1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**КУРСОВАЯ РАБОТА**

**Разработка программы-парсера для языка утилиты Make**

по дисциплине «Формальные грамматики и теория компиляторов»

1. Выполнили Улькин А.А.
2. студенты гр. 4851003/90801 Кулеева А.Г.



1. Руководитель Семьянов П.В.
2. ст. преподаватель

«\_\_» \_\_\_\_\_\_\_\_\_\_ 2022 г.

1. Санкт-Петербург
2. 2022

Содержание

|  |  |  |
| --- | --- | --- |
| Цель и задачи ............................................................................... | | 3 |
| 1 | Теоретическая часть ............................................................ | 4 |
| 1.1 | Особенности языка .............................................................. | 4 |
| 2 | Практическая часть ............................................................. | 5 |
| 2.1 | Разработка лексического анализатора языка .................... | 5 |
| 2.2 | Разработка синтаксического анализатора языка .............. | 9 |
| 2.3 | Тестирование ....................................................................... | 12 |
|  | [Р](https://docs.google.com/document/d/1aq0ZjT0QTs0RoOTwSiIlp-OHbh3CBBfx/edit#heading=h.1t3h5sf) |  |
| Заключение.................................................................................... | | 15 |
| Список использованных источников.......................................... | | 56 |

ЦЕЛИ

Главной целью данной курсовой работы является разработка распознавателя современного формального языка, используемого для работы с утилитой ОС Linux Make. Разработка осуществляется с помощью генератора синтаксических анализаторов YACC, генератора лексических анализаторов Flex и языка C. Разработанный анализатор должен предупреждать о наличии базовых синтаксических и лексических ошибок в реализованном программистом makefile.

Второй, но не менее важной целью, является исследование возможности построения распознавателя современного формального языка с использованием средств YACC и Flex.

ЗАДАЧИ

Для достижения поставленных целей необходимо:

1. изучить основы написания makefile: синтаксис, спецификацию;
2. разработать программу для Flex, с целью создания лексического анализатора для грамматики;
3. разработать грамматику для генератора синтаксических анализаторов YACC;
4. собрать и протестировать полученный распознаватель;
5. сделать вывод об актуальности YACC и Flex.

# Теоретическая часть

## Особенности языка

Makefile это файл, используемый утилитой make для более простой и быстрой компиляции программ. Стандартный файл состоит из “правил” со следующей формой:

target … : prerequisites …

recipe

…

…

*Целью* обычно является имя файла, который генерируется программой; примерами целей являются исполняемые или объектные файлы. Целью также может быть имя выполняемого действия, например clean, all и т.д.

Обязательным *условием* является файл, который используется в качестве входных данных для создания целевого объекта. Цель часто зависит от нескольких файлов, поэтому также условия могут называться *зависимостями*.

*Рецепт* — это действие, которое осуществляется утилитой make. Рецепт может содержать более одной команды, либо в одной строке, либо каждая в своей отдельной строке. Важно всегда ставить символ табуляции в начале каждой строки рецепта!

Обычно рецепт находится в правиле с предварительными условиями и служит для создания целевого файла, если какие-либо из предварительных условий изменяются. Однако правило, которое определяет рецепт для целевого объекта, не обязательно должно иметь предварительные условия. Например, правило, содержащее команду удаления, связанную с целью clean, не имеет предварительных условий.

Таким образом, *правило* объясняет, как и когда изменять определенные файлы, которые являются целями конкретного правила. Make выполняет рецепт для предварительных условий для создания или обновления целевого объекта. Правило также может объяснить, как и когда выполнять действие.

# Практическая часть

## Разработка лексического анализатора языка

Был разработан лексический анализатор с помощью flex. Разберем его код. Зададим определения чисел и имён. При этом имя переменной может содержать, буквы и цифры, а также имя обрамляется скобками и символом «$» в начале. За все это отвечает переменная VAR\_REF. Переменные RUN\_CMD и CMD\_S отвечают за аргументы командной строки, которые могут содержаться в файле. Их различие состоит в том, что RUN\_CMD ориентирована на самописные скрипты, которые запускаются с помощью ./script, а CMD\_S на встроенные в систему скрипты типа /bin/bash и им подобным. Они задаются регулярными выражениями.

DIGIT [0-9]

UNIT\_NAME [\_a-zA-Z\\*\-\+\@\?]

VAR\_REF "$""("({UNIT\_NAME}|{DIGIT})+")"

RUN\_CMD ^(\@[^\@]\*\@)?\t\.\/[^\n]\*

CMD\_S   ^(\@[^\@]\*\@)?\t\*[^\n]\*

ANY\_EQ [":"|"!"|"?"|"+"|"::"]?=[^\n]\*

Следующая конструкция определяет цель и зависимости. Поскольку зависимостей может быть несколько, то иногда их указывают в новой строке через символ «\», например:

Цель: зависимость1 \

Зависимость2 \

Зависимость3

Хотя такое определение в документации отсутствует, однако оно не запрещено и всё-таки используется на практике, поэтому было принято решение добавить обработку данной конструкции.

"\\\n"[ \t]\*   {yylval.ptrchar = yytext; return TARGET\_SLASH;}

Также выяснилось, что для объявления пустой цели, которая ничего не делает, используется символ «;», его обработка также добавлена. Пример: target:;

Поскольку зависимости — это файлы, то необходимо добавить их распознавание как по имени, так и по пути.

({UNIT\_NAME}|{DIGIT})+        {yylval.ptrchar = yytext; return UNIT\_NAME;}

({UNIT\_NAME}|{DIGIT})+([\.]({UNIT\_NAME}|{DIGIT})+)+   { yylval.ptrchar = yytext; return FILE\_NAME; }

(\/)?(({VAR\_REF}|{UNIT\_NAME}|{DIGIT}|[\.]|[\.\.])+(\/)?)+ { yylval.ptrchar = yytext; return FILE\_PATH; }

\%({UNIT\_NAME}|{DIGIT}|[\.])\*           { yylval.ptrchar = yytext; return TEMPLATE;}

Также было добавлено распознавание не только команд консоли, но и цельных строк, которые должны быть заключены в кавычки, чтобы считаться полноценной строкой, а не набором отдельных команд. Пример такой строки:  “echo "aaaaa" > /dev/null”.

"\'".\*"\'"     {yylval.ptrchar = yytext; return ANY\_STRING;}

"\`".\*"\`"     {yylval.ptrchar = yytext; return ANY\_STRING;}

"\"".\*"\""     {yylval.ptrchar = yytext; return ANY\_STRING;}

Далее следует обработка команд cmd, которые выступают в качестве рецептов. Здесь по сути может быть любая последовательность символов.

{RUN\_CMD}      {yylval.ptrchar = yytext; return RUN\_COMMAND;}

{CMD\_S}        {yylval.ptrchar = yytext; return COMMAND;}

Комментарии в makefile пишутся с помощью символа «#».

"#"[^\n]\*[\n]?

Была реализована обработка условных операторов и директивы define-endef.

"ifeq "      { return IFEQ;}

"ifneq "     { return IFNEQ;}

"ifdef "     { return IFDEF;}

"ifndef "    { return IFNDEF;}

"else"[ ]?      { return ELSE;}

"endif"     { return ENDIF;}

"define" {yylval.ptrchar = yytext; return DEFINE;}

"endef" {yylval.ptrchar = yytext; return ENDEF;}

Также была реализована обработка специальных слов.

".PHONY"                    { return SPECIAL;}

".SUFFIXES"                 { return SPECIAL;}

".DEFAULT"                  { return SPECIAL;}

".PRECIOUS"                 { return SPECIAL;}

".INTERMEDIATE"             { return SPECIAL;}

".SECONDARY"                { return SPECIAL;}

".DELETE\_ON\_ERROR"          { return SPECIAL;}

".IGNORE"                   { return SPECIAL;}

".SILENT"                   { return SPECIAL;}

".EXPORT\_ALL\_VARIABLES"     { return SPECIAL;}

".NOTPARALLEL"              { return SPECIAL;}

"include"   { return INCLUDE;}

"export"   {yylval.ptrchar = yytext; return EXPORT;}

Кроме того, в makefile есть некоторые свои встроенные функции, каждая из которых обладает собственным синтаксисом. Аргументы функции могут передаваться напрямую (1) или через переменные (2):

$(subst ee,EE,feet on the street)

$(subst $(space),$(comma),$(foo))

Поскольку внутри функций можно ссылаться на переменные и другие функции, оборачиваемые скобками, то такие конструкции было решено обрабатывать следующим образом: распознается начало функции, вызывается функция, которая из stdin посимвольно считывает входную последовательность, проверяя скобки. Если скобки считаются закрытыми, то мы выходим из функции, тем самым обозначая конец конструкции.

$(if condition,then-part[,else-part])

$(or condition1[,condition2[,condition3…]])

$(and condition1[,condition2[,condition3…]])

Поскольку некоторые параметры данных функций могут быть опциональными, они также укладываются в регулярное выражение .\*, и их реализация также ложится на плечи программиста.

Использование функций ограничивается жесткими правилами, поэтому пользователь должен также самостоятельно проверять корректность вводимых аругментов. Разработанный в данной работе анализатор не берет на себя такую ответственность, он служит лишь помощником с точки зрения синтаксиса и лексики файла.

"$"[(\{]"abspath " { yylval.ptrchar = yytext; check\_brackets(yylval.ptrchar); return FUNCTION;}

"$"[(\{]"addsuffix " { yylval.ptrchar = yytext; check\_brackets(yylval.ptrchar); return FUNCTION;}

"$"[(\{]"addprefix " { yylval.ptrchar = yytext; check\_brackets(yylval.ptrchar); return FUNCTION;}

…

При нахождении любого другого символа или слова, которые не описаны, будет возвращена ошибка «Неизвестный символ».

.         {printf("\n%c\n", \*yytext); yyerror("Unknown symbol");}

## Разработка синтаксического анализатора

Рассмотрим код синтаксического анализатора. В качестве входного потока может поступить либо строка, либо несколько строк.

%%

input: line

     | input line

;

Строка представляет из себя либо пустую строку, либо цель, либо инициализацию переменных, либо инклуды. Функция может быть вызвана внутри таргета, а может быть и вне его. Она может быть присвоена переменной, а может просто выполнить какое-то действие и не вернуть ничего.

line: EOL

   | COMMAND

   | target

   | variable\_init

   | condition EOL

   | FUNCTION EOL

   | INCLUDE spec\_incs EOL

   | SPECIAL COLON spec\_incs EOL

   | SPECIAL COLON EOL

   | DEFINE variable\_init ENDEF EOL

   | EXPORT spec\_incs EOL

   | UNEXPORT spec\_incs EOL

   | EXPORT variable\_init

   | UNEXPORT variable\_init

   | target\_names COLON ';' EOL

   | target\_names { warning("Lonely units"); }

Из чего может состоять цель? Определим переменную target\_names, которая будет рассмотрена позже. После цели всегда стоит двоеточие. Наличие пререквизитов и рецептов не обязательно, поэтому рассматриваются различные вариации. Также возможно присутствие специальных слов.

target: target\_names COLON EOL recipes

| target\_names COLON target\_names EOL recipes

| target\_names COLON target\_names EOL

| target\_names COLON ';' EOL

;

Теперь рассмотрим упомянутую выше переменную. Она может состоять из одиночного имени или из нескольких имен.

target\_names:

target\_name

| target\_names target\_name

| target\_names TARGET\_SLASH target\_name

;

Одиночное имя может включать в себя переменную, имя файла, путь до файла и любые другие строки, которые не относятся к вышеперечисленномy.

target\_name:

VAR\_REF

| UNIT\_NAME

| FILE\_PATH

| FILE\_NAME

| TEMPLATE

| '('TEMPLATE')'

;

Имя переменной состоит из самого имени, некоторого присваивания и присваиваемого значения, которого может и не быть. Если же оно есть, то оно может быть также представлено командой или функцией.

variable\_inits:

   variable\_init EOL

   | FUNCTION

   | target

   | EOL

   | condition

   | COMMAND EOL       { if (!in\_target) yyerror("command out of target"); command\_count++; }

   | INCLUDE spec\_incs EOL

   | SPECIAL COLON spec\_incs EOL

   | SPECIAL COLON EOL

   | DEFINE variable\_init ENDEF EOL

   | EXPORT spec\_incs EOL

   | UNEXPORT spec\_incs EOL

   | EXPORT variable\_init

   | UNEXPORT variable\_init

   | variable\_inits variable\_init EOL

   | variable\_inits COMMAND EOL { if (!in\_target) yyerror("command out of target"); command\_count++; }

   | variable\_inits condition EOL

   | variable\_inits target

   | variable\_inits EOL

   | variable\_inits INCLUDE spec\_incs EOL

   | variable\_inits SPECIAL COLON spec\_incs EOL

   | variable\_inits SPECIAL COLON EOL

   | variable\_inits DEFINE variable\_init ENDEF EOL

   | variable\_inits EXPORT spec\_incs EOL

   | variable\_inits UNEXPORT spec\_incs EOL

   | variable\_inits EXPORT variable\_init

   | variable\_inits UNEXPORT variable\_init

   | variable\_inits FUNCTION

variable\_init:

UNIT\_NAME ANY\_EQ

;

В самом конце файла содержатся рецепты, которые должны завершаться символом конца строки.

recipes:

  recipe EOL

| recipes recipe EOL

;

Рецепты состоят из команд, которые были обнаружены при лексическом анализе.

recipe:

  RUN\_COMMAND

| COMMAND

| condition

;

Также была добавлена обработка условных операторов. Сами условные конструкции выглядят следующим образом: обязательно наличие if, условия и завершающего endif. Возможно также наличие else, при этом такие конструкции могут быть рекурсивными. Для этого была реализована конструкция post\_condition.

condition:

if cond\_construction EOL post\_condition

| ifdef cond EOL ENDIF

| ifdef cond EOL ELSE EOL ENDIF

| ifdef cond EOL variable\_inits ENDIF

| ifdef cond EOL ELSE EOL variable\_inits ENDIF

| ifdef cond EOL variable\_inits ELSE EOL ENDIF

| ifdef cond EOL variable\_inits ELSE EOL variable\_inits ENDIF

В данной конструкции рассмотрены всевозможные варианты использования else.

post\_condition:

  ENDIF

| elses ENDIF

| elses ELSE EOL ENDIF

| elses ELSE EOL variable\_inits ENDIF

| ELSE EOL ENDIF

| ELSE EOL variable\_inits ENDIF

| variable\_inits ENDIF

| variable\_inits elses ENDIF

| variable\_inits elses ELSE EOL ENDIF

| variable\_inits elses ELSE EOL variable\_inits ENDIF

| variable\_inits ELSE EOL variable\_inits ENDIF

| variable\_inits ELSE EOL ENDIF

;

Также была создана отдельная конструкция для рекурсивного распознавания “иначе”.

elses:

ELSE cond\_construction EOL variable\_inits

| ELSE cond\_construction EOL

| elses ELSE cond\_construction EOL variable\_inits

| elses ELSE cond\_construction EOL

;

Под cond\_construction понимается цельная конструкция, которая может быть использована ifeq или ifneq.

cond\_construction:

'(' cond ',' cond ')'      {if (full\_info) printf("Condition at line %i\n", lines\_count);}

| '(' ',' cond ')'        {if (full\_info) printf("Condition at line %i\n", lines\_count);}

| '(' cond ',' ')'        {if (full\_info) printf("Condition at line %i\n", lines\_count);}

| '(' ',' ')'            {if (full\_info) printf("Condition at line %i\n", lines\_count);}

| ANY\_STRING ANY\_STRING  {if (full\_info) printf("Condition at line %i\n", lines\_count);}

;

В само слово cond вложен следующий смысл:

cond:

UNIT\_NAME    {  }

| FILE\_PATH   { }

| FILE\_NAME   { }

| VAR\_REF     { }

| ANY\_STRING  { }

| FUNCTION    { }

| SHELL\_COMMAND {}

;

Как видно выше, используется только одна возможная грамматика для  ifeq и ifneq. Данные слова соответствуют операторам если равно и если не равно. Слова  ifdef или ifndef служат для получения информации о том, определена ли переменная или нет соответственно.

if:

      IFEQ

    | IFNEQ

    ;

ifdef:

      IFDEF

    | IFNDEF

    ;

## Тестирование

Вся программа собирается при помощи следующих команд:

win\_bison.exe -dt makeg.y

win\_flex.exe terminals.l

gcc -w main.c makeg.tab.c lex.yy.c -o maker

Для ОС Windows используется bat файл, для ОС Linux – makefile.

На вход программе подается путь до анализируемого файла и режим работы. Реализовано три режима работы программы:

* 0 или без аргумента: выводится статистика файла (Рисунок 1);
* 1: выводится информация из yydebug;
* 2: выводится статистика и локация частей файла по строкам, а именно переменные, цели, команды, условия и комментарии (Рисунок 2).

В случае ошибок программа выведет ошибку, её смысл и номер строки (Рисунок 3).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1 – Режим работы 0

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 2 – Режим работы 2

Также хотелось бы обратить внимание на некоторые тесты. Были отдельно реализованы тесты для условных конструкций (Рисунок 3) и функций. Тест bigvim это файл на 4450 строк. Также была реализована проверка на 12 тестах openssl. Ещё несколько файлов были найдены на просторах интернета.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3 – Тестирование условных конструкций

В случае ошибок программа выведет ошибку, её смысл и номер строки (Рисунок 4). Рассмотрим пример с неверным использованием скобок: в строке 5 поставим больше скобок, чем требуется. Аналогичный результат, если скобок не будет хватать.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 4 – Обработка ошибок

## Возникшие трудности

В ходе работы столкнулись с тем, что оригинальный make игнорирует пустые строки внутри цели, если за пустой строкой следует рецепт. Внесение символа переноса строки в правило цели приводило к ошибкам reduce/reduce, а вынесение его за границы цели позволяло писать shell команды вне таргета, но не приводило к критическим ошибкам. Таким образом, было решено допускать распознавание команд вне цели, но проверяя, что команда будет расположена внутри нее.

line:

…

| COMMAND (if (!in\_target) yyerror(“out of target”);)

…

Похожая ситуация с условными конструкциями. Их можно указывать как внутри цели, так и за ее пределом, но с условием, что вне цели не будут вызываться команды. Проверка аналогична предыдущей.

Также много проблем составили переменные. Переменная может содержать почти любую последовательность символов и не обрабатывается в момент объявления, потому что в дальнейшем будет включена в команду и анализироваться утилитой bash. Изначальная идея парсинга переменной в момент инициализации была отброшена, и в файле flex была заменена конструкцией “=”[\n]\*.

Для анализа встроенных в make переменных была добавлена функция, которая самостоятельно проверяет корректность расставленных скобок, поскольку на вход могут подаваться переменные и другие функции, содержащие множество скобок. В этом случае ни одно регулярное выражение не распознает вложенные скобки, поэтому проверка сводилась к их подсчету.

Заключение

В ходе выполнения курсовой работы был получен практический опыт разработки программы-парсера средствами yacc и flex. Реализованный подход позволяет сначала определить общие структурные конструкции языка, а затем конкретизировать и уточнить частные случаи, что является наиболее практичным и удобным методом разработки грамматики.

Кроме того, в данной курсовой работе была осуществлена обработка ошибок не только синтаксического, но и семантического характера, для реализации которой были использованы вставки, написанные на языке программирования Си, который, за счет совместимости с yacc и flex, предоставляет мощный инструмент для расширения функционала парсинга. В частности, с помощью таких вставок были реализованы счетчики грамматических конструкций, используемые в выводе статистических сообщений. Была реализована функция подсчета скобок и несколько режимов работы программы также с помощью языка Си.

В разработанной программе присутствует всего 13 конфликтов типа shift/reduce, что достаточно мало в сравнении с некоторыми другими языками. Также данные конфликты решаются, поскольку Bison предназначен для их разрешения путем выбора shift, если иное не указано в объявлениях приоритета оператора.

Исходя из результатов проделанной работы можно сделать вывод, что yacc и flex подходят для создания качественных распознавателей синтаксически простых языков, таких как язык make. Однако, нельзя однозначно утверждать пригодность вышеупомянутых средств в применении к более сложным языкам.