Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
Пермский национальный исследовательский политехнический университет
(ПНИПУ)

Факультет: Электротехнический (ЭТФ)
Направление: 09.03.04 – Программная инженерия (ПИ)
Профиль: Разработка программно-информационных систем (РИС)
Кафедра информационных технологий и автоматизированных систем (ИТАС)

УТВЕРЖДАЮ
Зав. кафедрой ИТАС: д-р экон. наук, проф.
_____ Р.А. Файзрахманов
«_____»_____ 2025 г.

КУРСОВАЯ РАБОТА
по дисциплине
«Системное программирование»
на тему
«Разработка компилятора для языка программирование GamePLang»

Студент: _____ 19.12.25 _____ Камалетдинов Максим Валерьевич
(подпись, дата)
Группа: РИС-23-2б

Дата защиты 19.12.25
Оценка отл

Руководитель КР: _____ стар. преп. каф. ИТАС Кузнецов Д.Б.
(подпись, дата)

Пермь 2025

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
Пермский национальный исследовательский политехнический университет
(ПНИПУ)

Факультет: Электротехнический (ЭТФ)
Направление: 09.03.04 – Программная инженерия (ПИ)
Профиль: Разработка программно-информационных систем (РИС)
Кафедра информационных технологий и автоматизированных систем (ИТАС)

УТВЕРЖДАЮ
Зав. кафедрой ИТАС: д-р экон. наук, проф.
_____ Р.А. Файзрахманов
«_____»_____ 2025 г.

ЗАДАНИЕ
на выполнение курсовой работы

Фамилия, имя, отчество: Камалетдинов Максим Валерьевич
Факультет Электротехнический Группа: РИС-23-2б
Начало выполнения работы: 02.11.2025
Контрольные сроки просмотра работы: 7.11, 18.11, 30.11, 05.12
Защита работы: 19.12.2025
1. Наименование темы: «Разработка компилятора для языка программирование GamePLang».
2. Исходные данные к работе (проекта):
Объект исследования – Компилятор.
Предмет исследования – Алгоритм работы языка программирования GamePLang.
Цель работы (проекта) – Разработать компилятор для языка программирования GamePLang.
3. Содержание:
3.1 Исследование предметной области курсовой работы
3.1.1 Этапы анализа исходного кода программы
3.1.2 Лексический анализ исходного кода
3.1.3 Синтаксический анализ
3.1.4 Семантический анализ
3.1.5 Генерация кода
3.2 Разработка компилятора для собственного языка программирования
3.2.1 Разработка собственного языка программирования
3.2.2 Выполнение трансляции собственного языка программирования под целевую платформу
3.2.3 Разработка лексического анализатора
3.2.4 Разработка синтаксического анализатора
3.2.5 Компиляция компилятора
3.2.6 Создание Makefile и скрипта для автоматической сборки и проверки компилятора
3.2.7 Контрольный пример

Руководитель КР: _____ стар.преп. каф. ИТАС Кузнецов Д.Б.
(подпись, дата)

Задание получил: 02.11.25 М.В. Камалетдинов
(подпись, дата)

КАЛЕНДАРНЫЙ ГРАФИК ВЫПОЛНЕНИЯ
КУРСОВОЙ РАБОТЫ

№ пп	Этапы работы	Объём этапа, %	Сроки выполнения		Примечания
			Начало	Конец	
1.	Исследование предметной области	10	02.11.25	06.11.25	
2.	Анализ исходных данных задания	5	06.11.25	10.11.25	
3.	Разработка собственного языка программирования	10	10.11.25	14.11.25	
4.	Выполнение трансляции собственного языка программирования под целевую платформу	5	14.11.25	17.11.25	
5.	Разработка лексического анализатора	15	17.11.25	24.11.25	
6.	Разработка синтаксического анализатора	15	24.11.25	01.12.25	
7.	Компиляция компилятора	15	01.12.25	02.12.25	
8.	Создание Makefile и скрипта для автоматической сборки и проверки компилятора	10	02.12.25	03.12.25	
9.	Выполнение контрольного примера	5	03.12.25	04.12.25	
10.	Оформление курсовой работы	10	04.12.25	18.12.25	
11.	Защита курсовой работы		19.12.25		

Руководитель КР: _____ стар.преп. каф. ИТАС Кузнецов Д.Б.
(подпись, дата)

Задание получил: 02.11.25 Камалетдинов Максим Валерьевич
(подпись, дата)

РЕФЕРАТ

Отчет 60 с., 12 рис., 5 источн,. 2 прил.

Цель работы - разработать компилятор для языка программирования GamePLang, обеспечивающий трансляцию программ с разработанного языка на языке целевой платформы, посредством компиляции языка программирования GamePLang.

Объектом исследования является собственный язык программирования GamePLang.

Предмет исследования – компилятор языка программирования GamePLang, реализующий лексический и синтаксический анализаторы.

При разработке компилятора применялись утилиты GNU flex и GNU bison, позволяющие описать лексическую и синтаксическую структуру языка.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Создание специализированных языков программирования позволяет адаптировать синтаксис и структуру кода под конкретную предметную область, повышая его наглядность и удобство. В данной работе представлена разработка компилятора для языка GamePLang, где программа моделируется как спортивная тактическая схема.

Целью работы является разработка компилятора, транслирующего код на языке GamePLang в исполняемый код на языке C с последующей компиляцией в машинный код, построить лексический и синтаксический анализаторы, провести семантический анализ с последующей генерацией кода.

Актуальность работы заключается в том, что она позволяет на практике изучить создание специализированных языков программирования. Разрабатываемый язык GamePLang использует спортивную терминологию, что делает его примером подхода, когда код пишется не просто командами, а как понятная инструкция или тактический план. Создание компилятора для такого нестандартного языка помогает глубже разобраться в том, как работают инструменты вроде Flex и Bison, и как можно превращать удобные для человека команды в строгий компьютерный код.

1 Исследование предметной области курсовой работы

1.1 Этапы анализа исходного кода программы

Архитектура компилятора языка GamePLang построена по классической схеме и включает несколько этапов преобразования исходного кода. Данный конвейер обработки необходим для корректной трансформации пользовательских скриптов в исполняемый вид или для трансляции в целевой язык Си.

Начальной стадией является лексический анализ. На этом этапе происходит считывание потока символов входного файла и группирование их в значащие единицы — токены. На этом уровне система распознает категории лексем: зарезервированные (ключевые) слова языка, числовые константы, строковые литералы и операторы. Результатом является очищенная от пробелов и комментариев последовательность токенов.

Вторым этапом выступает синтаксический анализ. Парсер принимает поток токенов и проверяет их порядок на соответствие формальной грамматике GamePLang. Основная цель этого этапа — это построение иерархической структуры программы, в языке GamePLang в виде Абстрактного Синтаксического Дерева (AST). Именно здесь выявляются вложенности циклов, структуры условных переходов и составы выражений.

Завершающей фазой анализа является семантическая проверка. Поскольку GamePLang является статически типизированным языком с неявной типизацией, на этом этапе контролируется согласованность типов данных и корректность объявления переменных. Успешное прохождение этого этапа гарантирует, что построенное дерево готово для финальной кодогенерации на языке С.»

1.2 Лексический анализ исходного кода

Лексический анализ представляет собой начальный этап обработки программы на языке GamePLang. Его основная задача — последовательное

чтение исходного текста и группировка символов в смысловые единицы — токены. Для автоматизации этого процесса используется утилита Flex, которая на основе заданных регулярных выражений генерирует эффективный сканер на языке C.

В ходе анализа выделяются различные категории лексем, специфичные для GamePLang. Особое внимание уделяется идентификаторам переменных, которые в данном языке обязательно начинаются с символа «#» (например, #score), а также словесным операторам (таким как SCORE, ADD), заменяющим традиционные математические знаки. Также распознаются ключевые слова, определяющие структуру программы (FORMATION, DRILL), и числовые литералы.

Параллельно с распознаванием токенов выполняется предварительная очистка кода: анализатор игнорирует пробельные символы и однострочные комментарии, не передавая их на следующие этапы. Полученная цепочка токенов служит входными данными для синтаксического анализатора, обеспечивая проверку грамматической структуры программы.

1.3 Синтаксический анализ

Синтаксический анализ (парсинг) представляет собой второй этап трансляции, в ходе которого линейная последовательность токенов, полученная от лексического анализатора, преобразуется в иерархическую структуру — Абстрактное Синтаксическое Дерево (AST). Для автоматизации этого процесса в проекте используется генератор синтаксических анализаторов GNU Bison, работающий на основе контекстно-свободных грамматик и алгоритма восходящего разбора LALR.

Bison обрабатывает файл спецификации, содержащий описание грамматики языка GamePLang в нотации, близкой к форме Бэкуса-Наура (BNF). В разделе определений с помощью директивы %union задаются типы данных, которые могут быть связаны с нетерминалами: строковые значения

для идентификаторов и указатели на узлы дерева (struct ASTNode*) для синтаксических конструкций. Здесь же объявляются токены и определяются приоритеты операций (директивы %left) для корректного разрешения математических выражений.

Правила грамматики описывают структуру программы как набор вложенных блоков. Корневым символом является program, определяющий общую схему программы (FORMATION). Грамматика жестко задает порядок секций:

1) Секция PLAYERS: отвечает за декларацию переменных (игроков) и их инициализацию. Парсер проверяет корректность типов данных при создании массивов и скалярных переменных;
2) Секция SEQUENCE: содержит главную последовательность инструкций (аналог функции main), которая управляет ходом выполнения программы;
3) Секция функций (PLAY): описывает подпрограммы, которые могут быть вызваны из основного потока.

Каждому правилу грамматики сопоставлено семантическое действие на языке C. При успешном распознавании конструкции вызываются специальные функции-конструкторы (например, new_node, new_arr_set), которые создают узлы синтаксического дерева. Например, при разборе цикла DRILL создается узел типа NODE_LOOP, хранящий информацию о счетчике, условии завершения и теле цикла.

Особое внимание в синтаксическом анализаторе уделено управляющим конструкциям. Условный оператор WHEN, THEN, OTHERWISE обрабатывается как единый узел ветвления. Арифметические и логические выражения, использующие словесные операторы (ADD, SCORE, ABOVE), преобразуются в бинарные узлы дерева с учетом ассоциативности.

Результатом работы синтаксического анализатора является полностью сформированное дерево AST, которое точно отражает логику исходной программы и не содержит синтаксических ошибок. Это дерево передается на следующий этап для генерации целевого кода на языке C.

1.4 Семантический анализ

Семантический анализ является третьим этапом работы компилятора GamePLang, обеспечивающим проверку смысловой корректности программы. Если синтаксический анализатор гарантирует правильность структуры предложений языка, то семантический анализатор проверяет, имеют ли эти предложения смысл в контексте статической типизации.

В данной реализации семантический анализ выполняется интегрированно с синтаксическим разбором (метод синтаксически-управляемой трансляции). Центральным элементом этого этапа является таблица символов (Symbol Table). При обработке секции PLAYERS компилятор заносит в таблицу имена всех объявленных переменных и их выведенные типы (INT, STRING, ARRAY). Это позволяет реализовать контроль области видимости: при любом обращении к переменной в основном коде проверяется факт её предварительного объявления.

Ключевой задачей семантического анализатора GamePLang является контроль типов данных. Особое внимание уделяется инициализации массивов: специальная процедура верификации гарантирует однородность данных, запрещая создание структур, содержащих одновременно и числа, и строки (например, [10, "error"]). В случае обнаружения несовместимости типов или использования необъявленного идентификатора, анализатор прерывает компиляцию с выдачей сообщения об ошибке, предотвращая генерацию некорректного C-кода.

1.5 Генерация код

Генерация кода является заключительным этапом работы компилятора GamePLang. На этой стадии происходит преобразование верифицированного Абстрактного Синтаксического Дерева (AST) в эквивалентную программу на целевом языке, в качестве которого был выбран язык C (Си). Это позволяет обеспечить переносимость итогового приложения и использовать оптимизации компилятора GCC.

Процесс генерации реализован через рекурсивный обход дерева функцией codegen. При посещении узлов AST компилятор формирует соответствующие конструкции языка C:

1) Трансляция переменных: Идентификаторы GamePLang, начинающиеся со спецсимвола #, преобразуются в безопасные имена C с префиксом gp_. Числовые переменные отображаются в тип int, а строковые — в указатели char*;
2) Управляющие конструкции: Спортивные циклы DRILL и QUARTER разворачиваются в стандартные циклы for. Условный оператор WHEN транслируется в конструкцию if-else;
3) Операции: Арифметические выражения конвертируются напрямую. Для строковых операций (например, конкатенации через +) генерируются вызовы вспомогательных функций, так как C не поддерживает перегрузку операторов;
4) Ввод-вывод: Команды HUDDLE и RECRUIT преобразуются в вызовы библиотечных функций printf и scanf с соответствующими спецификаторами формата (%d для чисел).

Результатом работы модуля является готовый файл исходного кода, содержащий необходимые заголовочные файлы, объявления переменных и функцию main, полностью повторяющую логику исходного скрипта.

2 Разработка компилятора для собственного языка программирования

2.1 Разработка собственного языка программирования

Разработанный язык программирования GamePLang представляет собой компилируемый язык со строгой структурой. Его главная особенность заключается в том, что исходный код сначала транслируется в код на языке Си (C), после чего превращается в исполняемую программу. Такой подход позволяет объединить простоту написания кода с высокой скоростью работы готовых программ.

Язык спроектирован так, чтобы программист четко разделял данные (переменные) и логику (команды). Синтаксис языка использует английские ключевые слова в верхнем регистре, что делает код читаемым и понятным.

Лексические и синтаксические особенности

В отличие от многих популярных языков (C++, Java), в GamePLang не используются фигурные скобки для обозначения блоков кода. Вместо этого применяются явные ключевые слова для начала и конца каждой конструкции (например, начало цикла и конец цикла).

Основные правила написания кода:

1) Переменные: Для быстрого распознавания имен переменных в тексте программы используется специальный символ решетки # перед названием (например, #count, #result);
2) Команды: Все зарезервированные слова пишутся заглавными буквами;
3) Завершение строк: Каждая инструкция должна заканчиваться точкой с запятой (;).

Структура программы

Любая программа на GamePLang имеет жесткую структуру, состоящую из главного блока FORMATION, который делится на три обязательные секции:

1) PLAYERS (Секция данных): Здесь происходит объявление всех переменных, которые будут использоваться в программе. Это позволяет заранее выделить память под числа и массивы;
2) SEQUENCE (Секция логики): Это главная часть программы, которая начинает выполняться при запуске. Здесь описывается последовательность действий;
3) PLAY (Секция подпрограмм): Здесь описываются вспомогательные алгоритмы (функции), которые можно многократно вызывать из главной секции.

Типы данных и переменные

Язык использует статическую типизацию. Это значит, что тип переменной (число или строка) определяется один раз при её создании и не может меняться в процессе работы.

Числа: Целые числа (тип INT).

Строки: Текстовые данные (тип STRING).

Массивы: Списки элементов фиксированного размера. Для создания массива используется специальный синтаксис: #имя = [размер, значение_по_умолчанию].

Операции и выражения

Для выполнения математических действий язык поддерживает как символы, так и словесные команды, что снижает вероятность ошибки при чтении кода:

ADD — сложение (для строк работает как склеивание текста);

SUBTRACT — вычитание;

MULTIPLY — умножение;

DIVIDE — деление нацело.

Также реализованы специальные команды для быстрого изменения значений:

SCORE — увеличить значение переменной на указанное число.

RECRUIT — считать данные с клавиатуры и записать их в переменную. При этом выполняется проверка: если ожидается число, а введен текст, программа сообщит об ошибке.

Управление ходом программы

Для создания ветвлений (выбора действий в зависимости от условия) используется конструкция WHEN ... THEN ... OTHERWISE. Она проверяет истинность выражения и выполняет либо основной блок команд, либо альтернативный.

Для организации повторяющихся действий (циклов) предусмотрено несколько конструкций:

DRILL: Базовый цикл, который выполняет блок команд заданное количество раз. Внутри цикла доступна переменная-счетчик.

EXERCISE: Расширенная конструкция для создания вложенных циклов. Она удобна при работе со сложными данными, например, двумерными массивами.

Подпрограммы

Чтобы не дублировать код, повторяющиеся участки логики можно вынести в отдельные функции (PLAY). Вызов этих функций осуществляется командой EXECUTE. Также поддерживается передача параметров внутрь функции с помощью конструкции CALL ... WITH, что делает подпрограммы более гибкими.

Таким образом, GamePLang представляет собой структурированный язык, ориентированный на последовательное выполнение команд, строгий контроль типов данных и удобство чтения исходного кода.

2.2 Выполнение трансляции собственного языка программирования под целевую платформу

Для верификации работоспособности разработанного компилятора и корректности правил трансляции была разработана тестовая программа, реализующая алгоритм сортировки пузырьком («Bubble Sort»). Данный алгоритм был выбран, так как он задействует ключевые механизмы языка: работу с массивами, вложенные циклы, условные переходы, ввод-вывод данных и вызов подпрограмм. Исходный код программы на языке GamePLang представлен в Приложении А.1.

Процесс трансляции начинается с лексического и синтаксического анализа исходного текста. Компилятор определяет структуру программы, начиная с ключевого слова FORMATION ChampionshipSort.

На этапе генерации кода выполняются следующие преобразования, результат которых (код на языке C) представлен в Приложении А.2:

1) Трансляция переменных (Секция PLAYERS):

   Все переменные, объявленные в секции PLAYERS, переносятся в глобальную область видимости целевого файла. Для предотвращения коллизий имен применяется механизм «манглинга» (name mangling): к каждому идентификатору добавляется префикс gp_;

   - Массив #team_scores преобразуется в массив int gp_team_scores_arr[100];
   - Скалярные переменные, такие как #count, #i, #total_score, преобразуются в тип int.

В начале функции main генерируется код инициализации, обнуляющий значения всех переменных, что гарантирует предсказуемое поведение программы.

2) Ввод-вывод данных:
   - Команда HUDDLE, используемая для вывода сообщений (например, "Coach, how many..."), транслируется в стандартную функцию printf;
   - Команда RECRUIT транслируется в функцию scanf. Важной особенностью реализации является автоматическая генерация кода обработки ошибок: если scanf не может считать число, программа завершает работу с кодом ошибки exit(1). Это видно в строках проверки ввода переменных gp_count и элементов массива gp_team_scores_arr.
3) Управляющие конструкции:
   - Циклы DRILL, используемые в GamePLang для итерации по массиву и реализации алгоритма сортировки, однозначно отображаются в циклы for языка C. Компилятор корректно переносит условия остановки цикла (#count) и переменные-счетчики (#i, #j);
   - Условная конструкция WHEN преобразуется в оператор if. Вложенность условий (проверка выхода за границы массива и сравнение элементов для перестановки) полностью сохранена в целевом коде.
4) Операции и выражения:

   Словесные операторы языка GamePLang заменяются на их математические аналоги в C:
   - Оператор ADD в выражении #next_idx = #j ADD 1 заменяется на +;

- Спортивный модификатор SCORE (в строке #total_score SCORE ...) транслируется в оператор составного присваивания +=, что позволяет накапливать сумму очков;
- Операторы DIVIDE и MULTIPLY преобразуются в / и * соответственно.

5) Подпрограммы:

   Функция show_standings, описанная в блоке PLAY, вынесена в отдельную функцию void func_show_standings() в целевом коде. Её вызов через команду EXECUTE в основной секции транслируется в прямой вызов функции в main.

Анализ полученного файла output.c (Приложение А.2) показывает, что компилятор успешно построил семантически эквивалентную программу. В сгенерированном коде присутствуют необходимые заголовочные файлы (<stdio.h>, <stdlib.h>), корректно объявлены типы данных и соблюдена логика алгоритма. Полученный файл готов к компиляции стандартным компилятором GCC и последующему запуску на целевой платформе.

2.3 Разработка лексического анализатора

Лексический анализатор (файл lexer.l) реализован с использованием инструмента Flex. Его основная функция — сканирование входного потока символов исходного кода GamePLang и их группировка в значащие последовательности — токены, которые затем передаются синтаксическому анализатору. Код лексического анализатора представлен в Приложении А.3.

Анализатор настроен на распознавание следующих категорий лексем, специфичных для разработанного языка:

1) Ключевые слова: Зарезервированные команды, определяющие структуру программы и управляющие конструкции (например, FORMATION, SEQUENCE, DRILL, WHEN, RECRUIT);

2) Идентификаторы переменных (Игроков): Уникальной особенностью лексики является формат имен переменных. Они распознаются по обязательному префиксу #, за которым следует буквенно-цифровая последовательность (например, #score, #team1). Регулярное выражение для них имеет вид: #{LETTER}({LETTER}|{DIGIT})*;
3) Литералы:
   - Числовые: Последовательности цифр, интерпретируемые как целые числа;
   - Строковые: Текст, заключенный в двойные кавычки (например, "Game Over"). Анализатор корректно обрабатывает содержимое строки, включая экранированные символы.
4) Операторы: Лексер поддерживает двойную систему обозначений: символьную (например, =, >) и словесную (например, ADD, SCORE, ABOVE). При обнаружении словесного оператора он преобразуется в соответствующий токен операции (например, слово ADD возвращает токен PLUS_OP);
5) Разделители: Знаки пунктуации, такие как двоеточие (COLON) для начала блоков и точка с запятой (SEMICOLON) для завершения инструкций.

Обработка строковых литералов в GamePLang реализована через стандартные регулярные выражения без использования отдельных состояний сканера. При обнаружении идентификатора или литерала их текстовое значение копируется в глобальную переменную yylval.string_value с помощью функции strdup, что позволяет сохранить данные для последующего использования в парсере.

Анализатор выполняет фильтрацию входного потока: пробельные символы (пробелы, табуляция, переводы строк) игнорируются. Также

реализован пропуск однострочных комментариев, начинающихся с символов // и продолжающихся до конца строки. Любые символы, не подпадающие ни под одно из правил (например, недопустимые спецсимволы), вызывают ошибку лексического анализа.

Структура файла спецификации Flex соответствует стандарту:

1) Секция определений (между %{ и %}): Здесь подключается заголовочный файл parser.tab.h, сгенерированный Bison, что обеспечивает единую нумерацию токенов между лексером и парсером;
2) Секция правил (после первого %%): Содержит пары «регулярное выражение — действие на языке C». Именно здесь определяется логика возврата токенов;
3) Пользовательский код: Содержит вспомогательные функции для обработки ошибок.

Такая реализация обеспечивает полную совместимость с синтаксическим анализатором и гарантирует корректное выделение всех лексических единиц языка GamePLang перед этапом построения абстрактного синтаксического дерева.

2.4 Разработка синтаксического анализатора

Синтаксический анализатор для GamePLang создан с использованием GNU Bison. Он проверяет структуру программы по заданной грамматике и генерирует код на C. Каждое синтаксическое правило содержит семантическое действие, которое формирует соответствующий фрагмент кода на C при разборе конструкций языка. Анализатор обрабатывает объявления функций, создание переменных, выполнение команд, условия, циклы и выражения, обеспечивая корректную структуру программы и преобразование в целевой код. В приложении А.4 представлен код синтаксического анализатора.

Для однозначной интерпретации исходного кода компилятором была разработана контекстно-свободная грамматика, описывающая синтаксические правила языка GamePLang. Для наглядного представления правил вывода использованы синтаксические диаграммы, которые позволяют визуализировать структуру языковых конструкций и порядок следования лексем. Полные схемы грамматики вынесены в Приложение Б.

Общая структура программы

Корневым нетерминалом грамматики является Program, который определяет общую архитектуру файла с исходным кодом. Любая программа представляет собой единый блок FormationBlock, инкапсулирующий все данные и логику.

Согласно правилам, представленным в Приложении Б.1, структура программы строго регламентирована и состоит из последовательности:

1) Ключевого слова FORMATION и идентификатора программы;
2) Секции объявления данных (PlayersDecl);
3) Основной секции выполнения (Sequence);
4) Секции определения подпрограмм (FunctionsDecl);
5) Завершающего терминала END_FORMATION.

Объявление данных

Секция PlayersDecl (см. Приложение Б.2) отвечает за объявление глобальных переменных. Грамматика различает два типа объявлений:

1) Скалярные переменные (ScalarDecl): инициализация переменной, начинающейся с символа #, результатом вычисления выражения;
2) Массивы (ArrayDecl): инициализация переменной списком выражений, заключенным в квадратные скобки.

Конструкция VarDecls реализована рекурсивно, что позволяет объявлять неограниченное количество переменных, разделенных точкой с запятой.

Управляющие конструкции и операторы

Основная логика программы описывается нетерминалом Sequence, который содержит список операторов StatementList. В Приложении Б.3, Приложении Б.4 приведена детализация возможных операторов (Statement), к которым относятся:

1) Присваивание значений (Assignment) и модификация переменных (VarModification с операторами SCORE, ADD и т.д.);
2) Ввод-вывод данных (InputStatement, PrintStatement);
3) Вызовы функций (FunctionCall);
4) Блоки управления потоком выполнения (LoopStatement, WhenStatement).

Циклические конструкции

Язык GamePLang обладает развитой системой циклов, схемы которых представлены в Приложении Б.5. Грамматика выделяет три типа итерационных конструкций:

1) Простые циклы (DrillLoop и QuarterLoop): используют счетчик итераций и ключевое слово TIMES;
2) Составные циклы (ExerciseBlock): представляют собой сложную вложенную структуру, где блок EXERCISE обязательно содержит SETS, который, в свою очередь, содержит REPS. Такая жесткая иерархия гарантирует правильную структуру вложенности, характерную для предметной области языка.

Условные конструкции

Ветвление в программе реализуется через нетерминал WhenStatement (см. Приложение Б.6). Грамматика поддерживает две формы:

1) SimpleWhen: конструкция с единственной веткой THEN;
2) WhenElse: полная конструкция, включающая альтернативную ветку OTHERWISE.

Особенностью грамматики является использование явных терминалов начала и конца блока (WHEN ... END_WHEN), что исключает неоднозначность «висячего else», характерную для многих C-подобных языков.

Подпрограммы

Определение функций описывается правилом FunctionDef, представленным в Приложении Б.7. Грамматика разделяет функции на два типа:

1) SimpleFunction: подпрограммы без аргументов;
2) FunctionWithParams: подпрограммы, принимающие список параметров (ParamList) через ключевое слово WITH.

Тело функции (StatementList) использует те же правила построения операторов, что и основной блок программы.

Выражения и лексические единицы

Низкоуровневая структура языка, включая приоритеты операций и состав выражений (Expression), приведена в Приложении Б.8. Диаграммы демонстрируют иерархию от простейших факторов (Literal, Variable) до термов и полных арифметических выражений. Здесь же описаны правила формирования идентификаторов переменных (PlayerRef), приведена в Приложении Б.9, Приложении Б.10, требующих обязательного префикса #, и строковых литералов.

Разработанная грамматика обеспечивает однозначный разбор исходного кода и служит основой для автоматической генерации синтаксического анализатора.

2.5 Компиляция компилятора

Процесс создания исполняемого файла компилятора GamePLang представляет собой многоступенчатую процедуру трансляции, в которой задействованы генераторы кода и системный компилятор языка C. Данный процесс трансформирует высокоуровневые спецификации лексики и грамматики в машинный код.

Сборка выполняется в строгой последовательности, обусловленной зависимостями между модулями:

1) Генерация синтаксического анализатора. На первом этапе утилита Bison обрабатывает файл грамматики parser.y. С флагом -d она генерирует два файла:
   - parser.tab.c — исходный код парсера на языке C;
   - parser.tab.h — заголовочный файл, содержащий определения токенов (необходим для лексического анализатора).
2) Генерация лексического анализатора. Утилита Flex принимает на вход файл спецификаций lexer.l и, используя сгенерированный ранее заголовочный файл, создает файл lex.yy.c. Этот файл содержит реализацию конечного автомата для распознавания токенов;
3) Компиляция и линковка. На финальном этапе компилятор GCC компилирует полученные C-файлы (parser.tab.c и lex.yy.c). Затем происходит этап линковки (связывания) с библиотекой Flex (-lfl), в результате чего формируется единый исполняемый файл gameplan.

Полученный бинарный файл является готовым транслятором, способным принимать исходный код GamePLang и генерировать валидный

код на языке C. Компилятор принимает в качестве аргумента командной строки имя файла с исходным кодом на языке GamePLang и выводит на стандартный поток сгенерированный код на языке C.

2.6 Создание Makefile

Для оптимизации процесса разработки, управления зависимостями и автоматизации рутинных операций был разработан сценарий сборки для утилиты Make, файл Makefile (см. Приложение А.5). Использование данного инструмента гарантирует, что при изменении одного из исходных файлов будут перекомпилированы только необходимые компоненты проекта.

Структура разработанного Makefile включает следующие ключевые элементы:

1) Определение переменных: В начале файла задаются переменные для компилятора (CC=gcc), генераторов (BISON, FLEX), флагов линковки (LIBS) и имени целевого файла (TARGET=gameplan). Это позволяет легко менять инструменты или настройки проекта в одном месте;
2) Правила зависимостей:
   - Цель all указывает на главный исполняемый файл;
   - Правила для объектных файлов описывают, как из parser.y получить C-код парсера, а из lexer.l — код лексера. Make автоматически отслеживает время изменения файлов и запускает пересборку только при необходимости.
3) Очистка проекта: Цель clean содержит команды для удаления всех временных и сгенерированных файлов, возвращая директорию проекта в исходное состояние.

Скрипт автоматической проверки

Особое внимание уделено цели test, которая представляет собой скрипт для комплексной проверки работоспособности компилятора (End-to-End тестирование). При вызове команды make test FILE=имя_файла.gp выполняется следующий конвейер:

1) Сборка: Проверка актуальности сборки самого компилятора gameplan;
2) Трансляция: Запуск gameplan с передачей ему тестового файла. Результат трансляции перенаправляется в файл output.c;
3) Компиляция результата: Вызов GCC для компиляции сгенерированного файла output.c в исполняемый файл final_program;
4) Запуск: Автоматический запуск итоговой программы final_program.

Такой подход позволяет одной командой пройти полный цикл от исходного кода на GamePLang до работающего приложения, что существенно ускоряет процесс отладки и демонстрации возможностей языка.

2.7 Контрольный пример

Для проверки корректной работы компилятора языка GamePLang был выполнен контрольный пример. На вход компилятора подаётся исходный код программы на языке GamePLang, представленный в Приложении А.1. Результат трансляции кода на язык С и работы программы выведены в консоль (Рисунок1 и Рисунок 2 соответственно).

```
skydead@DESKTOP-I4TSEE6:~/kursach$ cat output.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Helpers
char* __gp_to_str(int v) { char* b = malloc(32); sprintf(b, "%d", v); return b; }
char* __gp_concat(char* a, char* b) {
    char* res = malloc(strlen(a) + strlen(b) + 1);
    strcpy(res, a); strcat(res, b); return res;
}

void func_show_standings();

// Переменные
int gp_team_scores_arr[100] = {0};
int gp_count = 0;
int gp_i = 0;
int gp_j = 0;
int gp_k = 0;
int gp_next_idx = 0;
int gp_val_current = 0;
int gp_val_next = 0;
int gp_temp = 0;
int gp_total_score = 0;
int gp_average_approx = 0;

int main() {
    { int count = 9; for(int i=0; i<count && i<100; i++) gp_team_scores_arr[i] = 0; }
    gp_count = 0;
    gp_i = 0;
    gp_j = 0;
    gp_k = 0;
    gp_next_idx = 0;
    gp_val_current = 0;
    gp_val_next = 0;
    gp_temp = 0;
    gp_total_score = 0;
    gp_average_approx = 0;

    // Код
    printf("%s\n", "Coach, how many players to sort (max 10)?");
    if(scanf("%d", &gp_count)!=1) { printf("Error: Number expected\n"); exit(1); }
    printf("%s\n", "Enter scores for each player:");
    for(gp_i = 0; gp_i < gp_count; gp_i++) {
    printf("%s\n", "Score for player:");
    if(scanf("%d", &gp_team_scores_arr[gp_i])!=1) { printf("Error: Number expected\n"); exit(1); }
    }
    printf("%s\n", "--- Starting Bubble Sort Drill ---");
    for(gp_i = 0; gp_i < gp_count; gp_i++) {
    for(gp_j = 0; gp_j < gp_count; gp_j++) {
    gp_next_idx = (gp_j + 1);
    if ((gp_next_idx < gp_count)) {
    gp_val_current = gp_team_scores_arr[gp_j];
    gp_val_next = gp_team_scores_arr[gp_next_idx];
    if ((gp_val_current > gp_val_next)) {
    gp_temp = gp_val_current;
    gp_team_scores_arr[gp_j] = gp_val_next;
    gp_team_scores_arr[gp_next_idx] = gp_temp;
    }
    }
    }
    }
    func_show_standings();
    printf("%s\n", "--- Calculating Stats ---");
    for(gp_k = 0; gp_k < gp_count; gp_k++) {
    gp_total_score += gp_team_scores_arr[gp_k];
    }
    printf("%s\n", "Total Team Score:");
    printf("%d\n", gp_total_score);
    gp_average_approx = (gp_total_score / gp_count);
    gp_average_approx = (gp_average_approx * 10);
    printf("%s\n", "Average Score (x10 scaled):");
    printf("%d\n", gp_average_approx);

    return 0;
}

void func_show_standings() {
    printf("%s\n", "--- Final Standings (Sorted) ---");
    for(gp_k = 0; gp_k < gp_count; gp_k++) {
    printf("%d\n", gp_team_scores_arr[gp_k]);
    }
}
skydead@DESKTOP-I4TSEE6:~/kursach$
```

Рисунок 1 – Результат компиляции кода на язык C

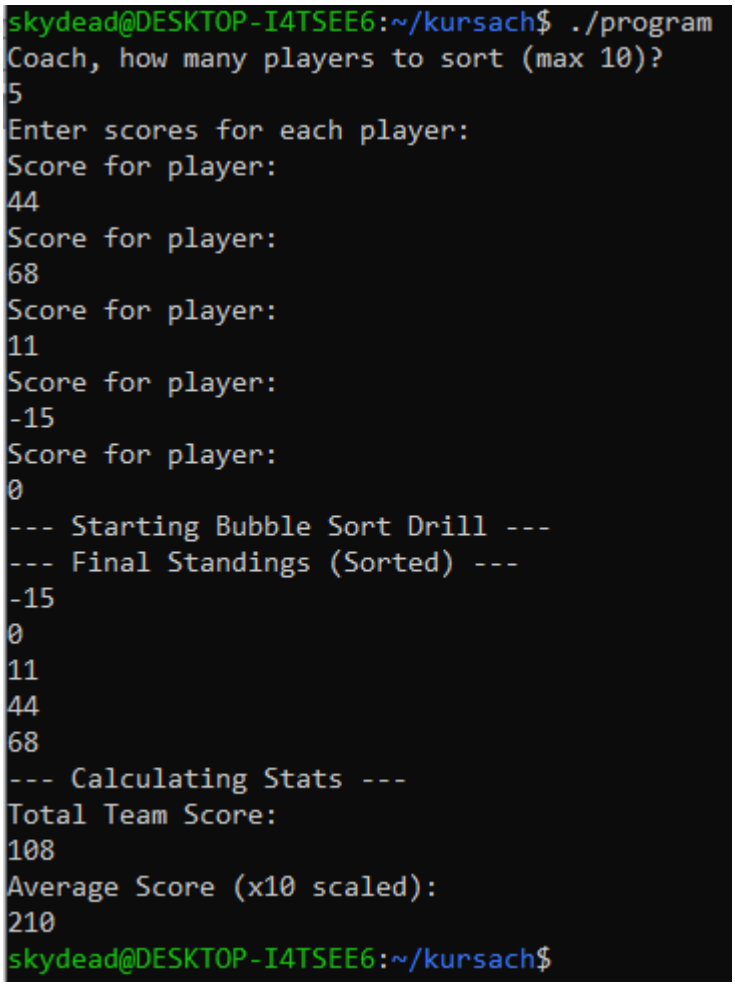

Рисунок 4 – Результат работы программы

Полученный результат соответствует ожидаемому, в связи с чем работу компилятора можно считать корректной.

ЗАКЛЮЧЕНИЕ

В результате работы разработан компилятор собственного языка программирования, генерирующий программный код на языке C. Язык программирования собственной разработки позволяет объявлять целочисленные переменные, уменьшать и увеличивать их значение на заданную величину, выводить значение переменной на печать, а также создавать циклы и функции.

В перспективе можно расширить возможности языка, увеличив количество типов данных, и добавив больше различных синтаксических конструкций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гриффитс Д., Гриффитс Д. Изучаем программирование на C: учебник. – СПб.: Питер, 2013. – 350 с.

2. Документация по языку программирования C [Электронный ресурс]. – 2022. – URL: https://learn.microsoft.com/ru-ru/cpp/c-language/?view=msvc-170&viewFallbackFrom=vs-2019 (дата обращения 20.11.2025).

3. Документация по утилите bison [Электронный ресурс]. – 2022. – URL: https://www.gnu.org/software/bison/manual/bison.html (дата обращения 05.12.2025);

4. Документация по утилите GNU Flex [Электронный ресурс]. – Режим доступа: https://docs.jade.fyi/gnu/flex.html (дата обращения: 02.12.2025).

ПРИЛОЖЕНИЕ А

Листинг кода

Листинг А.1 – исходный код программы сортировки bubbleFun.gp

```
FORMATION ChampionshipSort

PLAYERS:
   // Объявляем массив (с запасом, так как размер статический)
   #team_scores = [9, 0];

   // Переменные
   #count = 0;
   #i = 0;
   #j = 0;
   #k = 0;

   // Вспомогательные переменные для обмена
   #next_idx = 0;
   #val_current = 0;
   #val_next = 0;
   #temp = 0;

   // Для финальных расчетов
   #total_score = 0;
   #average_approx = 0;

SEQUENCE:
   // 1. Ввод данных (Recruiting phase)
   HUDDLE "Coach, how many players to sort (max 10)?";
   RECRUIT #count;

   HUDDLE "Enter scores for each player:";
   DRILL #count TIMES AS #i:
      HUDDLE "Score for player:";
      RECRUIT #team_scores[#i];
```

```
END_DRILL

// 2. Сортировка (Training phase)
HUDDLE "--- Starting Bubble Sort Drill ---";

// Внешний цикл
DRILL #count TIMES AS #i:

   // Внутренний цикл
   DRILL #count TIMES AS #j:

      // Используем слово ADD вместо +
      #next_idx = #j ADD 1;

      // Проверяем границы массива
      WHEN #next_idx < #count THEN:

         #val_current = #team_scores[#j];
         #val_next = #team_scores[#next_idx];

         // Если текущий больше следующего - меняем местами
         WHEN #val_current > #val_next THEN:
            #temp = #val_current;
            #team_scores[#j] = #val_next;
            #team_scores[#next_idx] = #temp;
         END_WHEN

      END_WHEN

   END_DRILL
END_DRILL

// 3. Вывод результатов через функцию
EXECUTE show_standings;
```

```
    // 4. Демонстрация словесных операций
    HUDDLE "--- Calculating Stats ---";

    DRILL #count TIMES AS #k:
        // SCORE работает как += (увеличивает счет)
        #total_score SCORE #team_scores[#k];
    END_DRILL

    HUDDLE "Total Team Score:";
    HUDDLE #total_score;

    // Используем слова DIVIDE и MULTIPLY
    #average_approx = #total_score DIVIDE #count;
    #average_approx = #average_approx MULTIPLY 10; // Просто для примера умножения

    HUDDLE "Average Score (x10 scaled):";
    HUDDLE #average_approx;

END_SEQUENCE

// Функция для вывода отсортированного массива
PLAY show_standings:
    HUDDLE "--- Final Standings (Sorted) ---";
    DRILL #count TIMES AS #k:
        HUDDLE #team_scores[#k];
    END_DRILL
END_PLAY

END_FORMATION
```

Листинг А.2 – результат трансляции исходного кода на язык C (output.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
// Helpers
char* __gp_to_str(int v) { char* b = malloc(32); sprintf(b, "%d", v); return b; }
char* __gp_concat(char* a, char* b) {
    char* res = malloc(strlen(a) + strlen(b) + 1);
    strcpy(res, a); strcat(res, b); return res;
}

void func_show_standings();

// Переменные
int gp_team_scores_arr[100] = {0};
int gp_count = 0;
int gp_i = 0;
int gp_j = 0;
int gp_k = 0;
int gp_next_idx = 0;
int gp_val_current = 0;
int gp_val_next = 0;
int gp_temp = 0;
int gp_total_score = 0;
int gp_average_approx = 0;

int main() {
    { int count = 9; for(int i=0; i<count && i<100; i++) gp_team_scores_arr[i] = 0; }
    gp_count = 0;
    gp_i = 0;
    gp_j = 0;
    gp_k = 0;
    gp_next_idx = 0;
    gp_val_current = 0;
    gp_val_next = 0;
    gp_temp = 0;
    gp_total_score = 0;
    gp_average_approx = 0;
```

```
    // Код
    printf("%s\n", "Coach, how many players to sort (max 10)?");
    if(scanf("%d", &gp_count)!=1) { printf("Error: Number expected\n"); exit(1); }
    printf("%s\n", "Enter scores for each player:");
    for(gp_i = 0; gp_i < gp_count; gp_i++) {
    printf("%s\n", "Score for player:");
    if(scanf("%d", &gp_team_scores_arr[gp_i])!=1) { printf("Error: Number expected\n"); exit(1); }
    }
    printf("%s\n", "--- Starting Bubble Sort Drill ---");
    for(gp_i = 0; gp_i < gp_count; gp_i++) {
    for(gp_j = 0; gp_j < gp_count; gp_j++) {
    gp_next_idx = (gp_j + 1);
    if ((gp_next_idx < gp_count)) {
    gp_val_current = gp_team_scores_arr[gp_j];
    gp_val_next = gp_team_scores_arr[gp_next_idx];
    if ((gp_val_current > gp_val_next)) {
    gp_temp = gp_val_current;
    gp_team_scores_arr[gp_j] = gp_val_next;
    gp_team_scores_arr[gp_next_idx] = gp_temp;
    }
    }
    }
    }
    func_show_standings();
    printf("%s\n", "--- Calculating Stats ---");
    for(gp_k = 0; gp_k < gp_count; gp_k++) {
    gp_total_score += gp_team_scores_arr[gp_k];
    }
    printf("%s\n", "Total Team Score:");
    printf("%d\n", gp_total_score);
    gp_average_approx = (gp_total_score / gp_count);
    gp_average_approx = (gp_average_approx * 10);
    printf("%s\n", "Average Score (x10 scaled):");
    printf("%d\n", gp_average_approx);
```

```
    return 0;
}

void func_show_standings() {
    printf("%s\n", "--- Final Standings (Sorted) ---");
    for(gp_k = 0; gp_k < gp_count; gp_k++) {
    printf("%d\n", gp_team_scores_arr[gp_k]);
    }
}
```

Листинг А.3 – Код лексического анализатора

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "parser.tab.h"

void yyerror(const char *s);
%}

%option noyywrap yylineno

DIGIT    [0-9]
LETTER   [a-zA-Za-яA-Я_]
ID       {LETTER}({LETTER}|{DIGIT})*
PLAYER_REF \#{ID}
STRING   \"([^"\\]|\\.)*\"
NUMBER   {DIGIT}+
COMMENT  \/\/[^\n]*

%%

[ \t\r\n]+          { /* Skip */ }

"FORMATION"         { return FORMATION; }
"END_FORMATION"     { return END_FORMATION; }
```

```
"SEQUENCE"        { return SEQUENCE; }
"END_SEQUENCE"    { return END_SEQUENCE; }
"PLAYERS"         { return PLAYERS; }
"PLAY"            { return PLAY; }
"END_PLAY"        { return END_PLAY; }

"WHEN"            { return WHEN; }
"THEN"            { return THEN; }
"OTHERWISE"       { return OTHERWISE; }
"END_WHEN"        { return END_WHEN; }

"DRILL"           { return DRILL; }
"END_DRILL"       { return END_DRILL; }
"QUARTER"         { return QUARTER; }
"END_QUARTER"     { return END_QUARTER; }
"EXERCISE"        { return EXERCISE; }
"END_EXERCISE"    { return END_EXERCISE; }
"SETS"            { return SETS; }
"END_SETS"        { return END_SETS; }
"REPS"            { return REPS; }
"END_REPS"        { return END_REPS; }
"TIMES"           { return TIMES; }
"AS"              { return AS; }

"HUDDLE"          { return HUDDLE; }
"EXECUTE"         { return EXECUTE; }
"CALL"            { return CALL; }
"RECRUIT"         { return RECRUIT; }
"RETURN"          { return RETURN; }
"BREAK_EXERCISE"  { return BREAK_EXERCISE; }

"SCORE"           { return SCORE_OP; }
"ADD"             { return PLUS_OP; }
"SUBTRACT"        { return MINUS_OP; }
"MULTIPLY"        { return MULTIPLY; }
```

```
"DIVIDE"          { return DIVIDE; }
"MODULO"          { return MODULO; }

"ABOVE"           { return GT_OP; }
"BELOW"           { return LT_OP; }
"EQUALS"          { return EQUALS_OP; }
"AND"             { return AND_OP; }

"WITH"            { return WITH; }
"="               { return ASSIGN_OP; }
">"               { return GT_OP; }
"<"               { return LT_OP; }
"{"               { return LBRACE; }
"}"               { return RBRACE; }
"("               { return LPAREN; }
")"               { return RPAREN; }
"["               { return LBRACKET; }
"]"               { return RBRACKET; }
","               { return COMMA; }
":"               { return COLON; }
";"               { return SEMICOLON; }

{PLAYER_REF}      { yylval.string_value = strdup(yytext); return PLAYER_REF; }
{STRING}          { yylval.string_value = strdup(yytext); return STRING_LITERAL; }
{NUMBER}          { yylval.string_value = strdup(yytext); return NUMBER_LITERAL; }
{ID}              { yylval.string_value = strdup(yytext); return IDENTIFIER; }

{COMMENT}         { }

.                 { fprintf(stderr, "Lexer error: %s\n", yytext); return -1; }

%%
```

Листинг А.4 – Код синтаксического анализатора

```
%{
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

extern int yylex();
extern int yylineno;
extern char* yytext;
void yyerror(const char *s);

/* === ТИПЫ ДАННЫХ === */
typedef enum {
    TYPE_UNKNOWN,
    TYPE_INT,       /* int */
    TYPE_STR,       /* char* */
    TYPE_ARR_INT,   /* int[] */
    TYPE_ARR_STR    /* char*[] */
} DataType;

typedef struct {
    char* name;
    DataType type;
} Symbol;

Symbol symbols[200];
int sym_count = 0;

/* === AST УЗЛЫ === */
typedef enum {
    NODE_SEQ, NODE_ASSIGN, NODE_ARR_INIT, NODE_ARR_SET, NODE_PRINT,
    NODE_LOOP, NODE_WHEN, NODE_OP, NODE_VAR, NODE_ARR_GET,
    NODE_CONST_INT, NODE_CONST_STR, NODE_CALL, NODE_INPUT,
NODE_MOD_VAR
} NodeType;

typedef struct ASTNode {
    NodeType type;
    struct ASTNode *left, *right, *cond;
```

```
    char* name;
    int op_code;

    int lit_int;        /* ТЕПЕРЬ ОБЫЧНЫЙ INT */
    char* lit_str;

    DataType dtype;
} ASTNode;

/* Хранилище функций */
typedef struct { char* name; ASTNode* body; } Function;
Function functions[50];
int func_count = 0;

char* param_buffer[10];
int param_buf_count = 0;

/* --- ПРОТОТИПЫ --- */
void add_var(char* name, DataType type);
DataType get_var_type(char* name);
char* to_c_name(char* name);
char* strip_quotes(char* s);

ASTNode* new_node(NodeType t, ASTNode* l, ASTNode* r);
ASTNode* new_int(int v);
ASTNode* new_str(char* s);
ASTNode* new_var(char* name);
ASTNode* new_arr_get(char* name, ASTNode* idx);
ASTNode* new_arr_set(char* name, ASTNode* idx, ASTNode* val);
ASTNode* new_arr_init(char* name, ASTNode* size, ASTNode* def_val);
ASTNode* new_call(char* name, ASTNode* args);

void register_func(char* name, ASTNode* body);

/* Генерация */
```

```
void codegen(ASTNode* n);
void generate_preamble();
void generate_vars();

%}

%union { char* string_value; struct ASTNode* node; }

%token <string_value> IDENTIFIER PLAYER_REF STRING_LITERAL NUMBER_LITERAL
%token FORMATION END_FORMATION SEQUENCE END_SEQUENCE PLAYERS PLAY END_PLAY
%token DRILL END_DRILL TIMES AS QUARTER END_QUARTER EXERCISE END_EXERCISE SETS END_SETS REPS END_REPS
%token WHEN END_WHEN THEN OTHERWISE HUDDLE EXECUTE CALL RETURN RECRUIT BREAK_EXERCISE
%token SCORE_OP PLUS_OP MINUS_OP MULTIPLY DIVIDE MODULO GT_OP LT_OP ASSIGN_OP EQUALS_OP AND_OP
%token COMMA COLON SEMICOLON LBRACE RBRACE LPAREN RPAREN LBRACKET RBRACKET WITH

%left AND_OP
%left EQUALS_OP GT_OP LT_OP
%left PLUS_OP MINUS_OP SCORE_OP
%left MULTIPLY DIVIDE MODULO

%type <node> statement statement_list sequence_section loop_statement when_statement expression expression_list array_literal
%type <node> drill_loop quarter_loop exercise_block sets_block reps_block call_statement arg_list_opt param_list var_decl var_decls players_opt

%start program

%%

program:
    FORMATION IDENTIFIER players_opt sequence_section functions_opt END_FORMATION {
```

```
        generate_preamble();
        printf("// Переменные\n");
        generate_vars();
        printf("\nint main() {\n");
        if ($3) codegen($3); // Инициализация
        printf("\n    // Код\n");
        codegen($4);         // Sequence
        printf("\n    return 0;\n");
        printf("}\n\n");
        for(int i=0; i<func_count; i++) {
            printf("void func_%s() {\n", functions[i].name);
            codegen(functions[i].body);
            printf("}\n");
        }
    }
  ;

players_opt: { $$ = NULL; } | PLAYERS COLON var_decls { $$ = $3; } ;
var_decls: { $$ = NULL; }
   | var_decls var_decl { if($1 == NULL) $$ = $2; else $$ = new_node(NODE_SEQ, $1, $2); } ;

var_decl:
     PLAYER_REF ASSIGN_OP expression SEMICOLON {
        add_var($1, $3->dtype);
        $$ = new_node(NODE_ASSIGN, $3, NULL);
        $$->name = strdup($1);
    }
  | PLAYER_REF ASSIGN_OP array_literal SEMICOLON {
        ASTNode* list = $3;
        ASTNode* size_node = NULL;
        ASTNode* def_node = NULL;

        if (list && list->type == NODE_SEQ) {
            size_node = list->left;
            if (list->right && list->right->type == NODE_SEQ) {
```

```
            def_node = list->right->left;
        }
    }

    if (!size_node || !def_node) {
        fprintf(stderr, "Syntax Error: Array declaration must be [SIZE, DEFAULT_VALUE]. Example: #arr = [10, 0];\n");
        exit(1);
    }

    DataType arr_type = (def_node->dtype == TYPE_STR) ? TYPE_ARR_STR : TYPE_ARR_INT;
    add_var($1, arr_type);

    $$ = new_arr_init($1, size_node, def_node);
  }
;

functions_opt: | functions_opt func_def ;
func_def:
   PLAY IDENTIFIER COLON statement_list END_PLAY { register_func($2, $4); }
 | PLAY IDENTIFIER WITH LPAREN param_list RPAREN COLON statement_list END_PLAY { register_func($2, $8); }
 ;

param_list:
   PLAYER_REF { param_buffer[param_buf_count++] = strdup($1); add_var($1, TYPE_INT); }
 | param_list COMMA PLAYER_REF { param_buffer[param_buf_count++] = strdup($3); add_var($3, TYPE_INT); }
 ;

sequence_section: SEQUENCE COLON statement_list END_SEQUENCE { $$ = $3; };

statement_list: { $$ = NULL; }
 | statement_list statement { if($1==NULL) $$=$2; else $$=new_node(NODE_SEQ, $1, $2); }
;
```

```
statement:
    PLAYER_REF ASSIGN_OP expression SEMICOLON {
        ASTNode* n = new_node(NODE_ASSIGN, $3, NULL); n->name = strdup($1); $$ = n;
    }
  | PLAYER_REF SCORE_OP expression SEMICOLON {
        ASTNode* n = new_node(NODE_MOD_VAR, $3, NULL); n->name = strdup($1); n->op_code = '+'; $$ = n;
    }
  | PLAYER_REF MINUS_OP expression SEMICOLON {
        ASTNode* n = new_node(NODE_MOD_VAR, $3, NULL); n->name = strdup($1); n->op_code = '-'; $$ = n;
    }
  | PLAYER_REF PLUS_OP expression SEMICOLON {
        ASTNode* n = new_node(NODE_MOD_VAR, $3, NULL); n->name = strdup($1); n->op_code = '+'; $$ = n;
    }
  | PLAYER_REF LBRACKET expression RBRACKET ASSIGN_OP expression SEMICOLON {
        $$ = new_arr_set($1, $3, $6);
    }
  | HUDDLE expression SEMICOLON { $$ = new_node(NODE_PRINT, $2, NULL); }
  | RECRUIT PLAYER_REF SEMICOLON {
        ASTNode* n = new_node(NODE_INPUT, NULL, NULL); n->name = strdup($2); $$ = n;
    }
  | RECRUIT PLAYER_REF LBRACKET expression RBRACKET SEMICOLON {
        ASTNode* n = new_node(NODE_INPUT, $4, NULL); n->name = strdup($2); $$ = n;
    }
  | call_statement SEMICOLON { $$ = $1; }
  | loop_statement { $$ = $1; }
  | when_statement { $$ = $1; }
  ;

call_statement:
    EXECUTE IDENTIFIER { $$ = new_call($2, NULL); }
  | CALL IDENTIFIER LPAREN arg_list_opt RPAREN { $$ = new_call($2, $4); }
  ;
```

```
arg_list_opt: { $$ = NULL; } | expression_list { $$ = $1; };
expression_list: expression { $$ = new_node(NODE_SEQ, $1, NULL); }
        | expression COMMA expression_list { $$ = new_node(NODE_SEQ, $1, $3); };

loop_statement: drill_loop | quarter_loop | exercise_block ;
drill_loop: DRILL expression TIMES AS PLAYER_REF COLON statement_list END_DRILL {
    add_var($5, TYPE_INT); ASTNode* n = new_node(NODE_LOOP, NULL, $7); n->cond = $2; n->name = strdup($5); $$ = n;
} ;
quarter_loop: QUARTER expression TIMES AS PLAYER_REF COLON statement_list END_QUARTER {
    add_var($5, TYPE_INT); ASTNode* n = new_node(NODE_LOOP, NULL, $7); n->cond = $2; n->name = strdup($5); $$ = n;
} ;
exercise_block: EXERCISE expression COLON sets_block END_EXERCISE { $$ = $4; } ;
sets_block: SETS expression TIMES AS PLAYER_REF COLON reps_block END_SETS {
  add_var($5, TYPE_INT); ASTNode* n = new_node(NODE_LOOP, NULL, $7); n->cond = $2; n->name = strdup($5); $$ = n;
} ;
reps_block: REPS expression TIMES AS PLAYER_REF COLON statement_list END_REPS {
  add_var($5, TYPE_INT); ASTNode* n = new_node(NODE_LOOP, NULL, $7); n->cond = $2; n->name = strdup($5); $$ = n;
} ;

when_statement:
    WHEN expression THEN COLON statement_list END_WHEN {
      ASTNode* n = new_node(NODE_WHEN, $5, NULL); n->cond = $2; $$ = n;
    }
  | WHEN expression THEN COLON statement_list OTHERWISE COLON statement_list END_WHEN {
      ASTNode* n = new_node(NODE_WHEN, $5, $8); n->cond = $2; $$ = n;
    }
  ;

expression:
    NUMBER_LITERAL { $$ = new_int(atoi($1)); }
```

```
   | STRING_LITERAL { $$ = new_str($1); }
   | PLAYER_REF { $$ = new_var($1); }
   | PLAYER_REF LBRACKET expression RBRACKET { $$ = new_arr_get($1, $3); }
   | LPAREN expression RPAREN { $$ = $2; }
   | expression PLUS_OP expression {
        ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '+';
        if ($1->dtype == TYPE_STR || $3->dtype == TYPE_STR) n->dtype = TYPE_STR;
        else n->dtype = TYPE_INT;
        $$ = n;
     }
   | expression MINUS_OP expression     { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '-'; n->dtype=TYPE_INT; $$=n; }
   | expression MULTIPLY expression     { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '*'; n->dtype=TYPE_INT; $$=n; }
   | expression DIVIDE expression       { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '/'; n->dtype=TYPE_INT; $$=n; }
   | expression MODULO expression       { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '%'; n->dtype=TYPE_INT; $$=n; }
   | expression GT_OP expression        { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '>'; n->dtype=TYPE_INT; $$=n; }
   | expression LT_OP expression        { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '<'; n->dtype=TYPE_INT; $$=n; }
   | expression EQUALS_OP expression    { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '='; n->dtype=TYPE_INT; $$=n; }
   | expression AND_OP expression       { ASTNode* n = new_node(NODE_OP, $1, $3); n->op_code = '&'; n->dtype=TYPE_INT; $$=n; }
   ;

array_literal: LBRACKET expression_list RBRACKET { $$ = $2; };

%%

/* === C CODE === */

char* to_c_name(char* name) {
   if(name[0] == '#') {
      char* buf = malloc(strlen(name) + 4); sprintf(buf, "gp_%s", name + 1); return buf;
   }
```

```
    return name;
}
char* strip_quotes(char* s) {
    if (!s) return NULL;
    if (s[0] == "") {
        char* new_s = strdup(s + 1);
        if(strlen(new_s)>0) new_s[strlen(new_s)-1] = '\0';
        return new_s;
    }
    return strdup(s);
}

void add_var(char* name, DataType type) {
    for(int i=0; i<sym_count; i++) if(strcmp(symbols[i].name, name)==0) return;
    symbols[sym_count].name = strdup(name);
    symbols[sym_count].type = type;
    sym_count++;
}
DataType get_var_type(char* name) {
    for(int i=0; i<sym_count; i++) if(strcmp(symbols[i].name, name)==0) return symbols[i].type;
    return TYPE_INT;
}

void register_func(char* name, ASTNode* body) {
    functions[func_count].name = strdup(name);
    functions[func_count].body = body;
    func_count++;
}

ASTNode* new_node(NodeType t, ASTNode* l, ASTNode* r) {
    ASTNode* n = malloc(sizeof(ASTNode));
    n->type = t; n->left = l; n->right = r; n->cond=NULL; n->name=NULL; n->dtype =
TYPE_INT;
    return n;
}
```

```
ASTNode* new_int(int v) { ASTNode* n = new_node(NODE_CONST_INT, NULL, NULL); n->lit_int = v; n->dtype = TYPE_INT; return n; }

ASTNode* new_str(char* s) { ASTNode* n = new_node(NODE_CONST_STR, NULL, NULL); n->lit_str = strip_quotes(s); n->dtype = TYPE_STR; return n; }

ASTNode* new_var(char* name) {

    ASTNode* n = new_node(NODE_VAR, NULL, NULL); n->name = strdup(name); n->dtype = get_var_type(name);

    return n;

}

ASTNode* new_arr_get(char* name, ASTNode* idx) {

    ASTNode* n = new_node(NODE_ARR_GET, idx, NULL); n->name = strdup(name);

    DataType arr_t = get_var_type(name);

    n->dtype = (arr_t == TYPE_ARR_STR) ? TYPE_STR : TYPE_INT;

    return n;

}

ASTNode* new_arr_set(char* name, ASTNode* idx, ASTNode* val) { ASTNode* n = new_node(NODE_ARR_SET, idx, val); n->name = strdup(name); return n; }

ASTNode* new_call(char* name, ASTNode* args) { ASTNode* n = new_node(NODE_CALL, args, NULL); n->name = strdup(name); return n; }


ASTNode* new_arr_init(char* name, ASTNode* size, ASTNode* def_val) {

    ASTNode* n = new_node(NODE_ARR_INIT, size, def_val);

    n->name = strdup(name);

    return n;

}


/* === ГЕНЕРАЦИЯ С-КОДА === */


void generate_preamble() {

    printf("#include <stdio.h>\n");

    printf("#include <stdlib.h>\n");

    printf("#include <string.h>\n\n");


    printf("// Helpers\n");

    printf("char* __gp_to_str(int v) { char* b = malloc(32); sprintf(b, \"%%d\", v); return b; }\n");

    printf("char* __gp_concat(char* a, char* b) { \n");

    printf("    char* res = malloc(strlen(a) + strlen(b) + 1); \n");
```

```
    printf("    strcpy(res, a); strcat(res, b); return res; \n");
    printf("}\n\n");

    for(int i=0; i<func_count; i++) printf("void func_%s();\n", functions[i].name);
    printf("\n");
}

void generate_vars() {
    for(int i=0; i<sym_count; i++) {
        char* c_name = to_c_name(symbols[i].name);
        if(symbols[i].type == TYPE_INT || symbols[i].type == TYPE_UNKNOWN) {
            printf("int %s = 0;\n", c_name);
        } else if(symbols[i].type == TYPE_STR) {
            printf("char* %s = \"\";\n", c_name);
        } else if(symbols[i].type == TYPE_ARR_INT) {
            printf("int %s_arr[100] = {0};\n", c_name);
        } else if(symbols[i].type == TYPE_ARR_STR) {
            printf("char* %s_arr[100] = {0};\n", c_name);
        }
    }
}

void codegen(ASTNode* n) {
    if (!n) return;

    switch(n->type) {
        case NODE_SEQ: codegen(n->left); codegen(n->right); break;

        case NODE_ASSIGN:
            printf("    %s = ", to_c_name(n->name));
            codegen(n->left);
            printf(";\n");
            break;

        case NODE_MOD_VAR:
```

```
            printf("    %s %c= ", to_c_name(n->name), n->op_code);
            codegen(n->left);
            printf(";\n");
            break;

        case NODE_INPUT: {
            DataType t = get_var_type(n->name);
            if(n->left) {
                if (t == TYPE_ARR_STR) {
                    printf("        { char buf[256]; if(scanf(\"%%255s\", buf)!=1) exit(1); %s_arr[", to_c_name(n->name));
                    codegen(n->left);
                    printf("] = strdup(buf); }\n");
                } else {
                    printf("    if(scanf(\"%%d\", &%s_arr[", to_c_name(n->name));
                    codegen(n->left);
                    printf("])!=1) { printf(\"Error: Number expected\\n\"); exit(1); }\n");
                }
            } else {
                if (t == TYPE_STR) {
                    printf("    { char buf[256]; if(scanf(\"%%255s\", buf)!=1) exit(1); %s = strdup(buf); }\n", to_c_name(n->name));
                } else {
                    printf("        if(scanf(\"%%d\", &%s)!=1) { printf(\"Error: Number expected\\n\"); exit(1); }\n", to_c_name(n->name));
                }
            }
            break;
        }

        case NODE_PRINT:
            if(n->left->dtype == TYPE_STR) {
                printf("    printf(\"%%s\\n\", ");
                codegen(n->left);
                printf(");\n");
            } else {
```

```
            printf("    printf(\"%%d\\n\", ");
            codegen(n->left);
            printf(");\n");
        }
        break;

    case NODE_LOOP:
        printf("    for(%s = 0; %s < ", to_c_name(n->name), to_c_name(n->name));
        codegen(n->cond);
        printf("; %s++) {\n", to_c_name(n->name));
        codegen(n->right);
        printf("    }\n");
        break;

    case NODE_WHEN:
        printf("    if ("); codegen(n->cond); printf(") {\n");
        codegen(n->left);
        printf("    }");
        if(n->right) { printf(" else {\n"); codegen(n->right); printf("    }"); }
        printf("\n");
        break;

    case NODE_CALL: printf("    func_%s();\n", n->name); break;

    case NODE_OP:
        if (n->op_code == '+' && n->dtype == TYPE_STR) {
            printf("__gp_concat(");
            if (n->left->dtype == TYPE_INT) { printf("__gp_to_str("); codegen(n->left); printf(")"); }
            else codegen(n->left);
            printf(", ");
            if (n->right->dtype == TYPE_INT) { printf("__gp_to_str("); codegen(n->right); printf(")"); }
            else codegen(n->right);
            printf(")");
        } else {
```

```
            printf("("); codegen(n->left);
            if(n->op_code=='=') printf(" == ");
            else if(n->op_code=='&') printf(" && ");
            else printf(" %c ", n->op_code);
            codegen(n->right); printf(")");
        }
        break;

    case NODE_VAR: printf("%s", to_c_name(n->name)); break;
    case NODE_CONST_INT: printf("%d", n->lit_int); break;
    case NODE_CONST_STR: printf("\"%s\"", n->lit_str); break;

    case NODE_ARR_GET:
        printf("%s_arr[", to_c_name(n->name)); codegen(n->left); printf("]");
        break;

    case NODE_ARR_SET:
        printf("    %s_arr[", to_c_name(n->name));
        codegen(n->left); printf("] = ");
        codegen(n->right); printf(";\n");
        break;

    case NODE_ARR_INIT:
        printf("    { int count = ");
        codegen(n->left);
        printf("; for(int i=0; i<count && i<100; i++) %s_arr[i] = ", to_c_name(n->name));
        codegen(n->right);
        printf("; }\n");
        break;
  }
}
void yyerror(const char *s) { fprintf(stderr, "Syntax Error: %s at line %d\n", s, yylineno); }
int main(int c, char **v) {
  if(c>1) { extern FILE* yyin; yyin=fopen(v[1],"r"); }
  yyparse();
```

```
    return 0;
}
```

Листинг А.5 – Код Makefile

```
CC = gcc
BISON = bison
FLEX = flex
LIBS = -lfl
TARGET = gameplan
PARSER_SRC = parser.y
LEXER_SRC = lexer.l

all: $(TARGET)

$(TARGET): parser.tab.c lex.yy.c
	$(CC) -o $(TARGET) parser.tab.c lex.yy.c $(LIBS)

parser.tab.c parser.tab.h: $(PARSER_SRC)
	$(BISON) -d $(PARSER_SRC)

lex.yy.c: $(LEXER_SRC) parser.tab.h
	$(FLEX) $(LEXER_SRC)

clean:
	rm -f $(TARGET) parser.tab.c parser.tab.h lex.yy.c output.c final_program

test: all
	./$(TARGET) $(FILE) > output.c
	$(CC) output.c -o final_program
	./final_program
```

ПРИЛОЖЕНИЕ Б

Схемы синтаксического анализатора

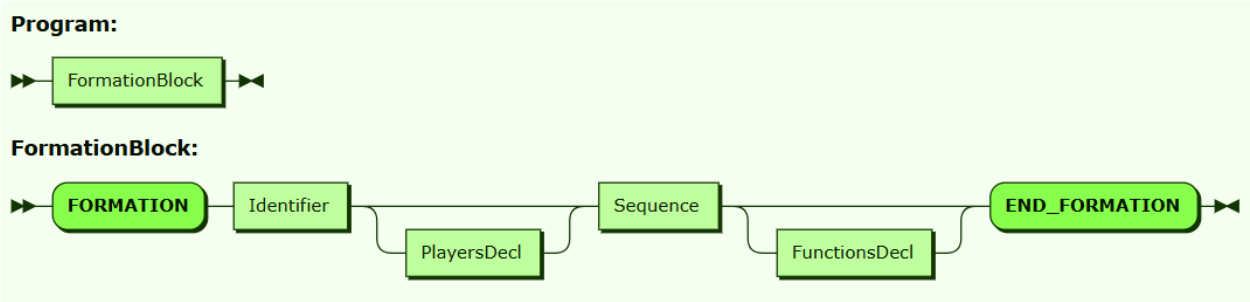

Рисунок Б.1 – Схема общей структуры

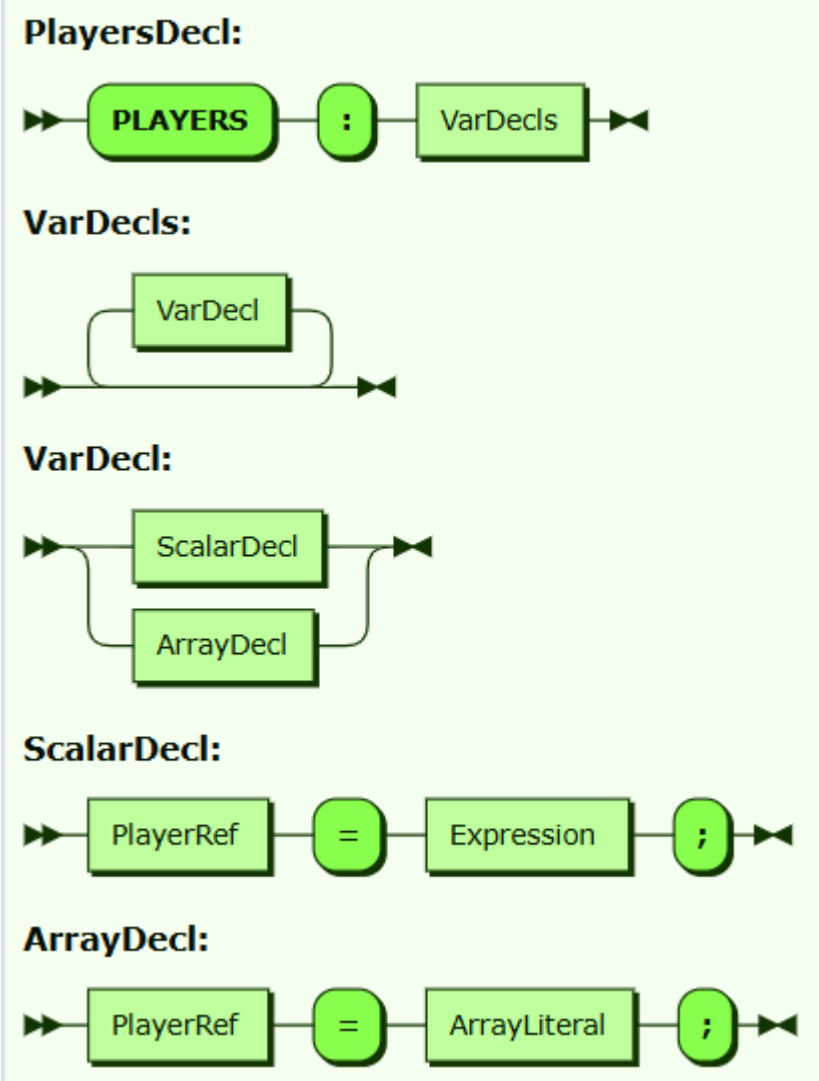

Рисунок Б.2 – Схема объявления данных

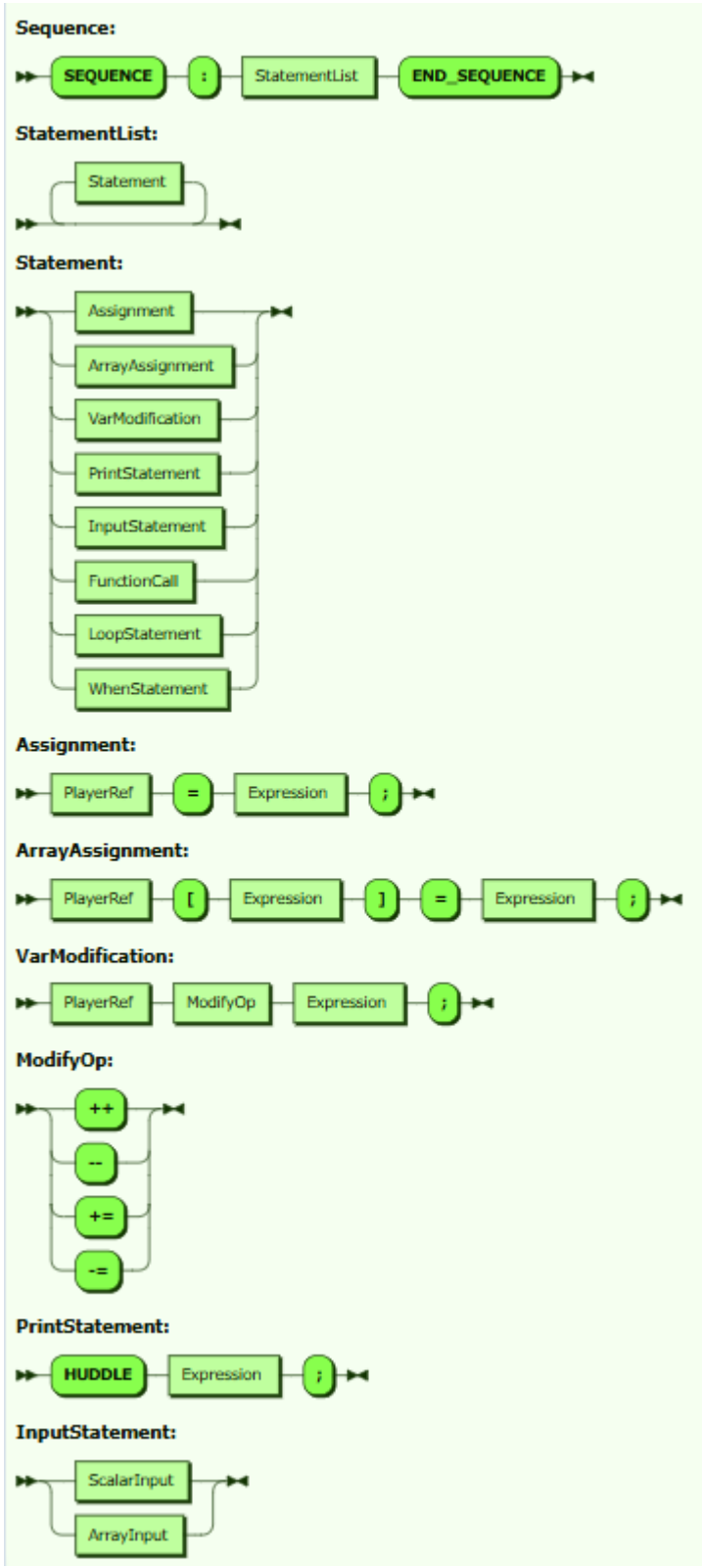

Рисунок Б.3 – Схема операторов часть 1

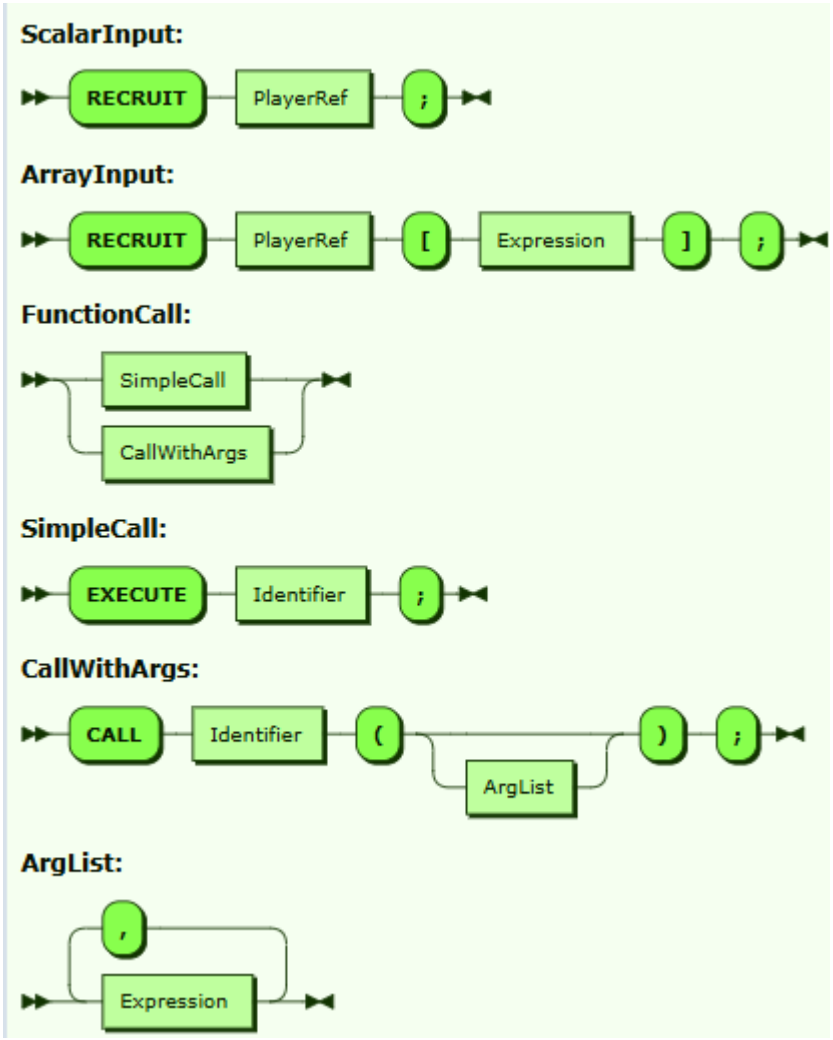

Рисунок Б.4 – Схема операторов часть 2

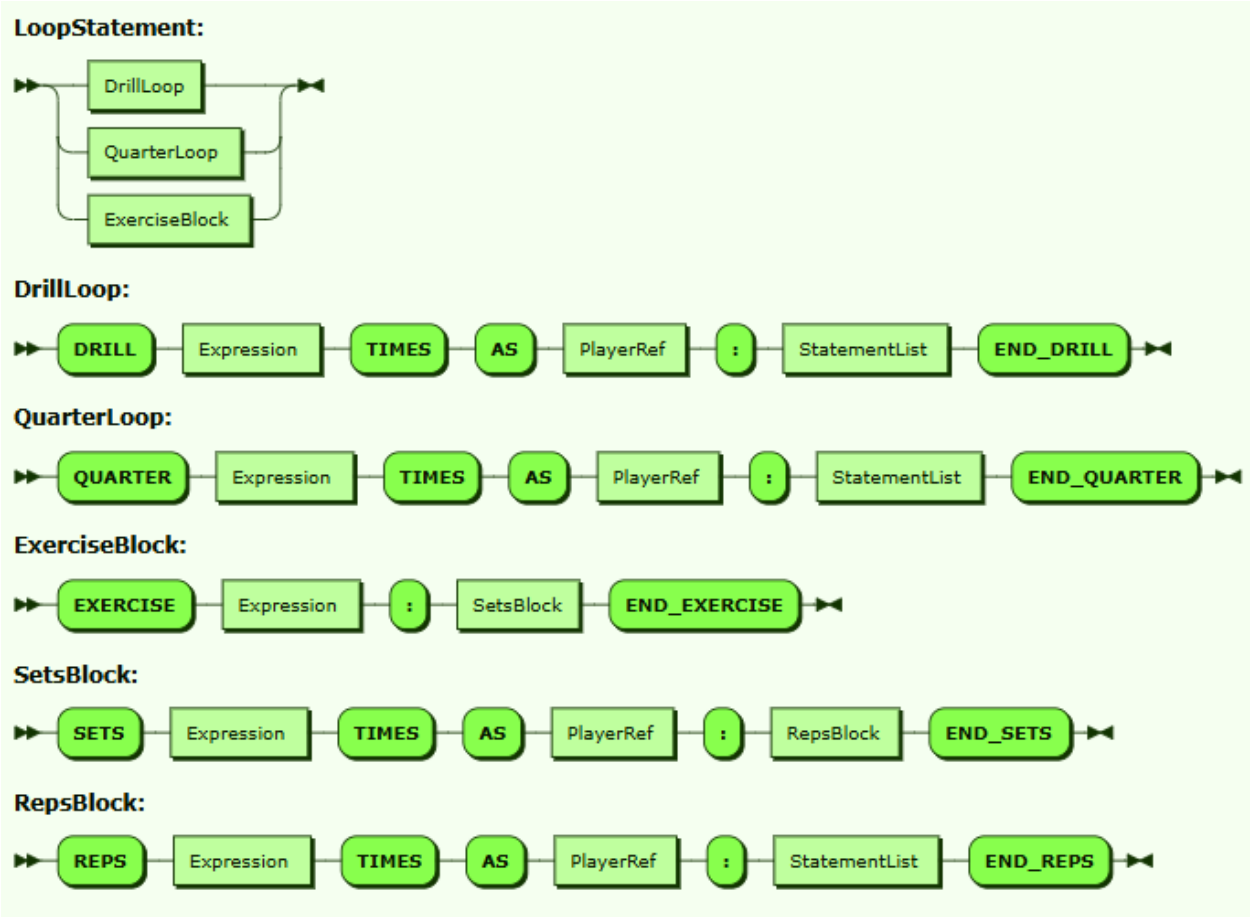

Рисунок Б.5 – Схема циклов

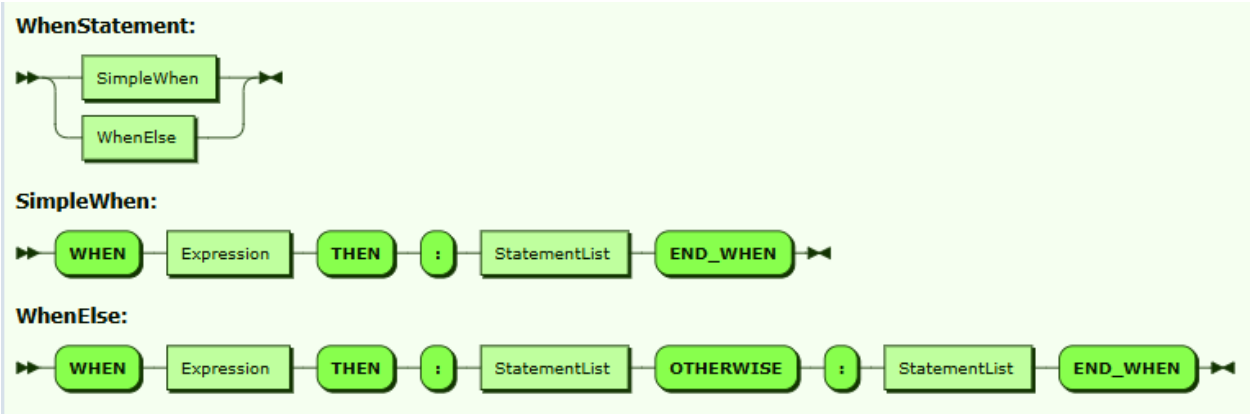

Рисунок Б.6 – Схема условий

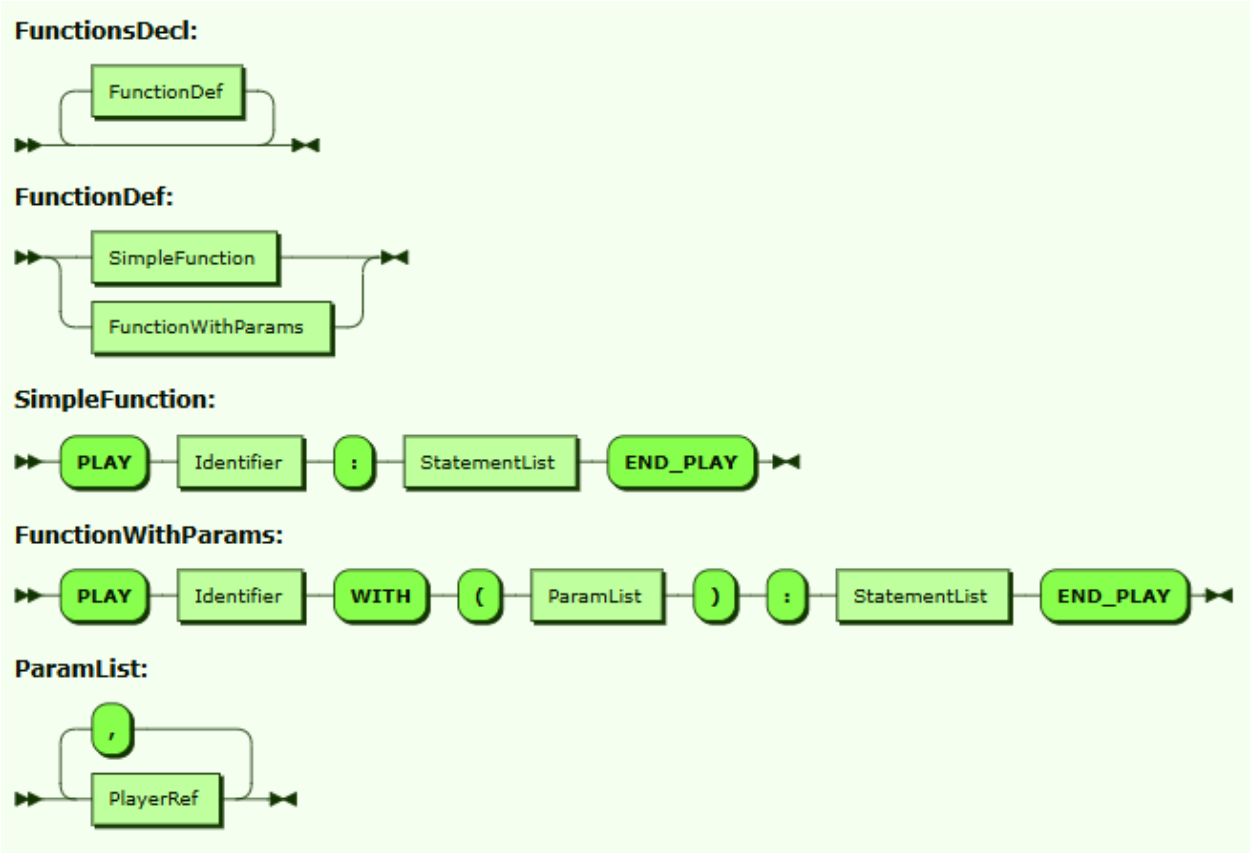

Рисунок Б.7 – Схема функций

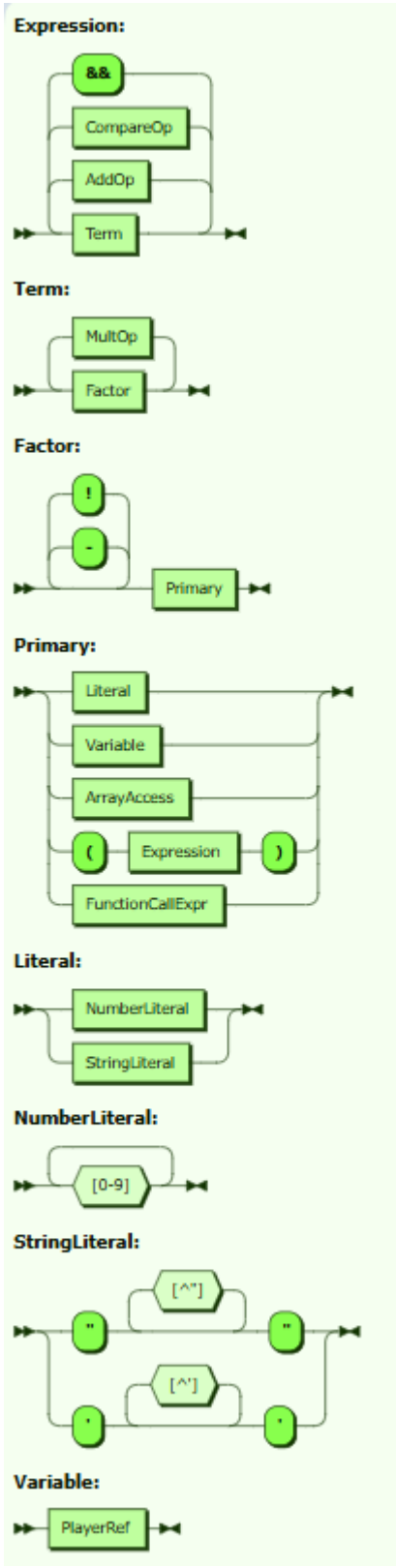

Рисунок Б.8 – Схема выражений и лексем

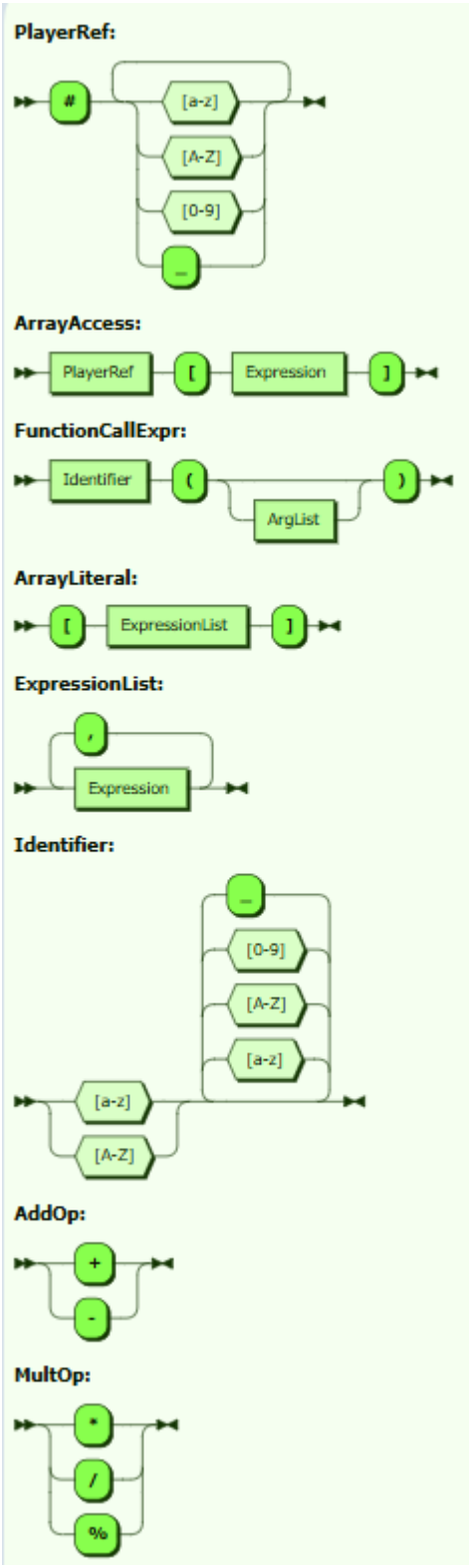

Рисунок Б.9 – Схема выражений и лексем

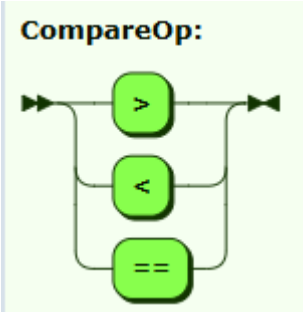

Рисунок Б.10 – Схема выражений и лексем