Національний університет “Львівська політехніка”
Інститут комп’ютерної техніки, автоматики та метрології
Кафедра СКС

Пояснювальна записка
до курсового проєкту "СИСТЕМНЕ ПРОГРАМУВАННЯ"
на тему: “РОЗРОБКА СИСТЕМНИХ ПРОГРАМНИХ МОДУЛІВ ТА КОМПОНЕНТ СИСТЕМ ПРОГРАМУВАННЯ”
Індивідуальне завдання “РОЗРОБКА ТРАНСЛЯТОРА З ВХІДНОЇ МОВИ ПРОГРАМУВАННЯ”
Варіант № 3

Виконала: ст. групи КІ-308
Бохонок О. П.

Прийняв: викладач каф. СКС
Козак Н. Б.

Львів-2024

Зміст

Анотація

Курсова робота з дисципліни "Системне програмування" вміщує в собі весь матеріал, вивчений протягом даного курсу а також задією отримані навички.

В даній роботі повинно бути продемонстровано розробку транслятора з вхідної мови програмування, яка була задана згідно з варіантом, на асемблерну мову, а саме створити виконуваний файл. Транслятор повинен виконувати: лексичний аналіз, синтаксичний аналіз, семантичний аналіз.

В цій курсовій роботі буде використовуватися лексичний аналізатор на базі скінченного автомата та на основі магазинного автомата.

Завдання на курсову роботу:

Блок тіла програми: Program <name>;

Var…; Begin-End

Оператори вводу-виводу: Get Put

Оператор присвоєння: ==>

Оператор:

- If-else (C)
- Goto (C)
- For-To (Паскаль)
- For-Downto (Паскаль)
- While (Бейсік)
- Repeat-Until (Паскаль)

Регістр ключових слів: Up-Low перший символ Up;

Регістр ідентифікаторів Up6;

Операції:

- Арифметичні: +; -; Mul; Div; Mod;
- Порівняння: Eg; Ne; >>; <<;
- Логічні: !; And; Or;

Типи даних: Int16;

Коментар: !!...!!

Вступ

Компілятор (англ. Compiler від англ. to compile збирати в ціле) - комп'ютерна програма, що перетворює (компілює) програмний код, написаний певною мовою програмування, на семантично еквівалентний код в іншій мові програмування, який, як правило, необхідний для виконання програми машиною.

Транслятор – це той самий компілятор, з тею різницею, що генерує він не об’єктний код, а код на іншій мові програмування.

Процес компіляції як правило складається з декількох етапів: лексичного, синтаксичного та семантичного (типозалежного) аналізів, генерації проміжного коду, оптимізації та генерації результуючого машинного коду. Крім цього, програма як правило залежить від сервісів, наданих операційною системою і сторонніми бібліотеками (наприклад, файловий ввід-вивід або графічний інтерфейс), і машинний код програми необхідно пов'язати з цими сервісами. Для зв'язування зі статичними бібліотеками виконується редактор зв'язків або компонувальник, а з операційною системою і динамічними бібліотеками зв'язування виконується на початку виконання програми завантажувача.

Основні задачі, які виконуються різними компіляторами та трансляторами, по суті, одні і ті ж. Розуміючи ці задачі, існує можливість створювати транслятори для різних початкових мов і цільових машин з використанням одних і тих же базових технологій.

В даній курсовій роботі створюватиметься транслятор мови програмування заданої варіантом, який включає:

- лексичний аналізатор, здатний розпізнавати лексеми, що є описані в формальному описі мови програмування.
- синтаксичний аналізатор на основі методу зсув-згортка.
- генератор коду, що генерує код який відповідає кожній конструкції вхідної мови.

Також повинне бути забезпечене виявлення лексичних та синтаксичних помилок, виконана загальна перевірка роботи компілятора.

1. Огляд методів та способів проектування трансляторів

Створення компіляторів відбувається в певних конкретних умовах: для різних мов, для різних цільових платформ, з різними вимогами для створення компіляторів. Є такі методи створення компіляторів:

1. Прямий метод - цільовою мовою і мовою реалізації є асемблер.
2. Метод розкрути - саме цей метод і використовується у даній курсовій роботі, тобто вибирається інструмент (в даній курсовій це мова асемблер), для якого вже існує компілятор.
3. Використання крос-трансляторів.
4. З використанням віртуальних машин – дає спосіб отримати переносимо програму.
5. Компіляція на ходу.

В даній курсовій роботі згідно із завданням для непарних варіантів необхідно реалізувати висхідний метод граматичного розбору.

При такій стратегії дерево синтаксичного аналізу будується, рухаючись від листя (вхідної програми, яка розглядається як рядок символів) до кореня дерева (аксіоми граматики). Аналізатор (розпізнавач) шукає частину рядка, яку можна звести до нетермінального символу. Таку частину рядка називають фразою. У більшості висхідних розпізнавачів відшукується найлівіша фраза, що безпосередньо зводиться до нетермінального символу (така фраза називається основою). Основа заміняється нетермінальним символом. У отриманому рядку знову відшукується основа, заміняється нетермінальним символом і т.д.

У загальному випадку процедуру побудови висхідного розпізнавача за заданою граматикою можна описати в такий спосіб:

1. Визначити для даної граматики функції ВПЕРВ і ВПІСЛЯ.
2. Побудувати детерміновану таблицю переходів, що має по одному стовпцю для кожного граматичного символу і по одному рядку для кожного граматичного входження і маркера дна.
3. Якщо таблиця, побудована на кроці 2, виходить недетермінованою (має більше одного стану), то потрібно перетворити цю таблицю в детерміновану, розглядаючи її як недетерміновану таблицю переходів кінцевого автомата з початковим станом h_0.
4. Стани, отримані на кроці 3 (крім стану, що відповідає порожній множині), варто використовувати як магазинні символи. Отримана таблиця переходів може містити переходи в порожню множину. Такі елементи варто розуміти як заборонені і розглядати переходи в них як помилки.

5. Керуючу таблицю заповнюють рядок за рядком відповідно до множини граматичних входжень, що позначають рядки.

Фази лексичного (ЛА) та синтаксичного (СА) аналізів розкладають початкову програму на частини. Генерація коду проміжною мовою (ГПК), оптимізація (ОК) та генерація коду (ГК) асемблера синтезують програму на вихідній мові. Керування таблицями (КТ) та обробка помилками (ОП) використовуються на всіх фазах трансляції.

Лексичний аналіз об`єднує літери в лексеми - службові слова, ідентифікатори, знаки операцій та пунктуації. Лексеми можна кодувати цілими числами, наприклад, do- одиницею, "+" - двійкою, ідентифікатор - трійкою, константу - четвіркою тощо. До коду лексем ідентифікаторів і констант додається ще одна величина - вид чи значення лексеми.

Синтаксичний аналіз групує лексеми в синтаксичні структури, які можуть бути складовими інших синтаксичних структур; наприклад, A+B може входити в оператор чи вираз. Як рекурсивні структури даних, вони зображуються (явно чи неявно) у формі дерева з лексемами в вузлах і з позначенням синтаксичних конструкцій у внутрішніх вузлах. Крони піддерев відтворюють частини програми відповідної синтаксичної конструкції.

2. Формальний опис вхідної мови програмування

2.1. Деталізований опис вхідної мови в термінах розширеної нотації Бекуса-Наура

Розширена форма Бекуса-Наура (скор. РБНФ) - формальна система визначення синтаксису, в якій одні синтаксичні категорії послідовно визначаються через інші. Використовується для опису контекстно-вільних формальних граматик. Запропоновано Никлаусом Віртом. Є розширеною переробкою форм Бекуса-Наура, відрізняється від БНФ більш «вмістимими» конструкціями, що дозволяють при тій же виразності дозволяє спростити і скоротити в обсязі опис.

label_point = "Label", ":" ;
goto_label = "Goto", ident, ";" ;
program_name = ident;
value_type = "Int16", ident, { "," , ident } ;
declaration_ident = ident;
other_declaration = ",", ident;
declaration = value_type, declaration_ident, { other_declaration } ;
operation_not = "!" , inseparable_expression ;
and_action = "And" , inseparable_expression ;
or_action = "Or" , high_priority_expression ;
equal_action = "Eg" , middle_priority_expression ;
not_equal_action = "Ne" , middle_priority_expression ;
less_or_equal_action = "<<" , middle_priority_expression ;
greater_or_equal_action = ">>" , middle_priority_expression ;
add_action = "+" , high_priority_expression ;
sub_action = "-" , high_priority_expression ;
mul_action = "Mul" , inseparable_expression ;
div_action = "Div" , inseparable_expression ;
mod_action = "Mod" , inseparable_expression ;
unary_operation = operation_not ;
inseparable_expression = group_expression | unary_operation | ident_read | value_read;
high_priority_left_expression = group_expression | unary_operation | ident_read | value_read ;
high_priority_action = mul_action | div_action | mod_action | and_action ;
high_priority_expression = high_priority_left_expression , { high_priority_action } ;

middle_priority_left_expression = high_priority_expression | group_expression | unary_operation | ident_read | value_read ;

middle_priority_action = add_action | sub_action | or_action;

middle_priority_expression = middle_priority_left_expression , { middle_priority_action } ;

low_priority_left_expression = middle_priority_expression | high_priority_expression | group_expression | unary_operation | ident_read | value_read ;

low_priority_action = less_or_equal_action | greater_or_equal_action | equal_action | not_equal_action ;

low_priority_expression = low_priority_left_expression , { low_priority_action } ;

group_expression = "(" , low_priority_expression , ")" ;

bind = ident_write , "==>", low_priority_expression ;

if_expression = expression

body_for_true = {statement}, tokenSEMICOLON ";"

body_for_false = tokenELSE, "Else"; {statement}, "statement"; tokenSEMICOLON ";"

cond_block = tokenIF, "If"; if_expression; body_for_true; [body_for_false]

cycle_begin_expression = low_priority_expression;

cycle_counter = ident;

cycle_counter_last_value = value;

cycle_body = "Do", statement, {statement};

forto_cycle = "For", cycle_begin_expression, "==>", cycle_counter, "To", cycle_counter_last_value, cycle_body, ",";

while_cycle_head_expression = low_prioryty_expression

while_cycle = "While" , while_cycle_head_expression , { statement } , ";" ;

tokenCONTINUE = "Continue" ;

tokenWHILE = "While" ;

tokenEXIT = "Exit" ;

continue_while = tokenCONTINUE , tokenWHILE ;

exit_while = tokenEXIT , tokenWHILE ;

statement_in_while_body = statement | continue_while | exit_while ;

repeat_until_cycle_cond = low_prioryty_expression ;

repeat_until_cycle = "Repeat" , { statement } , "Until" , repeat_until_cycle_cond ;

input = "Get" , "(" , ident_write , ")" ;

output = "Put", "(", low_priority_expression, ")";

statement = recursive_descent_end_point | bind | if_block | for_downto_cycle | while_cycle | do_while_cycle | label_point | goto_label | input | output;

program = "Program" , program_name , ";" , "Var", variable_declaration, ";", "Begin" , { statement } , "End" ;

digit = digit_0 | digit_1 | digit_2 | digit_3 | digit_4 | digit_5 | digit_6 | digit_7 | digit_8 | digit_9;

non_zero_digit = digit_1 | digit_2 | digit_3 | digit_4 | digit_5 | digit_6 | digit_7 | digit_8 | digit_9;

unsigned_value = (non_zero_digit , { digit } | "0") ;

value = [sign] , unsigned_value ;

letter_in_lower_case = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

letter_in_upper_case = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";

ident = letter_in_upper_case , letter_in_upper_case , letter_in_upper_case , letter_in_upper_case , letter_in_upper_case , letter_in_upper_case ;

2.2. **Термінальні символи та ключові слова.**

Якщо ми маємо якусь мову задану граматикою, то термінальними символами є всі символи, які з'являються в конструкціях мови. Термінальний означає кінцевий.

“Program<name>” – початок програми

“Var” – блок даних.

“Begin” – початок блоку коду.

“End” – кінець тіла програми (циклу).

“Get” – оператор вводу змінних.

“Put” – оператор виводу (змінних і рядкових констант).

“==>” - оператор присвоєння.

“ ” - пуста строка

“;”-кінець рядка програми

“If-else” – умовний оператор.

“Goto” – оператор безумовного переходу.

“For-To” – оператор циклу з нарощуванням.

“For-Downto” – оператор циклу зі зменшенням.

“While” – оператор циклу з умовою на початку.

“Repeat-Until “– оператор циклу з умовою на кінці.

“+” - операція додавання.

“-” - операція віднімання.

“Mul” – операція множення.

“Div” – операція ділення.

“Mod” – операція знаходження залишку від ділення.

“Eg” – рівність.

“Ne” – нерівність.

“>>” - більше.

“<<” - менше.

“!” – операція логічного заперечення.

“And” – кон’юнкція.

“Or” – диз’юнкція.

“Int16” – Тип даних для 16-розрядних цілих.

“!!..!!”... – коментар.

“” – початок/завершення рядкової константи при операції виводу;

“,” – розділювач між деклараціями змінних;

“(“– відкриваюча дужка;

“)” – закриваюча дужка;

Як термінальні символи використовуються також усі арабські цифри (0–9), латинські букви (a-z, A-Z), символи табуляції, символ переходу на нову стрічку, пробіл.

4.4 Перевірка роботи транслятора за допомогою тестових задач.

4.3.1. Тестова програма «Лінійний алгоритм»

1. Ввести два числа A і B (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання).

2. Вивести на екран:

A + B (результат операції додавання);

A - B (результат операції віднімання);

A * B (результат операції множення);

A / B (результат операції ділення);

A % B (результат операції отримання залишку від ділення).

3. Обрахувати значення виразів

X = (A - B) * 10 + (A + B) / 10

У = X + X % 10

4. Вивести значення X і У на екран.

Напишемо програму на вхідній мові програмування:

```
Program TASKKK;

Var
  Int16 NUMAAA, NUMBBB, RESXXX, RESYYY;

Begin

  Get NUMAAA;
  Get NUMBBB;

  Put NUMAAA + NUMBBB;

  Put NUMAAA - NUMBBB;

  Put NUMAAA Mul NUMBBB;

  If NUMBBB Ne 0
    Put NUMAAA Div NUMBBB;
  Else
    Put 'Division by zero!';

  If NUMBBB Ne 0
    Put NUMAAA Mod NUMBBB;
  Else
    Put 'Modulo by zero!';

  (NUMAAA - NUMBBB) Mul 10 + (NUMAAA + NUMBBB) Div 10 ==> RESXXX;
  RESXXX + RESXXX Mod 10 ==> RESYYY;

  Put RESXXX;
  Put RESYYY;
End
```

4.3.2. Тестова програма «Алгоритм з розгалуженням»

1. Ввести три числа A, B, C (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання). Використання вкладеного умовного оператора:

2. Знайти найбільше з них і вивести його на екран. Використання простого умовного оператора:

3. Вивести на екран число 1, якщо усі числа однакові (логічний вираз в умовному операторі має виглядати так: «(A=B) і (A=C) і (B=C)»), інакше вивести 0.

4. Вивести на екран число -1, якщо хоча б одне з чисел від’ємне (логічний вираз в умовному операторі має виглядати так: «(A<0) або (B<0) або (C<0)»), інакше вивести 0.

5. Вивести на екран число 10, якщо число A більше за суму чисел B і C (логічний вираз в умовному операторі має виглядати так: «!(A<(B+C))»), інакше вивести 0.

```
Program BRANCH;

Var
  Int16 NUMAAA, NUMBBB, NUMCCC, MAXNUM;

Begin

  Get NUMAAA;
  Get NUMBBB;
  Get NUMCCC;

  If NUMAAA >> NUMBBB
    If NUMAAA >> NUMCCC
      NUMAAA ==> MAXNUM
    Else
      NUMCCC ==> MAXNUM
  Else
    If NUMBBB >> NUMCCC
      NUMBBB ==> MAXNUM
    Else
      NUMCCC ==> MAXNUM;

  Put MAXNUM;

  If (NUMAAA Eg NUMBBB) And (NUMAAA Eg NUMCCC) And (NUMBBB Eg NUMCCC)
    Put 1
  Else
    Put 0;

  If (NUMAAA << 0) Or (NUMBBB << 0) Or (NUMCCC << 0)
    Put -1
  Else
    Put 0;

  If !(NUMAAA << (NUMBBB + NUMCCC))
    Put 10
  Else
    Put 0;
End
```

4.3.3 Тестова програма «Циклічний алгоритм»

1. Ввести два числа A і B, причому A<B (імена змінних можуть бути іншими і мають відповідати правилам запису ідентифікаторів згідно індивідуального завдання). Використання простого оператора циклу:

2. Вивести на екран квадрати чисел від A до B включно. Використання вкладеного оператора циклу:

3. Обрахувати X=A*B за наступним алгоритмом: X = 0 Цикл від 1 до A з кроком 1 Цикл від 1 до B з кроком 1 X = X + 1

4. Вивести значення X на екран.

```
Program LOOPAL;

Var
  Int16 NUMAAA, NUMBBB, RESXXX;
  Int16 IIIIII, JJJJJJ;

Begin
  Get NUMAAA;
  Get NUMBBB;

  If NUMAAA << NUMBBB
  Begin
    For NUMAAA ==> IIIIII To NUMBBB Do
      Put IIIIII Mul IIIIII;

    RESXXX ==> 0;
    For 1 ==> IIIIII To NUMAAA Do
      For 1 ==> JJJJJJ To NUMBBB Do
        RESXXX + 1 ==> RESXXX;

    Put RESXXX;
  End
  Else
    Put 'NUMAAA must be less than NUMBBB!';
End
```