Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

А.А. Карпец

Разработка утилиты «Kmake» для управления компиляцией и сборкой проектов на основе автоматного подхода

Объектно-ориентированное программирование с явным выделением состояний

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2004

Оглавление

Введение

Известно, что в процессе создания программы на алгоритмических языках важными являются следующие этапы: написание исходного кода, компиляция, сборка (редактирование связей).

Для управления компиляцией и сборкой программ обычно применяется одна из отличающихся функциональностью модификаций утилиты *make*, которая входит в состав компиляторов различных производителей (*Borland, Microsoft, Symantec*). При этом утилита *make* воспринимает файл *Makefile* и генерирует необходимые для компиляции и сборки команды. Она распространяется, в том числе, и с открытыми исходными кодами (*GNU make*). Даже в этом случае из-за отсутствия проектной документации весьма трудно понять принципы ее функционирования.

Поэтому цель настоящей работы состоит в разработке утилиты рассматриваемого класса с небольшой функциональностью и открытой проектной документации к ней. В частности, разрабатываемая утилита не реализует возможности использования переменных в файлах *Makefile*. При этом для упрощения понимания функционирования программы и ее составных частей применяется автоматный подход [1]. Он отличается от традиционно используемого в этой области подхода [2] и близок к подходу, изложенному в работе [3]. Разработанная утилита названа *Kmake*.

Для создания утилиты, являющейся консольным приложением, используется язык *Cu* ++.

1. Постановка задачи

Рассмотрим процесс компиляции и сборки на примере проекта, написанного для определенности на языке *C* или *C++*.

Первая стадия этого процесса – компиляция, во время которой для каждого исходного файла (единицы компиляции) на применяемом языке генерируется объектный код. Он помещается в файл, который называется объектным файлом (модулем) и обычно имеет тоже имя, что и файл с исходным текстом, но другое расширение:

- `.obj` для *DOS* и *Windows*;
- `.o` для *UNIX*.

В объектных файлах содержится исполняемый код приложения, но вместо вызовов функций, находящихся в других единицах компиляции, стоят имена этих функций.

Вторая стадия – сборка (link). На этой стадии сборщик (редактор связей) (linker) – утилита, поставляемая вместе с компилятором, производит замену имен

функций на их код, взятый из других единиц компиляции (в случае статической сборки), или на адрес вызова этой функции (в случае динамической сборки).

Предположим, что во время разработки приложения модифицирован один из файлов с исходным кодом на рассматриваемом языке программирования. Для модификации исполняемого кода требуется произвести два действия:

- компиляцию модифицированного исходного файла;
- редактирование связей всех объектных файлов, из которых один является модифицированным вследствие предыдущего действия – компиляции.

Если необходимо модифицировать несколько единиц компиляции, то требуется произвести более двух действий – заново скомпилировать измененные исходные файлы и произвести редактирование связей всех объектных модулей. При этом разработчик должен помнить, какие исходные файлы он модифицировал и ввести для них команды вызова компилятора, а затем и команду вызова редактора связей.

Отслеживать, что и как необходимо в проекте модифицировать обременительно. Отметим, что описанная схема компиляции и сборки проекта является довольно простой. В проектах для графических оконных систем **(**например, *MS Windows, X Window System* (для *UNIX*)**)** помимо исходных файлов на языке *C* или *C++* в проектах присутствуют файлы ресурсов и библиотеки.

Эту проблему разработчики решают по-разному.

В случае небольшого проекта иногда просто пишется пакетный исполняемый файл, который компилирует все исходные файлы, а затем вызывает редактор связей. Таким образом, пакетный исполняемый файл производит действия, которые необходимо было бы производить, только в случае, если все исходные файлы были модифицированы. Следовательно, действия пакетного файла излишни. Однако в случае небольших проектов временные затраты незначительны. Поэтому разработчику не приходится долго ждать, пока произойдет компиляция и сборка всего приложения.

Однако в случае большого проекта, когда временные затраты на компиляцию всех файлов и последующее редактирование связей велики такое решение неприемлемо. Различные компиляторы и среды разработки решают эту проблему внешне различными, но, по мнению автора, принципиально одинаковыми средствами.

Решение состоит в следующем. При создании проекта задаются зависимости, каждая из которых представляет собой факт того, требуется ли модификация определенного файла из проекта при известном факте модификации других файлов. В различных средах разработки данные о зависимостях хранятся в специальных файлах проекта, формат которых определяется разработчиков среды. Форматы проектных файлов у продуктов, например, фирм *Borland* и *Microsoft*, различаются как между собой, так и в зависимости от версии среды разработки (продукта). В ряде сред проектные файлы не являются текстовыми - создаются

самими средами и ими же интерпретируются. Очевидно, что из-за этого возникают трудности с переносимостью проектов.

Однако большинство сред включают в себя универсальный интерпретатор *make*. Это компактная утилита, воспринимающая файл *Makefile*, в котором и содержатся данные о зависимостях. Преимуществами утилиты *make* по сравнению с аналогами являются два факта:

- файлы с данными о зависимостях (*Makefile*) являются текстовыми;
- утилита *make* широко распространена.

Автор использовал рассматриваемую утилиту для своих проектов. При этом его заинтересовало два аспекта.

Во-первых, является интересным алгоритм, по которому утилита выбирает порядок действий. Принципы работы утилиты изложены в литературе «вскользь» - формального описания алгоритма работы этой утилиты автор не встречал. Поэтому в настоящей работе предложен алгоритм и его реализация, однако, ввиду учебных целей, с меньшей функциональностью.

Во-вторых, возможности утилиты *make* отличаются в зависимости от производителя. Например, утилиты *GNU make* по возможностям гораздо шире аналога фирмы *Borland*. Кроме того, интересно выяснить имеются ли какие-либо стандарты на эту утилиту.

2. Структура файлов *Makefile*

Файлы *Makefile* представляют собой последовательность следующих конструкций для каждого файла проекта [4]:

```
<цель>: <зависимость1> <зависомость2> ... <зависимостьn>
    <команда1>
    <команда2>
    ...
    <командаm>
```

Здесь **<цель>** - описываемый файл проекта; **<зависимостьk>** - файл, при модификации которого, описываемый файл проекта должен быть модифицирован; **<команда1> ... <командаm>** - последовательность команд для модификации описываемого файла проекта.

Первая такая структура должная описывать файл приложения – конечную цель построения. Описания остальных файлов могут следовать в произвольном порядке.

3. Структура программы

Программа *kmake* написана на языке *C++* с использованием методов объектно-ориентированного программирования с явным выделением состояний [5], но без наследования. Она представляет собой два последовательно работающих блока.

Первый блок – синтаксический анализатор, производящий разбор входного файла *Makefile*. Основой его является класс *Parser*. Синтаксический анализатор реализован в виде метода класса *Parser* – автомата *A0*. Он не только производит синтаксический анализ файла *Makefile*, но и строит граф зависимостей.

Граф зависимостей – ориентированный ациклический связный граф, вершины которого соответствуют файлам, используемым при построении приложения. Если модификация одного файла должна повлечь модификацию другого, то в графе присутствует дуга, соединяющая вершину, соответствующую второму файлу, с вершиной, соответствующей первому файлу. Других дуг в графе зависимостей нет.

Второго блок в качестве основного содержит класс *DependencyGraph*, который на основе графа зависимостей, полученных от класса *Parser*, определяет, какие действия должны быть произведены и в какой последовательности. Для этого используется алгоритм, являющийся модификацией алгоритма топологической сортировки.

Модификация заключается в том, чтобы расположить вершины графа в порядке, указанном топологической сортировкой, и, кроме того, выявить те файлы, которые подлежат изменению. Это выполняется следующим образом: если хотя бы один из файлов, соответствующих потомкам какой-либо вершины, требует модификации, то и файл, соответствующий этой вершине требует модификации. Обе эти части выполняются вместе за один проход графа зависимостей в глубину. Алгоритм реализуется автоматом *A1*.

4. Класс *Parser*

4.1. Словесное описание

Класс *Parser* содержит в себе автомат *A0* (синтаксический анализатор), а также обеспечивает протоколирование состояний автомата. Входные переменные автомата организованы как методы класса – функции, возвращающие значение типа `bool`. Выходные воздействия автомата также организованы как методы класса – функции, возвращающие значение типа `void`.

4.2. Автомат *A0*

4.2.1. Словесное описание

Автомат *A0* – синтаксический анализатор, который на каждом такте воспринимает один и только один символ из входного файла *Makefile*. На основании текущего символа и своего состояния автомат принимает решение о переходе в другое состояние и о вызове функций, отвечающих выходным воздействиям. Автомат использует вспомогательные структуры: классы *Buffer* и *DependencyGraph*.

Класс *Buffer* предназначен для временного структурированного хранения прочитанных символов. Под таким хранением символов автор понимает то, что буфер содержит два поля:

- для имени описываемого файла;
- соответствующей ему зависимости или команды.

Класс *DependencyGraph* представляет собой структуру данных для хранения графа зависимостей, построение которого и ведется автоматом *A0*.

Автомат *A0* распознает последовательность из структур:

```
<цель>: <зависимость1> <зависомость2> ... <зависимостьn>
    <команда1>
    <команда2>
    ...
    <командаm>
```

Здесь **<цель>** - описываемый файл; **<зависимостьk>** - файл, при модификации которого описываемый файл, должен быть модифицирован; **<команда1> ... <командаm>** - последовательность команд для модификации описываемого файла.

Каждую такую структуру автомат преобразует в одну вершину графа зависимостей. При этом автомат строит ребра, исходящие из данной вершины и соединяющие ее с вершинами, отвечающими зависимостям. Если вершин, соответствующих каким-либо файлам (описываемому файлу или файлам зависимостей), в графе зависимостей пока нет, то такая вершина добавляется в граф зависимостей.

4.2.2. Схема связей

На рис. 1 приведена схема связей автомата *A0*.

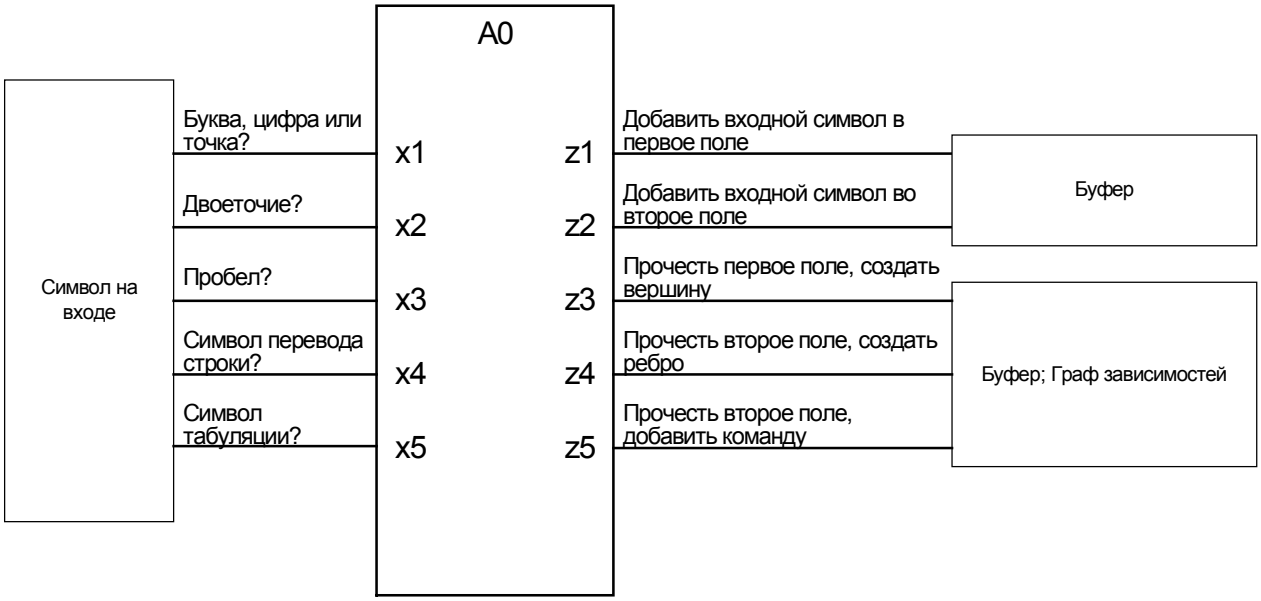

Рис. 1. Схема связей автомата *A0*

4.2.3. Граф переходов

На рис. 2 приведен граф переходов автомата *A0*.

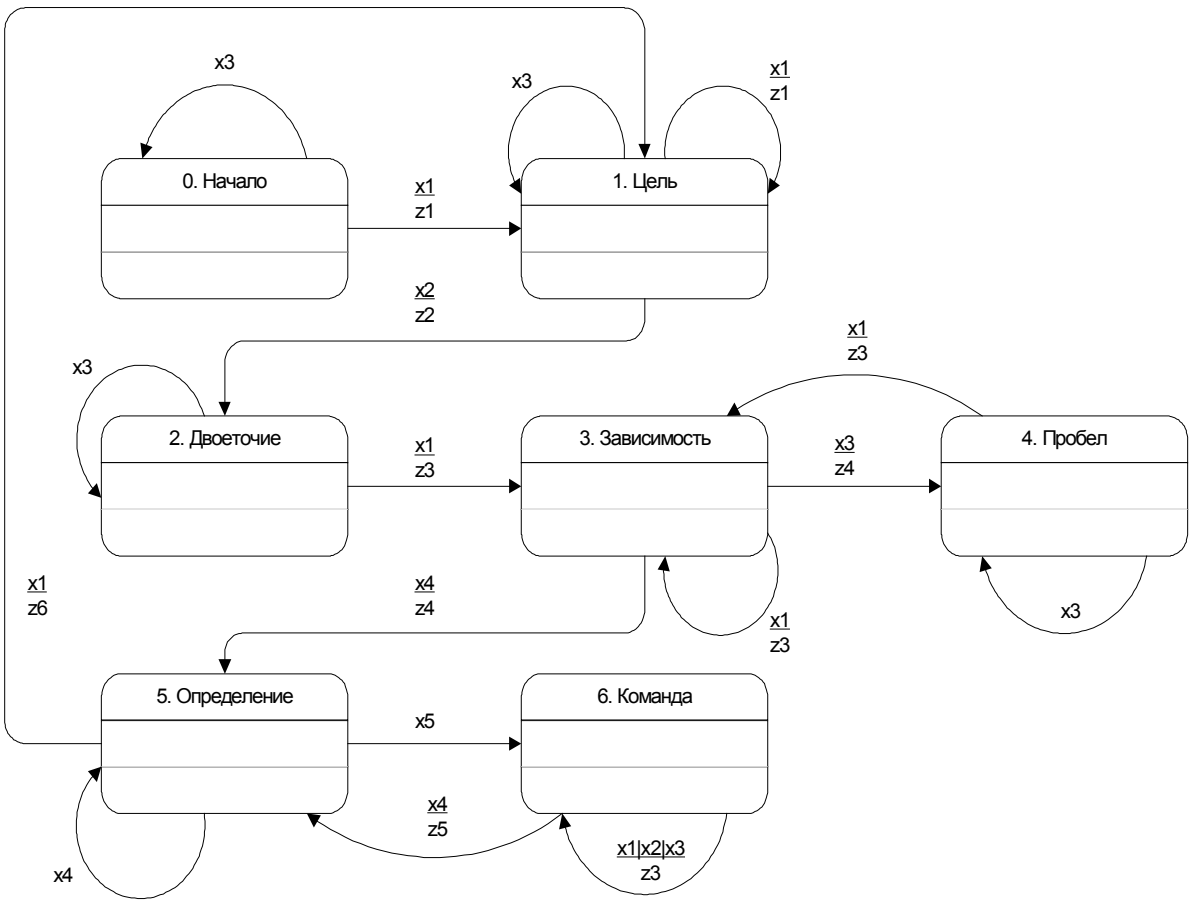

Рис. 2. Граф переходов автомата *A0*

5. Класс *DependencyGraph*

5.1. Словесное описание

Класс *DependencyGraph* создан для того, чтобы экземпляр класса (объект) являлся полем класса *Parser*, описанного выше.

Класс *DependencyGraph* содержит структуры данных, представляющие граф зависимостей, основанные на классе *Vertex*. Методы класса *DependencyGraph* реализуют действия по построению графа зависимостей, такие как:

- добавление вершины в граф;
- проверка на наличие в нем вершины, отвечающей определенному файлу;
- добавление ребра, соединяющего описываемый файл с его зависимостью;
- добавление команды в поле, описывающее вершину, отвечающую описываемому файлу.

Эти методы используются выходными воздействиями автомата *A0*.

После завершения работы автомата *A0* среда исполнения, реализованная в файле `Kmake.cpp`, запускает автомат *A1* с событием *e0*. Автомат *A1* представляет собой метод класса *DependencyGraph*. Класс *DependencyGraph* производит протоколирование состояний автомата *A1*.

5.2. Автомат *A1*

5.2.1. Словесное описание

Автомат *A1* осуществляет обход в глубину по графу зависимостей и располагает вершины в порядке окончания их обработки. Кроме того, он помечает файл, отвечающий вершине, как подлежащий модификации, если хотя бы одному потомку этой вершины соответствует файл, модифицированный или подлежащий модификации. Таким образом, автомат *A1* производит топологическую сортировку вершин графа зависимостей и определяет файлы, подлежащие модификации.

Автомат *A1* использует две вспомогательные структуры – два стека для вершин. Первый из них назовем *стеком вершин*, второй – *стеком родителей*. На схеме связей (рис.3) они обозначены как Стек 1 и Стек 2 соответственно. Второй стек назван стеком родителей потому, что в него помещаются только вершины, имеющие потомков.

Результатом работы автомата *A1* является список команд, для корректного построения приложения с учетом всех изменений (модификаций) исходных файлов, которым отвечают листья графа зависимостей.

5.2.2. Схема связей

Схема связей автомата *A1* приведен на рис.3.

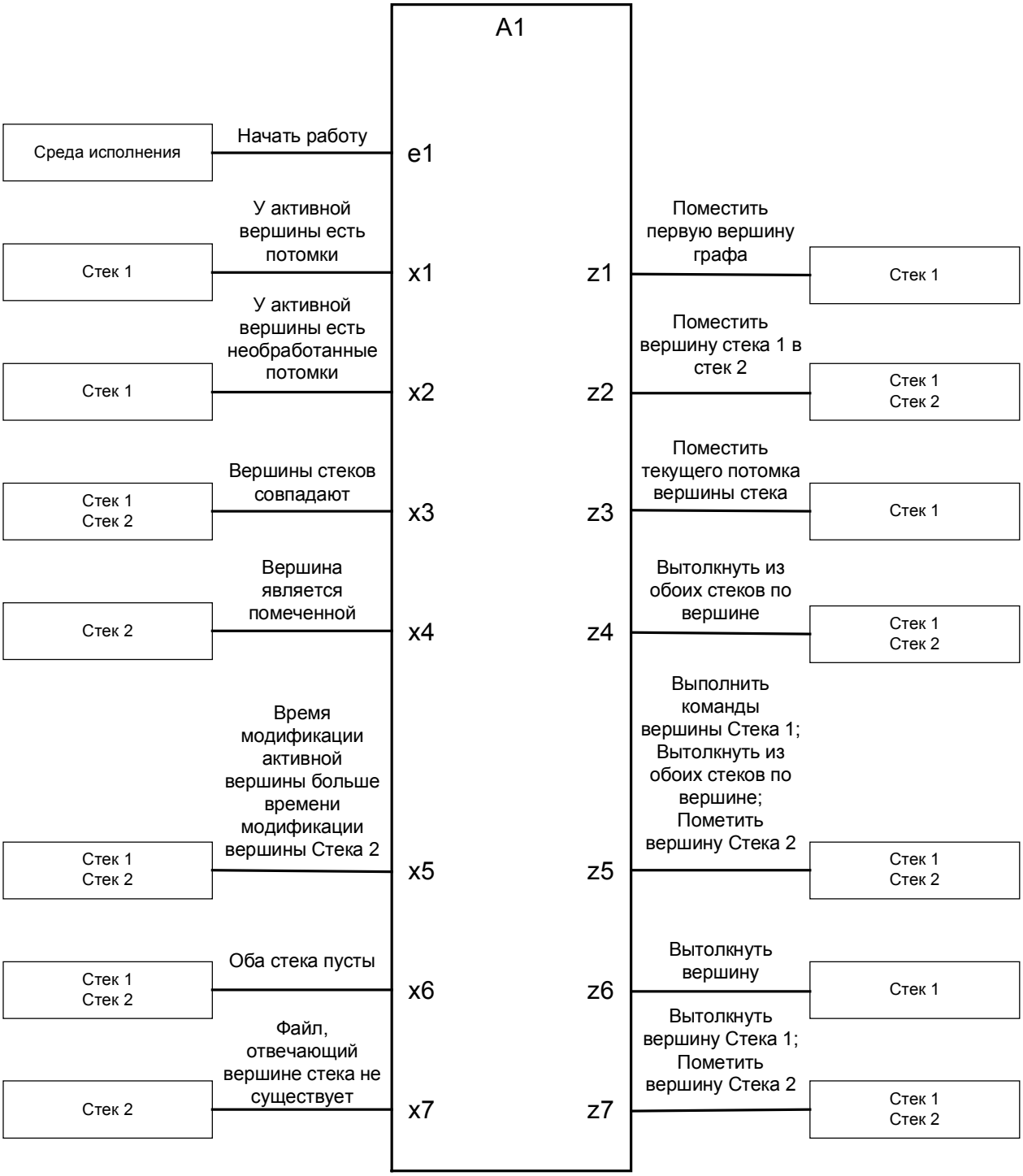

Рис. 3. Схема связей автомата *A1*

5.2.3. Граф переходов

Граф переходов автомата *A1* приведен на рис. 4.

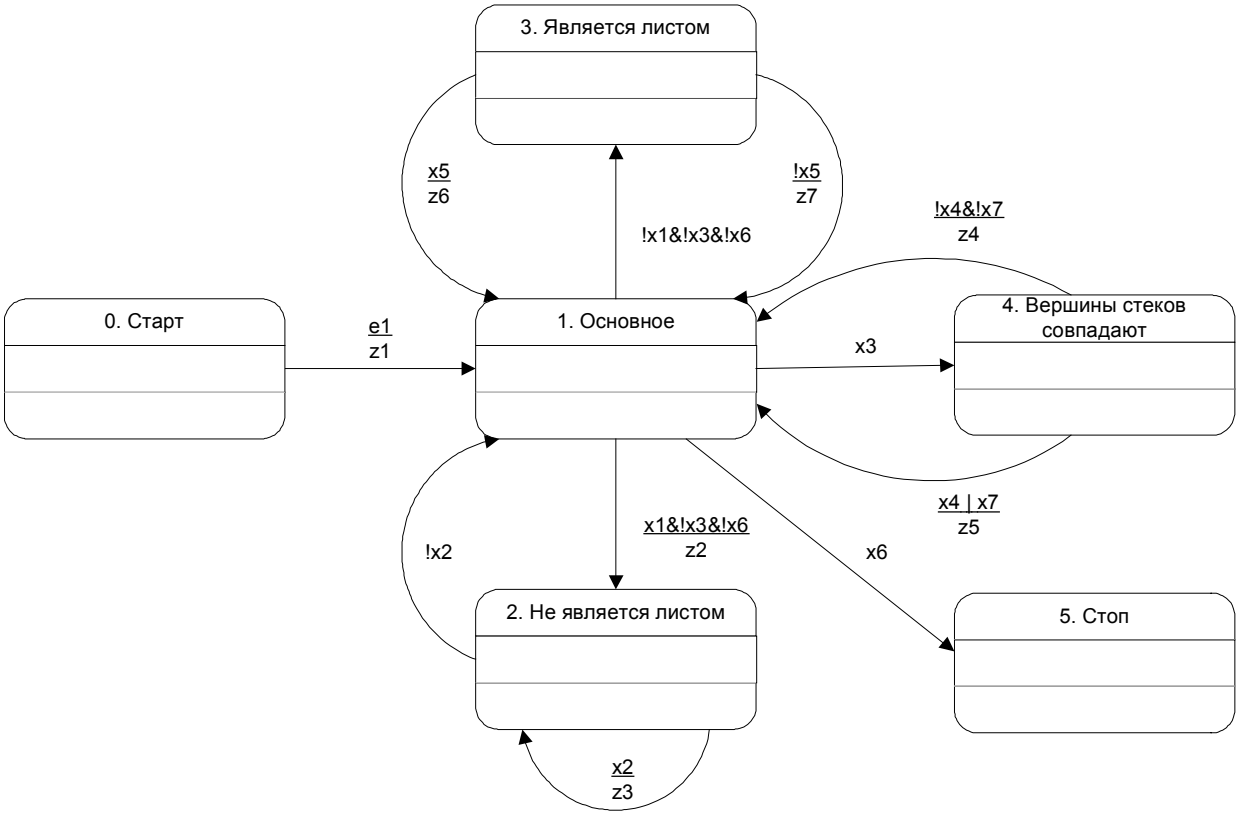

Рис. 4. Граф переходов автомата *A1*

6. Пример работы программы

В качестве примера рассмотрим проект, в котором три файла с исходными текстами:

- `main.c`
- `one.c`
- `two.c`

После компиляции каждого файла получается объектный файл с таким же именем, но с расширением `.obj`. За компиляцией следует сборка файла main.exe из полученных на предыдущем этапе объектных файлов. Пусть файл *makefile* для этого проекта выглядит следующим образом:

```
main.exe: main.obj one.obj two.obj
    bcc32 main.obj one.obj two.obj
```

```
    copy main.exe ..

main.obj: main.c
    bcc32 -c main.c

one.obj: one.c
    bcc32 -c one.c

two.obj: two.c
    bcc32 -c two.c
```

Поясним первую конструкцию:

```
main.exe: main.obj one.obj two.obj
    bcc32 main.obj one.obj two.obj
    copy main.exe ..
```

`main.exe` – описываемый файл-цель. Он должен быть модифицирован, если были внесены изменения хотя бы в один из перечисленных после двоеточия файлов: `main.obj`, `one.obj` или `two.obj`. Его модификация производится выполнением двух команд, каждая из которых расположена в строке, начинающейся с символа табуляции:

```
    bcc32 main.obj one.obj two.obj
    copy main.exe ..
```

Рассмотренный файл *makefile* поступает на вход программы и обрабатывается автоматом *A0* класса *Parser*, который строит граф зависимостей, представленный в виде списка смежности. Начальный фрагмент отладочного протокола работы автомата *A0* приведен в приложении 1.

Построенный граф зависимостей обрабатывается автоматом *A1* класса *DependencyGraph*, который генерирует искомую последовательность команд, требующуюся для построения приложения `main.exe` и записывает ее в файл `todo.bat`. Начальный фрагмент отладочного протокола работы автомата *A1* приведен в приложении 2.

В приложении 3 приведен исходный текст программы на языке *Cu++*.

Заключение

На основе выполненного проекта можно сделать следующие выводы:

- применение SWITCH-технологии позволило **формально реализовать** синтаксический анализатор разбора файлов *Makefile*, не используя классические подходы;
- написанный исходный код понятен, так как структура его основных модулей изоморфна структуре соответствующих графов переходов автоматов;
- разработка утилиты полностью документирована;

В процессе изучения источников автор не обнаружил ссылок на какие-либо стандарты на утилиты *make* (разд. 1).

Источники

1. *Шалыто А.А.*, SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
2. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2001.
3. *Штучкин А.А., Шалыто А.А.* Совместное использование теории построения компиляторов и SWITCH-технологии (на примере построения калькулятора). Проектная документация. http://is.ifmo.ru, раздел «Проекты».
4. *GNU Make Manual*, Free Software Foundation. http://www.gnu.org/software/make/manual/make.html
5. *Туккель Н.И., Шалыто А.А.* Система управления танком для игры Robocode. Объектно-ориентированное программирование с явным выделением состояний. Проектная документация. http://is.ifmo.ru, раздел «Проекты».
6. *Страуструп Б.* Язык программирования C++. М.: Бином, 1999.

Приложение 1. Начальный фрагмент отладочного протокола работы автомата *A0*

```
x1: текущий символ буква, цифра или точка: m
Автомат A0 перешел в состояние GOAL
x1: текущий символ буква, цифра или точка: a
Автомат A0 перешел в состояние GOAL
x1: текущий символ буква, цифра или точка: i
Автомат A0 перешел в состояние GOAL
x1: текущий символ буква, цифра или точка: n
Автомат A0 перешел в состояние GOAL
x1: текущий символ буква, цифра или точка: .
Автомат A0 перешел в состояние GOAL
x1: текущий символ буква, цифра или точка: e
Автомат A0 перешел в состояние GOAL
x1: текущий символ буква, цифра или точка: x
Автомат A0 перешел в состояние GOAL
x1: текущий символ буква, цифра или точка: e
Автомат A0 перешел в состояние GOAL
x2: текущий символ двоеточие: :
Автомат A0 перешел в состояние COLON
x3: текущий символ пробел:
Автомат A0 перешел в состояние COLON
x1: текущий символ буква, цифра или точка: m
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: a
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: i
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: n
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: .
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: o
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: b
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: j
Автомат A0 перешел в состояние DEPENDENCY
x3: текущий символ пробел:
Автомат A0 перешел в состояние SPACE
x1: текущий символ буква, цифра или точка: o
```

```
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: n
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: e
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: .
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: o
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: b
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: j
Автомат A0 перешел в состояние DEPENDENCY
x3: текущий символ пробел:
Автомат A0 перешел в состояние SPACE
x1: текущий символ буква, цифра или точка: t
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: w
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: o
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: .
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: o
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: b
Автомат A0 перешел в состояние DEPENDENCY
x1: текущий символ буква, цифра или точка: j
```

Приложение 2. Начальный фрагмент отладочного протокола работы автомата *A1*

```
e1: Событие e1 произошло
Автомат A1 перешел в состояние MAIN
x1 : у активной вершины есть потомки
!x3: вершины стеков не совпадают
!x6: стек 1 не пусть или стек 2 не пуст
Автомат A1 перешел в состояние NOT_LEAF
x2 : у активной вершины есть необработанные потомки
x2 : у активной вершины есть необработанные потомки
Автомат A1 перешел в состояние NOT_LEAF
x2 : у активной вершины есть необработанные потомки
x2 : у активной вершины есть необработанные потомки
Автомат A1 перешел в состояние NOT_LEAF
x2 : у активной вершины есть необработанные потомки
x2 : у активной вершины есть необработанные потомки
Автомат A1 перешел в состояние NOT_LEAF
!x2: у активной вершины нет необработанных потомков
Автомат A1 перешел в состояние MAIN
x1 : у активной вершины есть потомки
!x3: вершины стеков не совпадают
!x6: стек 1 не пусть или стек 2 не пуст
Автомат A1 перешел в состояние NOT_LEAF
x2 : у активной вершины есть необработанные потомки
x2 : у активной вершины есть необработанные потомки
Автомат A1 перешел в состояние NOT_LEAF
!x2: у активной вершины нет необработанных потомков
Автомат A1 перешел в состояние MAIN
!x1: у активной вершины нет потомков
!x1: у активной вершины нет потомков
!x3: вершины стеков не совпадают
!x6: стек 1 не пусть или стек 2 не пуст
Автомат A1 перешел в состояние LEAF
!x5: время модификации вершины стека 1 меньше или равно
времени модификации вершины стека 2
!x5: время модификации вершины стека 1 меньше или равно
времени модификации вершины стека 2
Автомат A1 перешел в состояние MAIN
x1 : у активной вершины есть потомки
x3 : вершины стеков совпадают
```

```
x1 : у активной вершины есть потомки
x3 : вершины стеков совпадают
Автомат A1 перешел в состояние TOPS
!x4: вершина стека 2 не является помеченной
!x7: файл, соответствующий вершине стека 2 существует
Автомат A1 перешел в состояние MAIN
x1 : у активной вершины есть потомки
!x3: вершины стеков не совпадают
!x6: стек 1 не пусть или стек 2 не пуст
Автомат A1 перешел в состояние NOT_LEAF
x2 : у активной вершины есть необработанные потомки
x2 : у активной вершины есть необработанные потомки
Автомат A1 перешел в состояние NOT_LEAF
!x2: у активной вершины нет необработанных потомков
Автомат A1 перешел в состояние MAIN
!x1: у активной вершины нет потомков
!x1: у активной вершины нет потомков
!x3: вершины стеков не совпадают
!x6: стек 1 не пусть или стек 2 не пуст
Автомат A1 перешел в состояние LEAF
!x5: время модификации вершины стека 1 меньше или равно
времени модификации вершины стека 2
!x5: время модификации вершины стека 1 меньше или равно
времени модификации вершины стека 2
```

Приложение 3. Исходный текст программы

Файл `Buffer.h`. Описание класса буфер.

```
#ifndef __BUFFER__H__
#define __BUFFER__H__

class Buffer
{
private:
    char *f, *d;
    unsigned int lengthF, lengthD;

public:
    Buffer ();
    void addCharF (char c);
    void addCharD (char c);
    char* getStringF ();
    char* getStringD ();
    void clearF ();
    void clearD ();
    ~Buffer ();
};

#endif /* __BUFFER__H__ */
```

Файл `Buffer.cpp`.

```
#include "Buffer.h"

Buffer::Buffer ()
{
    f = new char[256];
    d = new char[256];
    lengthF = 0;
    lengthD = 0;
}

void Buffer::addCharF (char c)
```

```
{
    f[lengthF++] = c;
}

void Buffer::addCharD (char c)
{
    d[lengthD++] = c;
}

char* Buffer::getStringF ()
{
    char *r = new char[lengthF + 1];
    unsigned int i;

    for ( i = 0; i < lengthF; i ++)
        r[i] = f[i];
    r[i] = 0x0;

    return r;
}

char* Buffer::getStringD ()
{
    char *r = new char[lengthD + 1];
    unsigned int i;

    for ( i = 0; i < lengthD; i ++)
        r[i] = d[i];
    r[i] = 0x0;

    return r;
}

void Buffer::clearF ()
{
    lengthF = 0;
}

void Buffer::clearD ()
{
    lengthD = 0;
}
```

```
Buffer::~Buffer ()
{
    delete[] f;
    delete[] d;
}
```

Файл DependencyGraph.h.

```
#ifndef __DEPENDENCY__GRAPH__
#define __DEPENDENCY__GRAPH__

#include <stdio.h>
#include <vector>
#include <stack>
#include "Vertex.h"

enum A1_STATE
{
    START,
    MAIN,
    LEAF,
    NOT_LEAF,
    TOPS,
    STOP
};

class DependencyGraph
{
private:
    A1_STATE state;
    std::stack<Vertex> *stack1, *stack2;

public:
    std::vector<Vertex> *v;
    bool event1;
    FILE *todo;

    DependencyGraph ();
    bool hasVertex (char *filename);
```

```
    void addVertex (char *filename);
    void addDependency (char *filename, char *dependency);
    void addCommand (char *filename, char *command);
    void A1 ();

private:
    void report ();

    bool e1 ();
    bool x1 ();
    bool x2 ();
    bool x3 ();
    bool x4 ();
    bool x5 ();
    bool x6 ();
    bool x7 ();

    void z1 ();
    void z2 ();
    void z3 ();
    void z4 ();
    void z5 ();
    void z6 ();
    void z7 ();

public:
    ~DependencyGraph ();
};

#endif /* __DEPENDENCY__GRAPH__ */
```

Файл DependencyGraph.cpp

```
#include <stdio.h>
#include <string.h>
#include <sys\stat.h>
#include "DependencyGraph.h"
#include "Vertex.h"
#include <vector>
```

```
DependencyGraph::DependencyGraph ()
{
    v = new std::vector<Vertex> ();
    stack1 = new std::stack<Vertex> ();
    stack2 = new std::stack<Vertex> ();
    state = START;
    event1 = false;
}

bool DependencyGraph::hasVertex (char *filename)
{
    unsigned int i;

    if (v -> size() == 0)
    {
        return false;
    }

    for (i = 0; (i < v -> size ()) && (strcmp(filename,
((*v)[i]).fname -> data() ) != 0); i ++);

    if (i == v -> size ())
        return false;

    return true;
}

void DependencyGraph::addVertex (char *filename)
{
    if (!(this -> hasVertex (filename)))
    {
        Vertex *newVertex = new Vertex (filename);
        v -> push_back (*newVertex);
    }
}

void DependencyGraph::addDependency (char *filename, char
*dependency)
{
    unsigned int i, j;

    this -> addVertex (filename);
```

```
    this -> addVertex (dependency);

    for (i = 0; (i < v -> size ()) && (strcmp (filename,
((*v)[i]).fname -> data()) != 0); i ++);

    for (j = 0; (j < v -> size ()) && (strcmp (dependency,
((*v)[j]).fname -> data()) != 0); j ++);

    ((*v)[i]).addEdge (j);
}

void DependencyGraph::addCommand (char *filename, char
*command)
{
    unsigned int i;

    for (i = 0; (i < v -> size ()) && (strcmp (filename,
((*v)[i]).fname -> data())); i ++);

    ((*v)[i]).addCommand (command);
}
```

// Реализация автомата A1

```
void DependencyGraph::A1 ()
{
    for ( ; state != STOP; )
    {
        switch (state)
        {
            case START :
                if (e1 ())
                {
                    z1 ();
                    state = MAIN;
                }
                break;
            case MAIN :
                if ((x1()) && (!x3()) && (!x6()))
                {
                    z2 ();
                    state = NOT_LEAF;
```

```
        }
        else
            if ((!x1()) && (!x3()) && (!x6()))
            {
                                state = LEAF;
            }
            else
                if (x3())
                {
                                state = TOPS;
                }
                else
                    if (x6())
                    {
                                state = STOP;
                    }
        break;
    case LEAF :
        if (x5())
        {
             z7 ();
            state = MAIN;
        }
        else
            if (!x5())
            {
                z6 ();
                state = MAIN;
            }
        break;
    case NOT_LEAF :
        if (!x2())
        {
            state = MAIN;
        }
        else
            if (x2())
            {
                z3 ();
                state = NOT_LEAF;
            }
        break;
```

```
            case TOPS :
                if ((!x4()) && (!x7()))
                {
                    z4 ();
                    state = MAIN;
                }
                else
                    if (x4() || x7())
                    {
                        z5 ();
                        state = MAIN;
                    }
                break;
            case STOP :
                break;
        }
        report ();
    }
    printf ("Автомат A1 закончил работу\n");
}

bool DependencyGraph::e1 ()
{
    if (event1)
            printf ("e1: Событие e1 произошло\n");

        return event1;
}
```

// Входные переменные автомата A1

```
bool DependencyGraph::x1 ()
{
    if (stack1 -> size () == 0)
        return false;

    if (((stack1 -> top ()).edges) -> size () > 0)
        printf ("x1 : у активной вершины есть потомки\n");
    else
        printf ("!x1: у активной вершины нет потомков\n");

    return (((stack1 -> top ()).edges) -> size () > 0);
```

```
}

bool DependencyGraph::x2 ()
{
    if ((stack2 -> top ()).count < ((stack2 -> top
()).edges) -> size ())
        printf ("x2 : у активной вершины есть
необработанные потомки\n");
    else
        printf ("!x2: у активной вершины нет
необработанных потомков\n");

    return ((stack2 -> top ()).count < ((stack2 -> top
()).edges) -> size ());
}

bool DependencyGraph::x3 ()
{
    if ((stack1 -> size () == 0) || (stack2 -> size () ==
0))
    {
        printf ("!x3: вершины стеков не совпадают\n");
        return false;
        }

    if (!strcmp ((stack1 -> top ()).fname -> data(),
(stack2 -> top ()).fname -> data()))
        printf ("x3 : вершины стеков совпадают\n");
    else
        printf ("!x3: вершины стеков не совпадают\n");

    return (!strcmp ((stack1 -> top ()).fname -> data(),
(stack2 -> top ()).fname -> data()));
}

bool DependencyGraph::x4 ()
{
    if ((stack2 -> top ()).marked)
        printf ("x4 : вершина стека 2 является
помеченной\n");
    else
```

```
        printf ("!x4: вершина стека 2 не является
помеченной\n");


    return (stack2 -> top ()).marked;
}

bool DependencyGraph::x5 ()
{
    struct stat s;
    const char *f1, *f2;
    _TIME_T t1, t2;

    f1 = ((stack1 -> top()).fname) -> c_str();
    stat (f1, &s);
    t1 = s.st_mtime;

    f2 = ((stack2 -> top()).fname) -> c_str();
    stat (f2, &s);
    t2 = s.st_mtime;

    if (t1 > t2)
        printf ("x5 : время модификации вершины стека 1
больше времени модификации вершины стека 2\n");
    else
        printf ("!x5: время модификации вершины стека 1
меньше или равно времени модификации вершины стека 2\n");

    return (t1 > t2);
}

bool DependencyGraph::x6 ()
{
    if ((stack1 -> empty ()) && (stack2 -> empty ()))
        printf ("x6 : стек 1 и стек 2 пусты\n");
    else
        printf ("!x6: стек 1 не пусть или стек 2 не
пуст\n");


    return (stack1 -> empty ()) && (stack2 -> empty ());
}
```

```
bool DependencyGraph::x7 ()
{
    struct stat s;
    _TIME_T t;

    stat (((stack2 -> top()).fname) -> c_str(), &s);
    t = s.st_mtime;

    if (t == 0)
        printf ("x7 :файл, соответствующий вершине стека 2
не существует\n");
    else
        printf ("!x7:файл, соответствующий вершине стека 2
существует\n");

    return (t == 0);
}
```

// Выходные воздействия автомата A1

```
void DependencyGraph::z1 ()
{
    stack1 -> push ((*v)[0]);
}

void DependencyGraph::z2 ()
{
    Vertex& refVertex = stack1 -> top();
    Vertex *copyVertex = new Vertex (refVertex);
    stack2 -> push (*copyVertex);
}

void DependencyGraph::z3 ()
{
    int nVertexIndex = ((stack2 -> top ()).count)++;
    int nVertexNumber = (*((stack2 -> top
()).edges))[nVertexIndex];
    stack1 -> push ((*v)[nVertexNumber]);
}

void DependencyGraph::z4 ()
```

```
{
    stack1 -> pop ();
    stack2 -> pop ();
}

void DependencyGraph::z5 ()
{
    for ( unsigned int i = 0; i < ((stack1 -> top
()).commands) -> size(); i ++)
        fprintf (todo, "%s\n", ((*((stack1 -> top
()).commands)) [i]).c_str());

    stack2 -> pop ();
    stack1 -> pop ();

    if (!(stack2 -> empty()))
        (stack2 -> top ()).marked = true;
}

void DependencyGraph::z6 ()
{
    stack1 -> pop ();
}

void DependencyGraph::z7 ()
{
    stack1 -> pop ();

    (stack2 -> top ()).marked = true;
}

void DependencyGraph::report ()
{
    char s[20];
    switch (state)
    {
        case START: strcpy (s, "START"); break;
        case MAIN: strcpy (s, "MAIN"); break;
        case LEAF: strcpy (s, "LEAF"); break;
        case NOT_LEAF: strcpy (s, "NOT_LEAF"); break;
        case TOPS: strcpy (s, "TOPS"); break;
        case STOP: strcpy (s, "STOP"); break;
```

```
    }

    printf ("Автомат A1 перешел в состояние %s\n", s);
}

DependencyGraph::~DependencyGraph ()
{
    delete v;
    delete stack1;
    delete stack2;
}
```

Файл Kmake.cpp

```
#include "Parser.h"

int main (int argc, char **argv)
{
    FILE *f;
    Parser *parser;

    if (argc != 2)
    {
        fprintf (stdout, "usage:\n\t%s <makefile>\n",
argv[0]);
        return 0;
    }

    parser = new Parser ();
    f = fopen (argv[1], "r");
    parser -> A0 (f);
    fclose (f);

    f = fopen ("todo.bat", "w");
    (parser -> graph) -> todo = f;
    (parser -> graph) -> event1 = true;
    (parser -> graph) -> A1();
    fclose (f);

    delete parser;
```

```
}
```

Файл Parser.h

```
#ifndef __PARSER__H__
#define __PARSER__H__

#include <stdio.h>
#include "Buffer.h"
#include "DependencyGraph.h"

enum A0_STATE
{
    LINE,
    GOAL,
    COLON,
    DEPENDENCY,
    SPACE,
    DETERMINE,
    COMMAND
};

class Parser
{
private:
    char c;
    A0_STATE state;
    Buffer *buffer;

public:
    DependencyGraph *graph;
    Parser ();
    void A0 (FILE *f);

private:
    void report ();

    bool x1 ();
    bool x2 ();
    bool x3 ();
    bool x4 ();
    bool x5 ();
```

```
    void z1 ();
    void z2 ();
    void z3 ();
    void z4 ();
    void z5 ();
    void z6 ();
public:
    ~Parser();
};

#endif /* __PARSER__H__ */
```

Файл Parser.cpp

```
#include "Parser.h"
#include <ctype.h>

Parser::Parser ()
{
    buffer = new Buffer ();
    graph = new DependencyGraph ();
    state = LINE;
}
```

// Реализация автомата A0

```
void Parser::A0 (FILE *f)
{
    for ( ; fscanf (f, "%c", &c) == 1; )
    {
        switch (state)
        {
            case LINE:
                if (x1())
                {
                    z1();
                    state = GOAL;
                }
                if (x3())
```

```
    {

    }
    break;
case GOAL:
    if (x1())
        z1();

    if (x2())
    {
        z3();
        state = COLON;
    }

    if (x3())
    {

    }

    break;
case COLON:
    if (x1())
    {
        z2();
        state = DEPENDENCY;
    }
    if (x3())
    {

    }
    break;
case DEPENDENCY:
    if (x1())
        z2();

    if (x3())
    {
        z4();
        state = SPACE;
    }

    if (x4())
```

```
        {
            z4();

            state = DETERMINE;
        }
        break;

    case SPACE:
        if (x1())
        {
            z2();
            state = DEPENDENCY;
        }
        if (x3())
        {

        }
        break;
    case DETERMINE:
        if (x1())
        {
            z6 ();
            state = GOAL;
        }
        if (x5())
            state = COMMAND;

        if (x4())
        {

        }

        break;
    case COMMAND:
        if (x1() || x2() || x3())
        {
            z2 ();
        }
        if (x4())
        {
            z5 ();
            state = DETERMINE;
```

```
                }
                break;
        }
        report ();
    }
    printf ("Автомат A0 закончил работу\n");
}

void Parser::report ()
{
    char s[20];
    switch (state)
    {
        case LINE: strcpy (s, "LINE"); break;
        case GOAL: strcpy (s, "GOAL"); break;
        case COLON: strcpy (s, "COLON"); break;
        case DEPENDENCY: strcpy (s, "DEPENDENCY"); break;
        case SPACE: strcpy (s, "SPACE"); break;
        case DETERMINE: strcpy (s, "DETERMINE"); break;
        case COMMAND: strcpy (s, "COMMAND"); break;
    }

    printf ("Автомат A0 перешел в состояние %s\n", s);
}
```

// Входные переменные автомата A0

```
bool Parser::x1 ()
{
    if ((isalnum (c)) || (c == '.') || (c == '-') || (c ==
'+'))
    {
        printf ("x1: текущий символ буква, цифра или
точка: %c\n", c);
        return true;
    }
    return false;
}

bool Parser::x2 ()
{
    if (c == ':')
```

```
    {
        printf ("x2: текущий символ двоеточие: %c\n", c);
        return true;
    }
    return false;
}

bool Parser::x3 ()
{
    if (c == ' ')
    {
        printf ("x3: текущий символ пробел: %c\n", c);
        return true;
    }
    return false;
}

bool Parser::x4 ()
{
    if ((c == 0xd) || (c == 0xa))
    {
        printf ("x4: текущий символ - символ перевода
строки: %c\n", c);
        return true;
    }
    return false;
}

bool Parser::x5 ()
{
    if (c == '\t')
    {
        printf ("x5: текущий символ - символ табуляции:
%c\n", c);
        return true;
    }
    return false;
}
```

// Выходные воздействия автомата A0

```
void Parser::z1 ()
```

```
{
    buffer -> addCharF (c);
}

void Parser::z2 ()
{
    buffer -> addCharD (c);
}

void Parser::z3 ()
{
    char *str = buffer -> getStringF ();

    graph -> addVertex (str);

    delete[] str;
}

void Parser::z4 ()
{
    char *str1 = buffer -> getStringF (),
         *str2 = buffer -> getStringD ();

    graph -> addDependency (str1, str2);

    buffer -> clearD ();

    delete[] str1;
    delete[] str2;
}

void Parser::z5 ()
{
    char *str1 = buffer -> getStringF (),
         *str2 = buffer -> getStringD ();

    graph -> addCommand (str1, str2);

    buffer -> clearD ();

    delete[] str1;
    delete[] str2;
```

```
}

void Parser::z6 ()
{
    buffer -> clearF ();
    buffer -> clearD ();

    buffer -> addCharF (c);
}

Parser::~Parser ()
{
    delete buffer;
    delete graph;
}
```

Файл Vertex.h

```
#ifndef __VERTEX__H__
#define __VERTEX__H__

#include <vector>
#include <string>

class Vertex
{
public:
    std::vector<std::string> *commands;
    std::string *fname;
    std::vector<int> *edges;
    unsigned int count;
    bool marked;

    Vertex (const char *filename);
    Vertex (const Vertex& v);
    void addEdge (int n);
    void addCommand (char *command);
    ~Vertex ();
};
```

```
#endif /* __VERTEX__H__ */
```

Файл Vertex.cpp

```
#include "Vertex.h"
#include <vector>
#include <string>

Vertex::Vertex (const char *filename)
{
    edges = new std::vector<int>;
    fname = new std::string(filename);
    commands = new std::vector<std::string>;
    count = 0;
    marked = false;
}

Vertex::Vertex (const Vertex &v)
{
    edges = new std::vector<int> (*(v.edges));
    fname = new std::string(*(v.fname));
    commands = new std::vector<std::string>(*(v.commands));
    count = 0;
    marked = false;
}

void Vertex::addEdge (int n)
{
    edges -> push_back (n);
}

void Vertex::addCommand (char *command)
{
    std::string *commandString = new std::string (command);
    commands -> insert (commands -> end (),
*commandString);
}

Vertex::~Vertex ()
{
    delete edges;
    delete fname;
```

```
    delete commands;
}
```