание грамматики представляет собой набор правил, который последовательно определяет все нетерминальные символы грамматики так, что каждый нетерминальный символ может быть сведён к комбинации терминальных символов путём последовательного (рекурсивного) применения правил. В определении РБНФ нет никаких специальных предписаний относительно порядка записи правил, хотя такие предписания могут вводиться при использовании РБНФ программными средствами, обеспечивающими автоматическую генерацию программ синтаксического разбора по описанию грамматики.

1.4.3 Выражения

Набор возможных конструкций РБНФ очень невелик. Это конкатенация, выбор, условное вхождение и повторение.

Конкатенация. Специального обозначения не имеет, определяется последовательной записью символов в выражении. Правило вида A = BC. обозначает, что нетерминал A состоит из двух символов — B и C. Элементы конкатенации называют ещё синтаксическими факторами, или просто факторами. В данном примере B и C — синтаксические факторы. Если конкатенируемые символы могут обозначаться более чем одним знаком (как это обычно бывает при описании языков программирования), они разделяются одним или более пробельными символами (пробелы, переводы строк, табуляции). Например, Присваивание = Переменная ":=" Выражение. — здесь нетерминал Присваивание определён как конкатенация термов Переменная, «:=» и Выражение.

Выбор. Обозначается вертикальной чертой. Правило вида A = B|C|D. обозначает, что нетерминал A может состоять либо из B, либо из C, либо из D. Элементы выбора называют ещё синтаксическими термами, или просто термами. В данном примере B, C, D — синтаксические термы.

Условное вхождение. Квадратные скобки выделяют необязательный элемент выражения, который может присутствовать, а может и отсутствовать. Правило вида A = [B]. обозначает, что нетерминал A либо является пустым, либо состоит из символа B.

Повторение. Фигурные скобки обозначают конкатенацию любого числа (включая нуль) записанных в ней элементов. Правило вида A = {B}. обозначает, что A — либо пустой, либо представляет собой конкатенацию любого числа символов B (то есть A — это либо пустой элемент, либо B, либо BB, либо BBB и так далее). Если требуется, чтобы A представлял собой либо B, либо произвольное число B, но не мог быть пустым, используется запись A = B{B}.

Помимо основных операций, в РБНФ могут использоваться обычные круглые скобки. Они применяются для группировки элементов при формировании сложных выражений. Например, правило A = (B|C)(D|E). обозначает, что A состоит из двух символов, первым из которых является либо B, либо C, вторым — либо D, либо E, то есть A может быть одной из цепочек BD, BE, CD, CE.

не общепринято! Также иногда имеет смысл использовать отрицание. Например, A = (B|D)!C означает, что A может быть B или D, но не BC или DC. Такой вариант позволяет четко отличить A от G = (B|D)C и упростить процедуру разбора.

не общепринято! Определение цифры включает в себя 10 символов — от '0' до '9'. Вполне логично описать понятие «цифра» перечислением: Digit = '0' | '1' | '2' | ... | '9'. Также можно определить понятие «символ».

1.4.4достоинства и недостатки РБНФ

РБНФ, как и её предшественница — БНФ, — чрезвычайно широко применяется как средство описания искусственных языков, прежде всего — языков программирования и родственных им систем обозначений. В частности, изобретатель РБНФ, Никлаус Вирт, использовал этот формализм в своих книгах для описания всех рассматриваемых там языков программирования.

Более высокая сложность РБНФ по сравнению с БНФ приводит к тому, что генераторов программ грамматического разбора, работающих на основе РБНФ, существенно меньше, чем тех, что основаны на БНФ. Тем не менее, они существуют и применяются. РБНФ является основой для генераторов программ грамматического анализа Spirit C++ Parser Framework, Coco/R, The SLK Parser Generator, а также ряда других. Для применения в таких системах синтаксис РБНФ расширяется в том же направлении, что синтаксис БНФ при использовании парсер-генераторов yacc или bison — в описание грамматики напрямую вставляется обрабатывающий её код, так или иначе организуется взаимодействие с лексическим анализатором. Могут накладываться дополнительные ограничения и на структуру правил — не все правила, которые можно описать на РБНФ, могут быть эффективно преобразованы в код.

К безусловным достоинствам РБНФ можно отнести простоту (сам язык РБНФ содержит всего 10 специальных знаков — три вида скобок, вертикальную черту, знак «равно» и кавычки, возможно, ещё запятую; его синтаксис определяется пятью правилами), достаточную мощность и наглядность, что делает его удобным для выполнения описаний, предназначенных не только для автоматической интерпретации, но и для чтения людьми. Близость конструкций РБНФ к синтаксическим диаграммам позволяет рисовать последние прямо по РБНФ-описанию.

К недостаткам РБНФ, как, впрочем, и БНФ, можно отнести тот факт, что они описывают грамматическую структуру формального языка без учёта контекстных зависимостей, а значит, при наличии таких зависимостей РБНФ-описание оказывается неполным, и некоторые правила синтаксиса описываемого языка приходится излагать в обычной текстовой форме. Это приводит к тому, что текст, точно соответствующий РБНФ-грамматике, может, тем не менее, оказаться синтаксически некорректным. Например, в РБНФ-грамматике невозможно естественным образом отобразить тот факт, что некоторая операция требует операндов одного и того же типа. Подобные проверки приходится проводить уже написанным вручную кодом грамматического анализатора. С другой стороны, системы описания грамматики, включающие определение контекстных зависимостей, например, грамматика ван Вейнгаардена, оказываются существенно сложнее, и использование их для автоматической генерации парсеров оказывается затруднительным.

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 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

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