3. Синтактичен анализ. Граматика на учебен програмен език STUDENT. Извеждане на изречения от правилата на граматиката

Цел на упражнението

Упражнението представя кратко въведение в синтактичния анализ. Дадена е граматиката на входния език на учебния компилатор, която дефинира неговия синтаксис. Демонстрира се как чрез правилата на граматиката се получават валидни програми на езика.

3.1. Същност и цел на синтактичния анализ

Основната задача на синтактичния анализ е разпознаване на изречение на входния език. В термините на синтактичния анализ изречение означава синтактично коректна програма на входния език. Синтактичният анализатор (нар. още парсер) проверява дали дадена входна програма е валидно изречение на входния език, т.е. дали изречението се поражда в резултат от краен брой прилагания на правилата на граматиката.

На входа на синтактичния анализатор се получава поток от лексеми, получен от лексическия анализатор. Резултатът от работата на синтактичния анализатор е синтактично дърво на входната програма.

На етапа на синтактичния анализ се откриват синтактични грешки, напр.:

- Незатворени скоби
- Липсваща точка и запетая
- Невалидни оператори

3.2. Програмен език STUDENT

За целите на учебния компилатор е разработен програмен език, наречен STUDENT. Езикът има следните общи функционалности:

- Типове данни
  - целочислен тип
  - булев тип
  - символен тип
  - едномерен масив
- Променливи
  - глобални
  - локални
- Изрази
  - аритметични
  - логически
- Оператори
  - аритметични оператори(+, -, *, /, %)
  - логически оператори(and, or, not)
  - оператори за сравнение(<, >, <=, >=, !=, ==)
  - оператор за присвояване(=)
  - унарни оператори(-)
  - условен оператор(if else)

    - оператор за цикъл(while)
    - съставен оператор({ … })
    - входно-изходни оператори(read, print)
- Рекурсия
- Character escaping (напр. ‘\n’, ‘\t’)
- Подпрограми - функции, които приемат параметри и връщат резултат

С цел опростяване на реализацията на компилатора в езика STUDENT са въведени ограничения, най-важните от които са:

- Езикът не е обектно-ориентиран
- Поддържа три прости типа данни (целочислен, булев, символен) и един структуриран тип данни (само едномерен масив)
- Поддържа само подпрограми функции

Примерна програма на STUDENT за изчисляване на 10!:

```
program {
 fact(int num) -> int {
      if (num == 0) { return 1; }
        return num * @fact(num - 1);
   }
   main() -> void {
      print(@fact(10));
   }
}
```

3.3. Формална граматика

Синтаксисът на всеки програмен език се описва с подходяща граматика. Синтаксисът на езика STUDENT, както и синтаксисът на повечето езици за програмиране, се дефинира с контекстно-свободна граматика (граматика от тип 2 в йерархията на Хомски).

Граматиката се състои от правила (продукции). В правилата участват нетерминални, терминални и специални символи. Нетерминален символ (или по-кратко нетерминал) е символ, който се замества с други символи от граматиката. Терминален символ (или по-кратко терминал) е символ, който не може да се замести с други символи. Специалните символи изразяват някои специфични операции при прилагането на правилата като алтернативни клонове на правило, повторение на символи и др.

Всяко правило се състои от лява и дясна страна. В контекстно-свободните граматики лявата страна на правилото съдържа единствен нетерминален символ, а дясната страна съдържа низ от терминални и/или нетерминални символи или празен низ.

В граматиката се използват специалните символи ‘(’, ‘)’, ‘|’, ‘?’, ‘*’, ‘+’ със следния смисъл:

- ‘(‘ част от правило ‘)’ – изпълнява се точно 1 път

- правило 1 ‘|’ правило 2 – прилагане на правило 1 ИЛИ правило 2

- ‘?’ – прилагане на част от правило 0 или 1 път

- ‘*’ – прилагане на част от правило 0 или повече пъти

- ‘+’ – прилагане на част от правило 1 или повече пъти

3.4. Граматика на STUDENT

Граматиката на езика STUDENT е представена в БНФ (Бакус-Наур форма).

(1) program : PROGRAM LBRACKET program_body RBRACKET

(2) program_body : (variable_definition SEMICOLON | function_definition)* main_function

(3) main_function : 'main' LPAREN RPAREN ARROW VOID block

(4) function_definition : IDENTIFIER LPAREN (formal_parameters)? RPAREN ARROW (VOID | type) block

(5) formal_parameters : type IDENTIFIER (COMMA type IDENTIFIER)*

(6) type : primitive_type | array_type

(7) primitive_type : INT | CHAR | BOOLEAN

(8) array_type : primitive_type LSQUARE RSQUARE

(9) block : LBRACKET (statement)* RBRACKET

(10) statement : simple_statement SEMICOLON | compound_statement

(11) simple_statement : variable_definition | assignment | function_call | return_statement | print_statement | read_statement

(12) compound_statement : if_statement | while_statement

(13) variable_definition : type assignment

(14) assignment : variable BECOMES assignable

(15) function_call : AT IDENTIFIER LPAREN (actual_parameters)? RPAREN

(16) return_statement : RETURN (assignable)?

(17) print_statement : PRINT LPAREN assignable (COMMA assignable)* RPAREN

(18) read_statement : READ LPAREN variable (COMMA variable)* RPAREN

(19) if_statement : IF LPAREN expression RPAREN block | IF LPAREN expression RPAREN block ELSE block

(20) while_statement : WHILE LPAREN expression RPAREN block

(21) actual_parameters : assignable (COMMA assignable)*

(22) assignable : array_initialization | character_literal | string_literal | expression

(23) array_initialization : primitive_type LSQUARE expression RSQUARE

(24) character_literal : SINGLE_QUOTE CHAR_LITERAL SINGLE_QUOTE

(25) string_literal : DOUBLE_QUOTES (CHAR_LITERAL)* DOUBLE_QUOTES

(26) expression : simple_expression | simple_expression relational_operator simple_expression

(27) simple_expression : signed_term ((additive_operator | OR) signed_term)*

(28) signed_term : (unary_operator)? term

(29) term : factor ((multiplicative_operator | AND) factor)*

(30) factor : variable
| INT_LITERAL
| BOOLEAN_LITERAl
| LPAREN expression RPAREN
| function_call
| array_length

(31) relational_operator : EQUAL
| NOT_EQUAL
| GREATER_THAN
| LESS_THAN
| GREATER_THAN_OR_EQUAL
| LESS_THAN_OR_EQUAL

(32) unary_operator : NOT | MINUS

(33) additive_operator : PLUS | MINUS

(34) multiplicative_operator : MUL | DIV | MOD

(35) variable : IDENTIFIER | index_variable

(36) index_variable : IDENTIFIER LSQUARE simple_expression RSQUARE

(37) array_length : LENGTH LPAREN variable RPAREN

В правилата на граматиката е използвана конвенцията нетерминалните символи да бъдат означени с малки букви, а терминалните символи с главни букви.

Примери за интерпретиране на правилата:

Пример 1. Правило (1)

program : PROGRAM LBRACKET program_body RBRACKET

Нетерминалните символи в правилото са program и program_body, а терминалните символи са PROGRAM, LBRACKET и RBRACKET.

При прилагане на правилото нетерминалният символ program се замества с низ, съставен от символите PROGRAM, LBRACKET, program_body и RBRACKET. Смисълът на това правило е, че програма на езика STUDENT започва с ключова дума ‘program’

(терминала PROGRAM), следвана от ‘{’ (терминала LBRACKET), тяло на програмата (терминала ‘program_body’) и завършва с ‘}’ (терминала RBRACKET).

Пример 2. Правило (2)

(2) program_body : (variable_definition SEMICOLON | function_definition)* main_function

Нетерминалните символи в правилото са program_body, variable_definition, function_definition и main_function, а терминален символ е SEMICOLON. В правилото са използвани следните специални символи:

- ‘|’ – означава, че при прилагане на правилото се използва един от двата алтернативни клона – variable_definition SEMICOLON или function_definition

- ‘*’ – означава, че частта от правилото, заградена в скоби, може да се пропусне или да се прилага произволен брой пъти

Смисълът на правилото е, че тялото на програма на езика STUDENT може да започва с произволен брой дефиниции на променливи (variable_definition SEMICOLON), или дефиниции на функции (function_definition), следвани от главна функция (main_function). Правилото допуска отсъствие на дефиниции на променливи и функции, т.е. единственият задължителен елемент от тялото на програмата е главна функция.

Пимер 3. Правило (15)

(15) function_call : AT IDENTIFIER LPAREN (actual_parameters)? RPAREN

Нетерминални символи са function_call и actual_parameters, а терминални са символите AT, IDENTIFIER, LPAREN и RPAREN. Специалният символ ‘?’ означава, че при прилагането на правилото символът actual_parameters може да се пропусне или да се приложи точно веднъж. Правилото дефинира, че извикване на функция в програма на STUDENT се състои от следните елементи:

- започва със символ ‘@’ (AT)

- следва идентификатор (името на функцията, IDENTIFIER)

- следва лява скоба ‘(’ (LPAREN)

- може да има веднъж актуални параметри или да няма (заради специалния символ ‘?’)

- завършва с дясна скоба ‘)’ (RPAREN)

3.5. Примери за валидни изречения на езика STUDENT

Пример 1. Да се състави програма на STUDENT, която извежда на екран низ „Hello, world!“. Да се демонстрира пораждането на изречението от правилата на граматиката.

Програмата представлява валидно изречение на езика. Пораждането на изречение от граматиката започва от стартовия символ, който винаги е нетерминален. Правилата могат да се прилагат в произволен ред, произволен брой пъти, но винаги по едно правило на всяка стъпка. При прилагане на правило нетерминален символ се замества с низ от терминални и/или нетерминални символи в съответствие с приложеното правило. Прилагането на правилата продължава до получаване на низ, в който участват само

терминални символи. Този низ е валидно изречение на входния език, защото е получен чрез правилата на граматиката.

По-долу е показана примерна последователност от прилагане на правилата за пораждане на изречение, т.е. програма, която извежда на екран низ „Hello, world!“. Стартовият символ на граматиката е нетерминалният символ program (правило 1), т.е. първото приложено правило трябва да бъде правило 1. Терминалните символи са заградени с апострофи, за да бъдат лесно различавани от нетерминалните символи.

Правило 1
program ----------> ‘PROGRAM’ ‘{’ program_body ‘}’

Правило 2
----------> ‘PROGRAM’ ‘{’ main_function ‘}’

Правило 3
----------> ‘PROGRAM’ ‘{’ ‘main’ ‘(’ ‘)’ ‘->’ ‘VOID’ block ‘}’
Правило 9
----------> ‘PROGRAM’ ‘{’ ‘main’ ‘(’ ‘)’ ‘->’ ‘VOID’ ‘LBRACKET’ statement ‘RBRACKET’ ‘}’

Правило 10
----------> ‘PROGRAM’ ‘{’ ‘main’ ‘(’ ‘)’ ‘->’ ‘VOID’ ‘LBRACKET’ simple_statement ‘SEMICOLON’ ‘RBRACKET’ ‘}’

Правило 11
----------> ‘PROGRAM’ ‘{’ ‘main’ ‘(’ ‘)’ ‘->’ ‘VOID’ ‘LBRACKET’ print_statement ‘SEMICOLON’ ‘RBRACKET’ ‘}’

Правило 17
----------> ‘PROGRAM’ ‘{’ ‘main’ ‘(’ ‘)’ ‘->’ ‘VOID’ ‘LBRACKET’ ‘PRINT’ ‘LPAREN’ assignable ‘RPAREN’ ‘SEMICOLON’ ‘RBRACKET’ ‘}’

Правило 22
----------> ‘PROGRAM’ ‘{’ ‘main’ ‘(’ ‘)’ ‘->’ ‘VOID’ ‘LBRACKET’ ‘PRINT’ ‘LPAREN’ string_literal ‘RPAREN’ ‘SEMICOLON’ ‘RBRACKET’ ‘}’

Правило 25
----------> ‘PROGRAM’ ‘{’ ‘main’ ‘(’ ‘)’ ‘->’ ‘VOID’ ‘LBRACKET’ ‘PRINT’ ‘LPAREN’ ‘Hello, world!’ ‘RPAREN’ ‘SEMICOLON’ ‘RBRACKET’ ‘}’

Демонстрираното прилагане на правилата има следния смисъл за синтаксиса на програмата. Правило 1 дефинира, че програма на езика STUDENT започва с ключова дума ‘program’ (терминален символ PROGRAM), следвана от лява фигурна скоба (терминален символ LBRACKET), следвана от нетерминален символ program_body и завършва с дясна фигурна скоба (терминален символ RBRACKET). Тъй като в тази програма няма нужда от дефиниране на допълнителни променливи или функции, в

правило 2 нетерминалният символ program_body се замества с друг нетерминален символ main_function (главна функция). От правило 3 се вижда, че главната функция в езика започва с терминален символ ‘main’; следват лява кръгла скоба (терминален символ LPAREN), дясна кръгла скоба (терминален символ RPAREN), стрелка (терминален символ ARROW) и ключова дума ‘void’ (терминален символ VOID). Терминалният символ стрелка ‘->’ в езика STUDENT се използва при дефиниране на функция за задаване на тип на резултата, който връща функцията. За извеждане на символен низ е необходим един прост оператор print_statement (правила 9, 10, 11, 17 и 22). Низът “Hello, world!” се получава от лексическия анализатор като лексема от тип низова константа (STRING_LITERAL).

В резултат програмата на езика STUDENT за извеждане на символен низ “Hello, world!” изглежда по следния начин:

```
program {
  main() -> void {
    print("Hello, world!");
  }
}
```

Пример 2. Да се състави програма на STUDENT, която извиква функция, намираща сумата на две цели числа, подадени като параметри на функцията.

Програмата изглежда по следния начин:

```
program {
  suma(int a, int b) -> int {
      return a + b;
  }
  main() -> void {
    print("Suma = ", @suma(10, 5));
  }
}
```

В този пример следва да се обърне внимание, че извикването на функция в езика STUDENT изисква използване на символ ‘@’ преди името на функцията (правило 15 за нетерминалния символ function_call):

```
(15) function_call          : AT IDENTIFIER LPAREN ( actual_parameters )? RPAREN
```

Контролни въпроси:

1. Дайте определения за нетерминален и терминален символ във формална граматика.

2. Какви символи от граматиката може да съдържа текста на дадена входна програма – само нетерминални, само терминални или комбинация от терминални и нетерминални символи? Обосновете отговора си.

3. Обяснете смисъла на следните правила от граматиката на STUDENT:

а) formal_parameters : type IDENTIFIER (COMMA type IDENTIFIER)*

б) statement : simple_statement SEMICOLON | compound_statement

в) simple_statement : variable_definition | assignment | function_call | return_statement |
print_statement | read_statement

г) compound_statement : if_statement | while_statement

д) print_statement : PRINT LPAREN assignable (COMMA assignable)* RPAREN

е) function_call : AT IDENTIFIER LPAREN (actual_parameters)? RPAREN

Задачи:

1. Да се приложат правилата за пораждане на изречението от пример 2 в текста на упражнението.

2. Да се напише програма на STUDENT, която извежда сумата на целите числа от 1 до 10.

3. Да се напише програма на STUDENT, която извежда стойността на факториел за число, подадено като параметър на функция, като се използва итеративен алгоритъм.

4. Да се напише програма на STUDENT, която извежда стойността на факториел за число, подадено като параметър на функция, като се използва рекурсивен алгоритъм.