С.О. Семеріков, І.О. Теплицький
Криворізький державний
педагогічний університет

Побудова найпростішого інтерпретатору в процесі вивчення теми “Основи компіляції”

Перше знайомство з процесом компіляції відбувається при вивченні основ програмування в старших класах середньої школи [1], тому в процесі співбесіди з шкільного курсу інформатики абітурієнти, як правило, досить чітко означають поняття компілятора (транслятора) мови програмування, процесу компіляції, інтегрованого середовища програмування тощо.

Поглиблення цих понять відбувається на І курсі фізико-математичних факультетів педагогічних ВНЗ при вивченні теми “Основи компіляції”. У лекції формалізуються основні поняття процесу компіляції, розглядається типова структура компілятора, інтегровані середовища та вимоги до розробки компіляторів [2]. Виходячи з того, що студенти постійно користуються компіляторами для створення власних програм, доцільно на практичному занятті із вказаної теми перейти від розгляду інтерфейсу конкретного середовища програмування до програмної реалізації наступної задачі:

Створити програму для обчислення арифметичних виразів, що містять цілі числа, з’єднані операціями додавання та множення; вирази можуть бути згруповані за допомогою дужок.

В цій задачі пропонується створити найпростіший інтерпретатор арифметичних виразів. Вибір цілих чисел та лише двох операцій зумовлений їх мінімальністю. Подальші модифікації програми (зміна типу даних, реалізація операцій множення, ділення тощо) є суто косметичними і можуть бути запропоновані як домашнє завдання.

Для визначення того, з яких компонентів буде складатися програма, спочатку пропонуємо розглянути декілька арифметичних виразів: 1+2+3*4, 76*(4+8)*2, -123 тощо. В процесі виконання цієї роботи звертаємо увагу на те, що вказані вирази

складаються з цілих чисел, знаків “+”, “*”, “(”, “)”, після чого робимо висновок про те, що саме ці елементи будуть головними лексичними одиницями програми. При цьому операції та дужки задаються лише одним знаком, тоді як число може складатися з декількох.

Дамо таке визначення: ціле число – це “0” або послідовність цифр, що починається з будь-якого елемента множини [“1”..“9”], за яким слідує нуль або більше будь-яких елементів множини [“0”..“9”]. Цілому числу може передувати необов’язковий знак “-”.

Використовуючи регулярні вирази, ціле число можна було б означити так:

```
[-]?([1-9][0-9]*|0)
```

Тут “?” означає необов’язковий елемент, круглі дужки використовуються для групування, “*” (зірочка Кліні) означає повторення попереднього елемента нуль та більше разів, “|” – операцію “або”.

У якості ознаки закінчення введення виразу та необхідності початку його обчислення оберемо клавішу “Enter”, а для завершення програми використаємо будь-яке доцільне слово (наприклад, “quit” – “вихід”). Нарешті, звертаємо увагу на те, що користувачем програми можуть бути введені й будь-які інші символи, що не повинні входити до складу арифметичного виразу. Ці три випадки також вважатимемо окремим лексемами.

Таким чином, перша частина програми – лексичний аналізатор – повинна розпізнавати 8 типів лексем.

Написання лексичного аналізатору є хоч і простою за структурою, проте трудомісткою роботою [3], яка в студентів-першокурсників відніме значний час. З метою його економії (адже при написанні програми ми повинні вкластися у 2 навчальні години) застосуємо програмний засіб для генерації лексичних аналізаторів – flex (lex), на вхід якого подамо файл наступної структури:

```
%{
  enum TOKENS {INTEGER=1, PLUS, UMNOJ, OTKSK, ZAKSK, ENDSTR, QUIT, NERASP};
%}

int       [-]?([1-9][0-9]*|0)
%%
{int}     return INTEGER;
"+"       return PLUS;
"*"       return UMNOJ;
"("       return OTKSK;
```

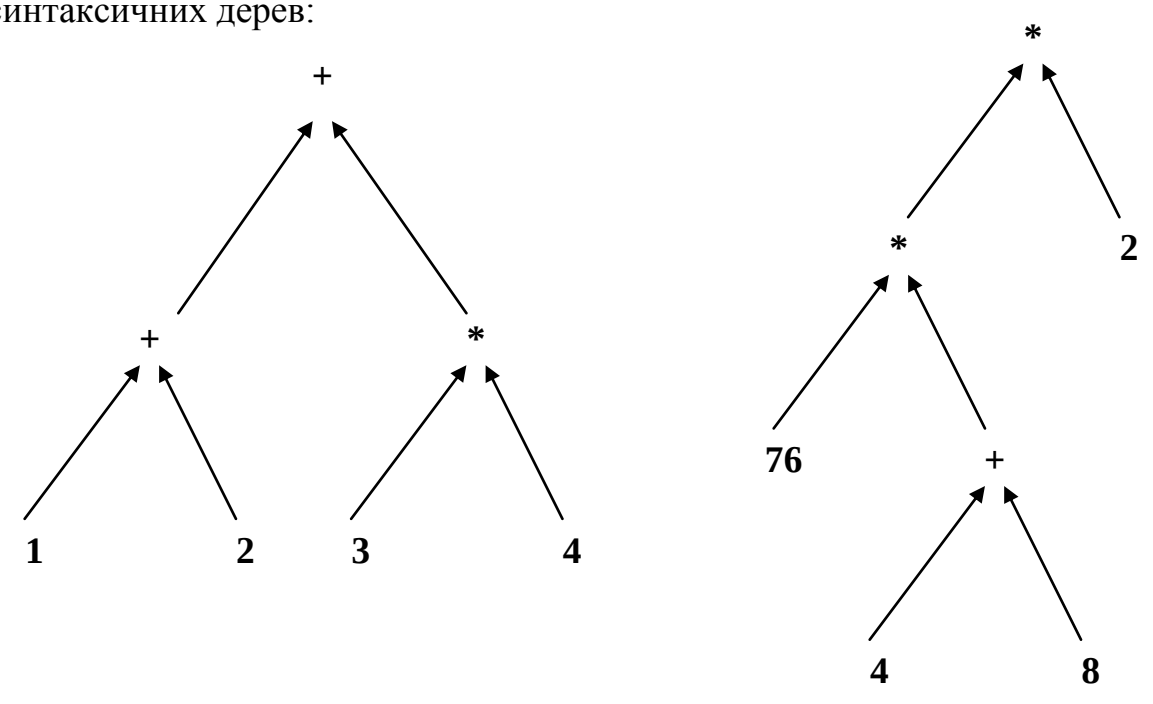
```
")"       return ZAKSK;
"quit"    return QUIT;
\n        return ENDSTR;
[\t ]     ;
.         return NERASP;
%%

int yywrap()
{
  return 1;
}
```

TOKENS є переліком 8 визначених лексем : ціле число, додавання, множення, відкриваюча та закриваюча круглі дужки, ознаки завершення рядка, програми та нерозпізнаного символу. Ціле число означаємо правилом розпізнавання {int}, що містить відповідний регулярний вираз, усі інші лексеми – відповідними символьними послідовностями: “+”, “*”, “(”, “)”, “quit”, “\n” (клавіша Enter), “.” (позначення будь-якого іншого символу, відмінного від описаних вище). Символи пробілу та табуляції ігноруватимемо як незначущі.

Після обробки наведеної лексичної структури flex-ом студенти отримують файл з текстом функції лексичного аналізу yylex(), кожен виклик якої повертає одну лексему, для зберігання якої в програмі передбачимо відповідну змінну:

```
int lexema;
```

Наступній частині роботи – побудові синтаксичного аналізатору – передує повернення до розгляду записаних арифметичних виразів. 1+2+3*4, 76*(4+8)*2 тощо з метою визначення правил їх обчислення. Для цього спочатку визначаємо пріоритет виконання кожної з операцій (дужки – вищій, множення – середній, додавання – нижчий), а потім зображаємо процес обчислення вказаних виразів у вигляді синтаксичних дерев:



З рисунків видно, що обчислення починається з найнижчих гілок дерева і виконується у порядку, в якому читається арифметичний вираз – зліва направо. Дужки на синтаксичному дереві не показуються, проте в процесі його побудови взятий у дужки вираз розташовується на рівень нижче.

Будь-який арифметичний вираз E можна подати у вигляді суми:

E=T+T+T+..., де T – інший вираз, що складається з множників:

T=F*F*F*..., де F – або число, або будь-який арифметичний вираз у дужках:

F=(E),

F=число.

Доданки та множники у виразах E і T повторюються щонайменше один раз, тому в термінах регулярних виразів правила обчислення (граматика) будь-якого арифметичного виразу можуть бути записані так:

1) E → T(“+” T)*

2) T → F(“*” F)*

3) F → “(” E “)”

4) F → “число”

Тут для уникнення плутанини зірочки Кліні та зірочки–множення, дужок групування та дужок в запису арифметичного виразу усі лексеми взяті у лапки.

Визначивши правила обчислення, наводимо приклади їх застосування для синтаксичного аналізу наведених вище прикладів, на кожному кроці застосовуючи лише одно правило, та порівнюємо цей процес із побудовою синтаксичного дерева:

$E \Rightarrow T + T + T \Rightarrow T + T + F * F \Rightarrow F + T + F * F \Rightarrow F + F + F * F \Rightarrow 1+2+3*4$

$E \Rightarrow T \Rightarrow F * F * F \Rightarrow F * (E) * F \Rightarrow F * (T + T) * F \Rightarrow F * (F + T) * F \Rightarrow$

$\Rightarrow F * (F + F) * F \Rightarrow 76*(4+8)*2$

Формалізація запису процесу обчислення за допомогою правил 1–4 дає підстави стверджувати, що його програмна реалізація – синтаксичний аналізатор являтиме собою послідовність виклику функцій E, T та F. Процес обчислення починається з виклику функції E, тому організуємо в програмі наступний цикл:

```
виконувати дії
```

```
      |  отримати наступну лексему
      |  викликати функцію E та отримати результат обчислення виразу
      |  роздрукувати результат обчислення виразу
      до тих пір, поки не буде введена ознака завершення програми
```

Програмна реалізація цього циклу матимемо такий вигляд:

```
main()// головна функція програми
{
  int result; // змінна для зберігання результату

  printf("Калькулятор: +,*,(,)\n\n"); // заставка програми
  do { // виконувати дії
    lexema=yylex(); // отримати наступну лексему
    result=E(); // викликати функцію E та отримати результат обчислення
    printf("%d\n",result); // роздрукувати результат обчислення виразу
  }while(lexema!=QUIT) // поки не буде введена ознака завершення програми
}
```

У відповідності до правила 1, функція E повинна виконувати наступні дії:

Алгоритм	Програмна реалізація

викликати функцію T та зберегти її результат
поки зчитана лексема – знак "+"
| отримати наступну лексему
| викликати функцію T, додавши її результат
| до попереднього
-
повернути результат з функції
	int E()
	{
	  int result=T();
	  while(lexema==PLUS)
	  {
	    lexema=yylex();
	    result=result+T();
	  }
	  return result;
	}

У відповідності до правила 2, функція T повинна виконувати наступні дії:

Алгоритм	Програмна реалізація

викликати функцію F та зберегти її результат
поки зчитана лексема – знак "*"
| отримати наступну лексему
| викликати функцію T, помноживши її
| результат на попередній
-
повернути результат з функції
	int T()
	{
	  int result=F();
	  while(lexema==UMNOJ)
	  {
	    lexema=yylex();
	    result=result*F();
	  }
	  return result;
	}

Цикл while у функціях E і T реалізує операції додавання та множення цілих чисел. Якщо ці операції відсутні, то маємо справу або з виразом у дужках, або просто з числом, що й реалізуємо у функції F.

Функція F реалізує одразу 2 правила – 3 та 4. Правило 3 стверджує, що, якщо вираз починається з відкриваючої дужки, слід викликати функцію E, після чого перевірити, чи є наступна лексема закриваючою дужкою. Порушення парності дужок вважатимемо помилкою. До правила 4 переходимо у випадку, коли правило 3 не

виконується. Єдиний допустимий тип лексеми в цьому правилі – ціле число, усі інші вважатимемо помилковими. За будь-яких обставин помилку необхідно діагностувати та, за можливістю, відновити аналіз виразу (наприклад, як це пропонується у [4]).

Алгоритм	Програмна реалізація

якщо зчитана лексема знак "(", то правило 3:
|
| отримати наступну лексему
| викликати функцію E
| якщо зчитана лексема – знак ")"
| | отримати наступну лексему
| інакше
| | помилка парності дужок
| -
інакше - переходимо до правила 4
| якщо зчитана лексема – ціле число
| | отримати результат зі змінної yytext,
| | що визначена функцією yylex(),
| | перетворивши yytext на ціле число
| | отримати наступну лексему
| |
| інакше
| |
| | помилка введення не числової лексеми
| | там, де вона очікувалася
| -
-
повернути результат з функції
	int F()
	{
	int result;
	
	if(lexema==OTKSK)
	{
	lexema=yylex();
	result=E();
	if(lexema==ZAKSK)
	lexema=yylex();
	else
	puts("Помилка: немає)");
	}
	else
	{
	if(lexema==INTEGER)
	{
	result=atoi(yytext);
	lexema=yylex();
	}
	else
	{
	puts("Помилка: не число");
	result=0;
	}
	}
	return result;
	}

Для унаочнення процесу обчислення виразу можна запропонувати доповнити кожну з наведених функцій налагоджувальним виводом. Для цього доцільно визначити макропідстановку DEBUG, в залежності від якої працюватимуть директиви умовної компіляції #ifdef – #endif. Тоді, наприклад, функція F може мати такий вигляд:

```
int F()
{
  int result; // змінна для зберігання результату

#ifdef DEBUG
  puts("F почалась");
#endif
  if(lexema==OTKSK) // якщо зчитана лексема знак "(", то правило 3:
  {
    lexema=yylex(); // отримати наступну лексему
#ifdef DEBUG
    puts("Виклик E з F");
```

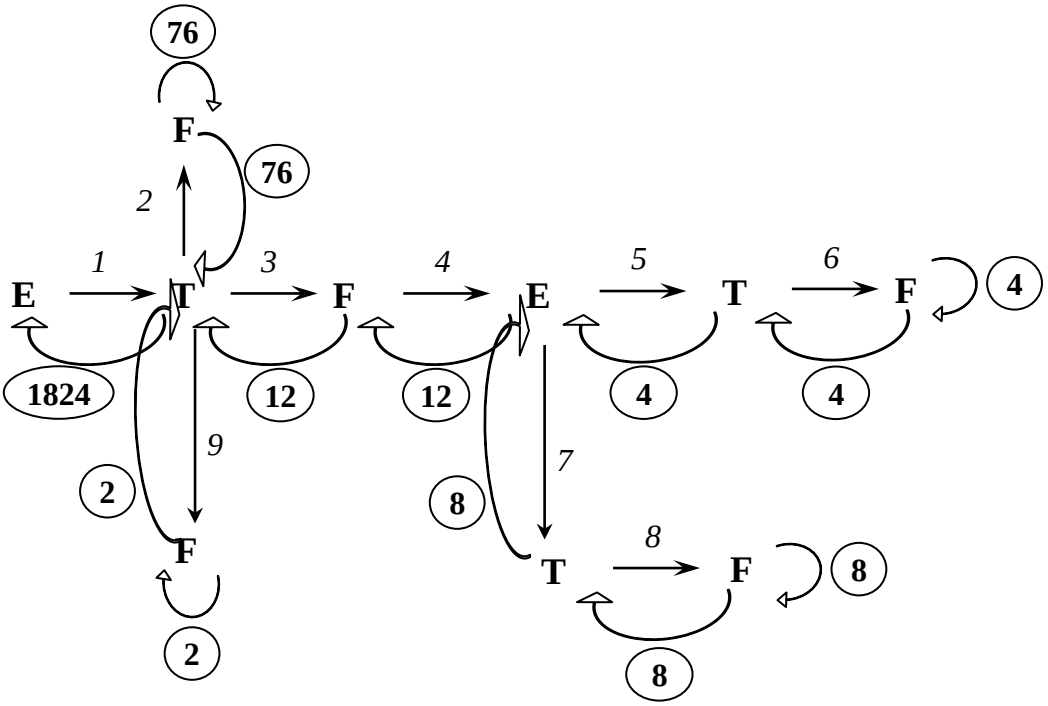
```
#endif
    result=E(); // викликати функцію E
    if(lexema==ZAKSK) // якщо зчитана лексема – знак ")"
      lexema=yylex(); // отримати наступну лексему
    else
      puts("Помилка: немає )"); // помилка парності дужок
  }
  else // інакше - переходимо до правила 4
  {
    if(lexema==INTEGER) // якщо зчитана лексема – ціле число
    {
#ifdef DEBUG
      printf("ЦІЛЕ ЧИСЛО: %s\n",yytext);
#endif
      result=atoi(yytext); /* отримати результат зі змінної yytext, що
                              визначена функцією yylex(),перетворивши yytext
                              на ціле число */
      lexema=yylex(); // отримати наступну лексему
    }
    else
    {
      puts("Помилка: не число"); // помилка введення не числової лексеми
      result=0;
    }
  }
#ifdef DEBUG
  printf("F закінчилась, результат=%d\n",result);
#endif
  return result; // повернути результат з функції
}
```

Приклад діалогу з побудованою програми у налагоджувальному режимі:

```
76*(4+8)*2
E почалась
   Виклик T з E
      T почалась
         Виклик F з T
            F почалась
               ЦІЛЕ ЧИСЛО: 76
            F закінчилась, результат=76
         Виклик F з T
            F почалась
               Виклик E з F
                  E почалась
                     Виклик T з E
                        T почалась
                           Виклик F з T
                              F почалась
                                 ЦІЛЕ ЧИСЛО: 4
                              F закінчилась, результат=4
                        T закінчилась, результат=4
                     Виклик T з E
                        T почалась
                           Виклик F з T
                              F почалась
                                 ЦІЛЕ ЧИСЛО: 8
                              F закінчилась, результат=8
                        T закінчилась, результат=8
                  E закінчилась, результат=12
            F закінчилась, результат=12
```

```
          Виклик F з T
             F почалась
                ЦІЛЕ ЧИСЛО: 2
             F закінчилась, результат=2
       T закінчилась, результат=1824
E закінчилась, результат=1824
1824
quit
```

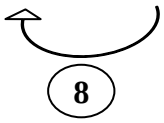
Відповідна схема викликів функцій в процесі синтаксичного аналізу:



Умовні позначення:

E – ім’я функції, що викликається;

1 → – номер виклику наступної функції;



– значення, що повертається у попередню функцію.

Запропонований підхід до побудови практичного заняття дозволяє на початку вивчення курсу інформатики наповнити практичним змістом поняття інтерпретації, лексичного та синтаксичного аналізу. Застосування генератора лексичних аналізаторів при цьому суттєво полегшує процес написання програми, дозволяючи максимально наблизити текст програмної реалізації до запису алгоритму.

У подальшому створена програма може бути використана як основа для створення компілятора з мови програмування, системи символьної математики та як елемент деякої інтегрованої системи для обчислення формул, що задаються користувачем.

Література:

1. Жалдак М.І., Морзе Н.В., Мостіпан О.І., Науменко Г.Г. Інформатика. 10-11 класи. Програми для загальноосвітніх навчальних закладів. – Кам’янець-Подільський: Абетка–НОВА, 2002. – 80 с.
2. Семеріков С.О., Теплицький І.О. Методичні аспекти вивчення теми “Основи компіляції” у підготовці майбутнього вчителя інформатики // Рідна школа. – 2004. – №4. – С. 32–33.
3. Полищук А.П., Семериков С.А. О реализации практикума по программированию лексических и синтаксических анализаторов при создании языковых интерпретаторов // Теорія та методика навчання математики, фізики, інформатики: Збірник наукових праць. Випуск 4: В 3-х томах. – Кривий Ріг: Видавничий відділ НМетАУ, 2004. – Т. 3: Теорія та методика навчання інформатики. – С. 250–259.
4. Шилдт Г. Полный справочник по C. – 4-е издание. – М.: Издательский дом «Вильямс», 2000.