

Отчёт по программе синтаксического анализа операторов цикла `while (. . .) . . .` `done`

Введение

Цель данной программы — разработать лексический и синтаксический анализаторы для языка, содержащего операторы цикла вида `while (. . .) . . . done`, разделённые символом `;`. Программа должна принимать текст на этом языке, выполнять лексический анализ (разбиение на токены), синтаксический анализ (проверку структуры) и строить дерево разбора (AST). Для реализации выбран язык программирования C++ с использованием современных стандартов и средств автоматизации сборки (Makefile).

Дерево разбора (AST):

```
1 Program
2   StatementList
3     WhileLoop
4       Condition
5         Identifier (x)
6         RelOp (<)
7         RomanNumeral (V)
8       Assignment
```

```

9      LValue (y)
10     RomanNumeral (I)

```

Основные задачи разработки

1. Разработка лексического анализатора, распознающего ключевые слова (`while`, `done`), идентификаторы, римские числа (записанные заглавными `I`, `V`, `X`), операторы сравнения (`<`, `>`, `=`), присваивания (`:=`) и разделителя (`;`).

Таблица LR-анализа

State	<code>while</code>	<code>done</code>	()	;	<code>IDENTIFIER</code>	<code>RC</code>
0	s2					s3	
1							
2			s4				
3							
4						s8	s9
5						s8	s9
6							
7							
8							
9							
10				r7			
11							
12							

State	while	done	()	;	IDENTIFIER	RC
13							
14				s15			
15						s17	
16		s18					
17							
18	r4				s20		
19						s8	s9
20	s2					s3	
21							
22							

Обозначения в заголовках таблицы

Терминалы (токены)

Токен	Описание
while , done	Ключевые слова языка
(,) , ;	Символы-разделители
IDENTIFIER	Идентификаторы (имена переменных)
ROMAN_NUMERAL	Римские числа (I, V, X)
:=	Оператор присваивания
< , > , =	Операторы сравнения

Нетерминалы (правила грамматики)

Нетерминал	Описание
Program	Корневое правило программы
StatementList	Список операторов
Statement	Одиночный оператор
Condition	Условие цикла
Body	Тело цикла
Assignment	Оператор присваивания
Expression	Выражение
RelOp	Оператор отношения

Цифровые обозначения

В действиях (sN , rN , gN)

Обозначение	Значение
N в sN	Номер состояния для перехода
N в rN	Номер правила грамматики для свёртки
N в gN	Номер состояния для перехода после свёртки

Состояния автомата

Диапазон	Назначение
0	Начальное состояние
1-22	Внутренние состояния LR-автомата

Специальные случаи

Обозначение	Ситуация	Реакция
Пустая ячейка	Ошибка синтаксиса	Анализатор останавливается с сообщением об ошибке
rN в столбце \$	Успешный разбор	Возможность завершения анализа
acc в столбце \$	Полное принятие	Строка полностью соответствует грамматике

Алгоритмы вычисления

Вычисление FIRST(X)

```
1  Если X – терминал:  
2      FIRST (X) = {X}  
3  
4  Если X – нетерминал:  
5      Для каждого правила X → Y1Y2...Yn:  
6          Добавить FIRST(Y1) в FIRST(X)  
7          Если ε ∈ FIRST(Y1), добавить FIRST(Y2)  
8          Если ε ∈ FIRST(Y2), добавить FIRST(Y3)  
9  
10  Если X → ε:  
11      Добавить ε в FIRST(X)
```

Вычисление FOLLOW(A)

```
1  1. Добавить $ в FOLLOW(S)
```

- 2
- 3 2. Для каждого правила $B \rightarrow \alpha A \beta$:
- 4 – Добавить $\text{FIRST}(\beta) - \{\epsilon\}$ в $\text{FOLLOW}(A)$
- 5 – Если $\epsilon \in \text{FIRST}(\beta)$, добавить $\text{FOLLOW}(B)$ в
 $\text{FOLLOW}(A)$
- 6
- 7 3. Для каждого правила $B \rightarrow \alpha A$:
- 8 – Добавить $\text{FOLLOW}(B)$ в $\text{FOLLOW}(A)$

2. Реализация синтаксического анализатора на основе грамматики:

```
1 Program      → StatementList
2 StatementList → Statement (';' Statement)*
3 Statement     → while '(' Condition ')' Body
                  done
4 Condition    → Expression RelOp Expression
5 Body         → Assignment
6 Assignment   → IDENTIFIER ':=' Expression
7 Expression   → IDENTIFIER | ROMAN_NUMERAL
8 RelOp        → '<' | '>' | '='
```

3. Построение и вывод дерева абстрактного синтаксиса (AST) для визуальной проверки корректности разбора.
4. Создание Makefile, обеспечивающего автоматическую сборку и запуск программы с пересборкой при каждом вызове.

Алгоритм решения

Программа выполняет два основных этапа:

- **Лексический анализ:** входная строка сканируется посимвольно. Распознаются ключевые слова, символы и лексемы. Слова из букв `I`, `V`, `X` интерпретируются как римские числа, остальные — как идентификаторы.
- **Синтаксический анализ:** на основе потока токенов рекурсивным спуском строится AST. Каждый узел дерева соответствует конструкции языка (цикл, условие, присваивание и т.д.).

Функция `tokenize` возвращает вектор токенов. Класс `Parser` строит AST. Функция `printAST` выводит дерево с отступами.

Makefile

Для упрощения сборки и запуска используется следующий Makefile:

Makefile

```

1 TARGET = main.exe
2 CXX = g++
3 CXXFLAGS = -Wall -std=c++17
4 SRC = main.cpp
5
6 all: clean $(TARGET)
7     ./$(TARGET)
8
9 $(TARGET) : $(SRC)
10    $(CXX) $(CXXFLAGS) $(SRC) -o $(TARGET)
11

```

```
12 clean:  
13     rm -f $ (TARGET)
```

Использование:

1. Команда `make` выполняет полную пересборку и запускает программу.
2. Цель `clean` удаляет старый исполняемый файл, гарантируя актуальную сборку.

Это обеспечивает удобство тестирования и отладки.

Тестирование и применение

Программа протестирована на следующих входных данных:



Plain text

```
1 while (x < V) y := I done  
2 while (a = I) b := x done; while (n > III) m  
:= a done
```

Вывод программы:

```
● a@DESKTOP-5VD9R0T:/mnt/c/Users/A/CODESPACE/LABS/Compilation-methods_7-semester/lab2$ make
rm -f main.exe
g++ -Wall -std=c++17 main.cpp -o main.exe
./main.exe
==== Тест 1 ====
Program
StatementList
  WhileLoop
    Condition
      Identifier (x)
      RelOp (<)
      RomanNumeral (V)
    Assignment
      LValue (y)
      RomanNumeral (I)

==== Тест 2 ====
Program
StatementList
  WhileLoop
    Condition
      Identifier (a)
      RelOp (=)
      RomanNumeral (I)
    Assignment
      LValue (b)
      RomanNumeral (X)
  WhileLoop
    Condition
      Identifier (n)
      RelOp (>)
      RomanNumeral (III)
    Assignment
      LValue (m)
      Identifier (a)
```

Также проверена реакция на ошибки (пропущенное `done`, неверный оператор и т.д.) — программа корректно завершается с диагностикой.

Заключение

Реализованная программа успешно решает задачу синтаксического анализа заданного языка. Архитектура «лексер → парсер → AST» соответствует классическим принципам построения компиляторов и может быть расширена для поддержки более сложных конструкций (блоков, выражений, семантического анализа).

Использование Makefile автоматизирует процесс сборки и

тестирования, а читаемый код с документацией обеспечивает поддерживаемость. Проект демонстрирует понимание основ лексического и синтаксического анализа и может служить основой для дальнейшего изучения теории компиляторов.