Оглавление

[1. Введение 2](#_Toc440728928)

[2. Постановка задачи 2](#_Toc440728929)

[3. Описание исходного языка 2](#_Toc440728930)

[4. Детерминированная автоматная модель синтаксического анализатора 4](#_Toc440728931)

[5. Грамматика свойств 4](#_Toc440728932)

[6. Структура разработанной программы 8](#_Toc440728933)

[7. Результаты тестирования 11](#_Toc440728934)

[8. Руководство пользователя 13](#_Toc440728935)

[9. Заключение 14](#_Toc440728936)

# Введение

Задачей курсового проекта является разработка программы, которая выполняет синтаксически управляемый анализ семантики идентификаторов в текстах программ на языке Pascal.

Программа, осуществляя анализ текста программы на языке Pascal, сообщает о правильности текста программы как синтаксически, так и семантически. В противном случае сообщает о выявленных семантических и синтаксических ошибках в тексте программы.

Язык Pascal – структурированный, компилируемый язык, который один из наиболее известных языков программирования, используется для обучения программированию в старших классах и на первых курсах вузов, является базой для ряда других языков.

# Постановка задачи

Разработать программу синтаксического и семантического анализа для подмножества языка программирования Pascal.

1. Синтаксический разбор реализовать с использованием детерминированного автомата с магазинной памятью.
2. Семантические аспекты описать грамматикой свойств.

Выбранное подмножество языка Pascal содержит следующее:

* Инициализация переменной либо числовых, либо строковых типов данных
* Блок begin-end
* Математические выражения
* Логические выражения
* Условный оператор if-then-else
* Присвоение строк

# Описание исходного языка

Регулярная грамматика, описывающая лексемы языка:

1. G\_VAR -> var
2. G\_STR -> :string
3. G\_INT -> :integer
4. G\_BEGIN -> begin
5. G\_END -> end
6. G\_IF -> if
7. G\_THEN -> then
8. G\_ELSE -> else
9. G\_AND -> and
10. G\_OR -> or
11. G\_IDENT -> [a-zA-Z][0-9a-zA-Z]\*
12. G\_NUMBER -> [0-9]+
13. G\_STRING -> [“].\*[“]
14. G\_ASSIGN -> “:=”
15. G\_NXTL -> “;”
16. G\_ENDL -> “.”
17. G\_PLUS -> “+”
18. G\_MINUS -> “-“
19. G\_MULT -> “\*”
20. G\_DIV ->” \”
21. G\_LPAR -> “(“
22. G\_RPAR -> “)”
23. G\_CMP -> “=”
24. G\_MORE -> “>”
25. G\_LESS -> “<”
26. G\_CMORE -> “>=”
27. G\_CLESS -> “<=”
28. G\_SPACE -> “ “

Контекстно-свободная грамматика, описывающая синтаксис языка:

1. <AXIOM> ::= <L\_DECL> <BLOCK> < G\_ENDL>
2. <L\_DECL> ::= <G\_VAR> <G\_SPACE> <L\_IDENT>
3. <L\_IDENT> ::= <G\_IDENT> <G\_STR> <G\_NXTL> <L\_IDENT>
4. | <G\_IDENT> <G\_INT> <G\_NXTL> <L\_IDENT>
5. | Λ
6. <BLOCK> ::= <G\_BEGIN> <ACTION> <G\_END>
7. <ACTION> ::= <OPER> <ACTION>
8. | Λ
9. <OPER> ::= <BLOCK> <G\_NXTL>
10. | <G\_IDENT> <G\_ASSIGN> <G\_STRING> <G\_NXTL>
11. | <G\_IDENT> <G\_ASSIGN> <E> <G\_NXTL>
12. | <IFRule>
13. <E> ::= <E> <N\_Plus\_Minus> <T>
14. | <T>
15. <T> ::= <T> <N\_Div\_Mult> <F>
16. | <F>
17. <F> ::= <G\_LPAR> <E> <G\_RPAR>
18. | <G\_IDENT>
19. | <G\_NUMBER>
20. <IFRule> ::= <G\_IF> <G\_SPACE> <LOG\_EXPR> <G\_SPACE> <G\_THEN> <N\_SPACE> <OPER> <N\_SPACE> <ELSERule>
21. <ELSERusle> ::= <G\_ELSE> <N\_SPACE> <OPER>
22. | Λ
23. <LOG\_EXPR> ::= <LOG\_EXPR> <SIGN> <T\_LOG>
24. | <T\_LOG>
25. <T\_LOG> ::= <T\_LOG> <LOG\_OP> <F\_LOG>
26. | <F\_LOG>
27. <F\_LOG> ::= <G\_LPAR> <LOG\_EXPR> <G\_RPAR>
28. | <G\_IDENT>
29. | <G\_NUMBER>
30. <LOG\_OP> ::= <G\_AND> | <G\_OR>
31. <SIGN> ::= <G\_CMP> | <G\_MORE> | <G\_LESS>
32. | <G\_CMORE> | <G\_CLESS>
33. <N\_Div\_Mult> ::= <G\_DIV> | <G\_MULT>
34. <N\_ Plus\_Minus> ::= <G\_PLUS> | <G\_MINUS>
35. <N\_SPACE> ::= <G\_SPACE> <N\_SPACE> | Λ

# Детерминированная автоматная модель синтаксического анализатора

Для реализации лексического анализа используется конечный автомат, получаемый утилитой Flex. Flex (Fast Lexical Analyzer) — генератор лексических анализаторов. На входе получает текст в свободном формате и правила выделения лексем. На выходе даёт код анализатора, в виде функции на языке C. Правила задаются в виде регулярных выражений.

Для реализации синтаксического анализа используется детерминированный автомат с магазинной памятью, получаемый утилитой Bison. Bison - программа, предназначенная для автоматического создания синтаксических анализаторов по данному описанию контекстно-свободной грамматики.

Flex используется для описания базовых токенов и создания программы перерабатывающей поток символов в поток токенов. Bison используется для описания грамматики, построенной на базе алфавита токенов, и используется для генерации программы, которая получает на вход поток токенов и находит в этом потоке структурные элементы (нетерминальные токены) согласно заданной грамматике.

# Грамматика свойств

Список свойств идентификаторов:

1. Идентификатор не имеет отношение к данной лексеме

(Нейтральное свойство)

1. Идентификатор объявлен как переменная числовых типов данных

2. Идентификатор объявлен как переменная символьных типов данных

3. Идентификатор используется как переменная числовых типов данных

4. Идентификатор используется как переменная символьных типов данных

5. Обнаружен идентификатор.

Список ошибок, которые будут обнаружены грамматикой свойств:

* Недопустимое использование идентификатора. (переменной типа integer присваевается строка)
* Использование необъявленной переменной

Контекстно-свободная грамматика, дополненная до грамматики свойств:

1. <AXIOM> ::= <L\_DECL> <BLOCK> < G\_ENDL>

|  |  |
| --- | --- |
| *ℓ* | *µ(1, ℓ)* |
| 0 0 0 | 0 |
| 1 0 0 | 0 |
| 1 3 0 | 0 |
| 2 0 0 | 0 |
| 2 4 0 | 0 |

1. <L\_DECL> ::= <G\_VAR> <G\_SPACE> <L\_IDENT>

|  |  |
| --- | --- |
| *ℓ* | *µ(2, ℓ)* |
| 0 0 0 | 0 |
| 0 0 1 | 1 |
| 0 0 2 | 2 |

1. <L\_IDENT> ::= <G\_IDENT> <G\_STR> <G\_NXTL> <L\_IDENT>

|  |  |
| --- | --- |
| *ℓ* | *µ(3, ℓ)* |
| 0 0 0 0 | 0 |
| 5 0 0 0 | 2 |
| 0 0 0 1 | 1 |
| 0 0 0 2 | 2 |

1. <L\_IDENT> ::= <G\_IDENT> <G\_INT> <G\_NXTL> <L\_IDENT>

|  |  |
| --- | --- |
| *ℓ* | *µ(4, ℓ)* |
| 0 0 0 0 | 0 |
| 5 0 0 0 | 1 |
| 0 0 0 1 | 1 |
| 0 0 0 2 | 2 |

1. <L\_IDENT> ::= Λ

|  |  |
| --- | --- |
| *ℓ* | *µ(5, ℓ)* |
| Λ | 0 |

1. <BLOCK> ::= <G\_BEGIN> <ACTION> <G\_END>

|  |  |
| --- | --- |
| *ℓ* | *µ(6, ℓ)* |
| 0 0 0 | 0 |
| 0 3 0 | 3 |
| 0 4 0 | 4 |

1. <ACTION> ::= <OPER> <ACTION>

|  |  |
| --- | --- |
| *ℓ* | *µ(7, ℓ)* |
| 0 0 | 0 |
| 0 3 | 3 |
| 3 3 | 3 |
| 3 0 | 3 |
| 0 4 | 4 |
| 4 4 | 4 |
| 4 0 | 4 |

1. <ACTION> ::= Λ

|  |  |
| --- | --- |
| *ℓ* | *µ(8, ℓ)* |
| Λ | 0 |

1. <OPER> ::= <BLOCK> <G\_NXTL>

|  |  |
| --- | --- |
| *ℓ* | *µ(9, ℓ)* |
| 0 0 | 0 |
| 3 0 | 3 |
| 4 0 | 4 |

1. <OPER> ::= <G\_IDENT> <G\_ASSIGN> <G\_STRING> <G\_NXTL>

|  |  |
| --- | --- |
| *ℓ* | *µ(10, ℓ)* |
| 0 0 0 0 | 0 |
| 5 0 0 0 | 4 |

1. <OPER> ::= <G\_IDENT> <G\_ASSIGN> <E> <G\_NXTL>

|  |  |
| --- | --- |
| *ℓ* | *µ(11, ℓ)* |
| 0 0 0 0 | 0 |
| 5 0 0 0 | 3 |
| 0 0 0 3 | 3 |

1. <OPER> ::= <IFRule>

|  |  |
| --- | --- |
| *ℓ* | *µ(12, ℓ)* |
| 0 | 0 |
| 3 | 3 |
| 4 | 4 |

1. <E> ::= <E> <N\_Plus\_Minus> <T>

|  |  |
| --- | --- |
| *ℓ* | *µ(13, ℓ)* |
| 0 0 0 | 0 |
| 0 0 3 | 3 |
| 3 0 0 | 3 |
| 3 0 3 | 3 |

1. <E> ::= <T>

|  |  |
| --- | --- |
| *ℓ* | *µ(14, ℓ)* |
| 0 | 0 |
| 3 | 3 |

1. <T> ::= <T> <N\_Div\_Mult> <F>

|  |  |
| --- | --- |
| *ℓ* | *µ(15, ℓ)* |
| 0 0 0 | 0 |
| 0 0 3 | 3 |
| 3 0 0 | 3 |
| 3 0 3 | 3 |

1. <T> ::= <F>

|  |  |
| --- | --- |
| *ℓ* | *µ(16, ℓ)* |
| 0 | 0 |
| 3 | 3 |

1. <F> ::= <G\_LPAR> <E> <G\_RPAR>

|  |  |
| --- | --- |
| *ℓ* | *µ(17, ℓ)* |
| 0 0 0 | 0 |
| 0 3 0 | 3 |

1. <F> ::= <G\_IDENT>

|  |  |
| --- | --- |
| *ℓ* | *µ(18, ℓ)* |
| 0 | 0 |
| 5 | 3 |

1. <F> ::= <G\_NUMBER>

|  |  |
| --- | --- |
| *ℓ* | *µ(19, ℓ)* |
| 0 | 0 |

1. <IFRule> ::= <G\_IF> <G\_SPACE> <LOG\_EXPR> <G\_SPACE> <G\_THEN> <N\_SPACE> <OPER> <N\_SPACE> <ELSERule>

|  |  |
| --- | --- |
| *ℓ* | *µ(20, ℓ)* |
| 0 0 0 0 0 0 0 0 0 | 0 |
| 0 0 3 0 0 0 3 0 0 | 3 |
| 0 0 3 0 0 0 0 0 3 | 3 |
| 0 0 3 0 0 0 3 0 3 | 3 |
| 0 0 0 0 0 0 0 0 3 | 3 |
| 0 0 0 0 0 0 3 0 0 | 3 |
| 0 0 0 0 0 0 3 0 3 | 3 |
| 0 0 0 0 0 0 4 0 0 | 4 |
| 0 0 0 0 0 0 0 0 4 | 4 |
| 0 0 0 0 0 0 4 0 4 | 4 |

1. <ELSERusle> ::= <G\_ELSE> <N\_SPACE> <OPER>

|  |  |
| --- | --- |
| *ℓ* | *µ(21, ℓ)* |
| 0 0 0 | 0 |
| 0 0 3 | 3 |
| 0 0 4 | 4 |

1. <ELSERusle> ::= Λ

|  |  |
| --- | --- |
| *ℓ* | *µ(22, ℓ)* |
| Λ | 0 |

1. <LOG\_EXPR> ::= <LOG\_EXPR> <SIGN> <T\_LOG>

|  |  |
| --- | --- |
| *ℓ* | *µ(23, ℓ)* |
| 0 0 0 | 0 |
| 0 0 3 | 3 |
| 3 0 0 | 3 |
| 3 0 3 | 3 |

1. <LOG\_EXPR> ::= <T\_LOG>

|  |  |
| --- | --- |
| *ℓ* | *µ(24, ℓ)* |
| 0 | 0 |
| 3 | 3 |

1. <T\_LOG> ::= <T\_LOG> <LOG\_OP> <F\_LOG>

|  |  |
| --- | --- |
| *ℓ* | *µ(25, ℓ)* |
| 0 0 0 | 0 |
| 0 0 3 | 3 |
| 3 0 0 | 3 |
| 3 0 3 | 3 |

1. <T\_LOG> ::= <F\_LOG>

|  |  |
| --- | --- |
| *ℓ* | *µ(26, ℓ)* |
| 0 | 0 |
| 3 | 3 |

1. <F\_LOG> ::= <G\_LPAR> <LOG\_EXPR> <G\_RPAR>

|  |  |
| --- | --- |
| *ℓ* | *µ(27, ℓ)* |
| 0 0 0 | 0 |
| 0 3 0 | 3 |

1. F\_LOG> ::= <G\_IDENT>

|  |  |
| --- | --- |
| *ℓ* | *µ(18, ℓ)* |
| 0 | 0 |
| 5 | 3 |

1. F\_LOG> ::= <G\_NUMBER>

|  |  |
| --- | --- |
| *ℓ* | *µ(29, ℓ)* |
| 0 | 0 |

# Структура разработанной программы

Программа имеет 4 основных блока, взаимодействующих между собой:

* Сгенерированый утилитой Flex код функции лексического анализа
* Сгенерированый утилитой Bison код функции синтаксического анализа
* Блок таблиц свойств
* Блок синтезируемых атрибутов

***Bison и Flex***

Программа использует сгенерированные коды функций утилит Flex и Bison для синтаксического и семантического анализа тескта программы на языке Pascal. Как выше упоминалось, Flex используется для описания базовых токенов и создания программы перерабатывающей поток символов в поток токенов. Bison используется для описания грамматики, построенной на базе алфавита токенов, и используется для генерации программы, которая получает на вход поток токенов и находит в этом потоке нетерминальные токены согласно заданной грамматике. Это используется в основном для синтаксического анализа.

Результат утилиты Flex: функция yylex() *-* распознаёт лексемы во входном потоке и передаёт их анализатору Bison.

Результат утилиты Bison: функция yyparse() - читает лексемы, выполняет действия анализатора, и в конце концов завершает работу, когда встречает конец входного текста или сталкивается с синтаксической ошибкой.

***Блок таблиц свойств***

Что касается семантического анализа, программа работает с таблицами грамматики свойств, хранящиеся в файле *propety\_table.db*. Перед началом работы выполняется чтение таблиц из файла функцией:

int get\_prop\_table(char\* filename, prop\_table\*\* p\_table, int \*count), где

char\* filename – имя файла, хранящее таблицы грамматики свойств

prop\_table\*\* p\_table – указатель на будущий массив таблиц грамматики свойств

int \*count – количество таблиц грамматики свойств.

В случае отсутсвия нужного файла, программа выведет соотвествующее сообщение и завершит свою работу.

Структура массива таблиц грамматики свойств:

typedef struct

{

int row; // кол-во строк в таблице

int col; // кол-во столбцов

int \*\*l; // строка свойств

int \*mu; // значение функции свойств mu(l)

} prop\_table;

Для подсчета синтезируемого атрибута для символа левой части правила используется функция:

int calculate\_mu(int\* l, prop\_table\* table, int index), где

int\* l – строка свойств

prop\_table\* table – массив таблиц грамматики свойств

int index – индекс таблицы грамматики свойств

Расчет выполняется поиском строки свойств *l* в таблице под индексом *index*

и если поиск удался, то возвращается свойство, которое сохраняется для текущего идентификатора. Если же поиск не удался, то возвращается признак семантической ошибки, который далее будет обрабатывться программой.

Функции выполняющие обработку массива таблиц грамматики свойств представлены в заголовочном файле *Property\_table.h.*

***Синтезируемые атрибуты***

Структура синтезируемых атрибутов левых и правых частей правил имеет слудующий вид:

typedef struct Table

{

char\* ident; // имя идентификатора

int prop; // текущее свойство индентификатора

struct Table\* next; // указатель на следующий элемент списка

} table;

Синтезируемый атрибут – однонаправленный список, элементы которого содержат сведения о идентификаторе (его имя и текущее свойств).

Семантический анализ производиться, функцией sem\_analyze. На вход пуступает пустая строка свойств, массив таблиц грамматики свойств, номер правила и два списка атрибутов. Для каждого элементы списка производится подсчет свойств функцией calculate\_mu(), причем рассматривается список list1. Если он пустой, то в строку свойств заноситься нуль, иначе берется первый элемент списка, в строку свойств заносится текущее значение свойства идентификатора и производиться поиск этого элемента в списке list2. Если list2 пустой, то в строку свойств заноситься нуль, иначе производиться поиск текущего выбранного элемента. Если элемент найден в list2, то в строку свойств заносится текущее значение свойства идентификатора из списка list2. Сформированная строка свойств отправляется как аргумент в функцию calculate\_mu() и создается(дополняется) список синтезируемых атрибутов. (Если list1 пустой, то рассматривается в качестве текущего индентификатора, для которого ведется подсчет, первый элемент list2, причем в первую позицию строки свойств заноситься нуль).

table\* sem\_analyze(int\* l, table \*list1, table \*list2, prop\_table \*p\_table, int rule\_number)

int\* l – строка свойств

table \*list1 – первый список атрибутов

table \*list2 – второй список атрибутов

prop\_table \*p\_table – массив таблиц грамматики свойств

int rule\_number – индекс таблицы грамматики свойств (= номер правила)

Функция выполняющая поиск идентификатора в списке, осуществляет последовательное сравнение имени индентификатора с именем текущего элемента списка.

Если поиск удался, то функция возвращает указатель на элемент, иначе возвращает нуль.

table\* find\_in\_table(char\* ident, table\* phead)

char\* ident – имя идентификатора

table\* phead – список атрибутов

Функции выполняющие обработку синтезируемых представлены в заголовочном файле T*able.h.*

После завершения работы программы производится освобождение памяти под динамические выделеные структуры функциями:

int free\_table(table\*\* phead) – функция освобождающая память, выделеную под список атрибутов. Вовращает нуль, если все прошло успешно, 1 – если прошло все безуспешно.

int free\_prop\_table(prop\_table\*\* table, int count) – функция освобождающая память, выделеную под массив таблиц грамматики свойств. Вовращает нуль, если все прошло успешно, 1 – если прошло все безуспешно.

# Результаты тестирования

*Тестирование №1*:

Входной текст:

var

i:integer;

s:string;

sr:string;

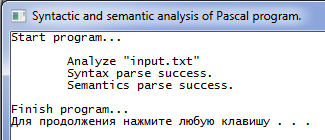
begin

if i>0 then i:=5; else i:=4;

s:="this is string";

end.

Результат выполнения программы:



*Тестирование №2:*

Входной текст:

var

i:integer;

s:string;

begin

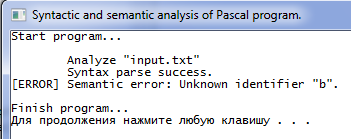
if i>0 then i:=5; else i:=4;

s:="this is string";

i:=i\***b**\(122-67\*i)+**b**;

end.

Результат выполнения программы:



*Тестирование №3:*

Входной текст:

var

i:integer;

b:integer;

s:string;

begin

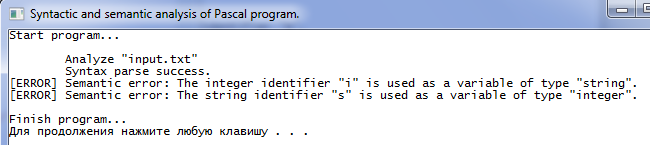
if i>0 then **s**:=5; else i:=4;

**i**:="this is string";

i:=i\*b\(122-67\*i)+b;

end.

Результат выполнения программы:



*Тестирование №4:*

Входной текст:

var

i:integer;

b:integer;

begin

if i>10 then

if b>i then

b:=12\*i;

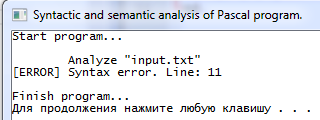
else

b:=7\*b;

i:=i\*b\(122-67\*i)+b;

ed.

Результат выполнения программы:



# Руководство пользователя

Перед началом работы убедитесь, что в директории с программой находятся файлы с таблицами грамматики свойств *property\_table.db* и с текстом программы на языке Pascal *input.txt*. В случае их отсутсвия программа работать не будет.

* Входной текст программы на языке Pascal должен находиться в файле *input.txt.*
* Для корректной работы программы входной текст должен быть структурирован, а именно исходный текст не должен быть написан в одну строчку.
* Блок begin-end должен начинаться с новой строки.
* Помните, синтаксический анализатор регистрозависимый, поэтому проверяйте используемые имена идентификаторов.
* Во избежание конфилктов сдвига/приведение в синтаксическом анализаторе (в функции yyparse()), описывая логическое выражение, не ставьте пробелы. Для разграничений используйте скобки.

Пример правильного использования блока begin-end:

var

i:integer;

b:integer;

begin

if i>10 then

begin

if b>i then

b:=12\*i;

else

b:=7\*b;

end;

i:=i\*b\(122-67\*i)+b;

end.

Пример правильного описания логических выражений в условном операторе:

var

i:integer;

b:integer;

s:string;

begin

if (((i>i)or(i=f))and(i<324)) then s:="this is string";

begin

i:=(123+i\67)\(54\4\*b-627\*(12-i\b));

end;

end.

Пример неверного использования блока begin-end:

var

i:integer;

b:integer;

begin

if i>10 then

**begin** i:=b; **end**;

end.

Пример неверного описания логических выражений в условном операторе:

var

i:integer;

b:integer;

begin

if (i>10 **and** i<40) then

begin

i:=b;

end;

end.

# Заключение

В данной курсовой работе была разработана программа, которая выполняет синтаксически управляемый анализ семантики идентификаторов в текстах программ на языке Pascal. Синтаксический разбор организован с использованием детерминированного автомата с магазинной памятью. Семантический разбор - грамматикой свойств. Программа анализирует тексты программы на языке Pascal на предмет семантических и синтаксических ошибок. Обнаружив синтаксическую ошибку, программа указывает проблемную строку, а при семантической ошибке указывает идентификатор, с которы возникли те или инные проблемы. Для работы использовались функции синтаксического и лексического анализа генерируемые утилитами Bison и Flex соотвественно.

Программа выполняет поставленную задачу и работает без ошибок (для корректных тестовых данных).