1. Министерство науки и высшего образования Российской Федерации
2. Санкт-Петербургский политехнический университет Петра Великого
3. —
4. Институт прикладной математики и механики
5. **Кафедра «Информационная безопасность компьютерных систем»**

**КУРСОВАЯ РАБОТА**

1. «ОПИСАНИЕ ГРАММАТИКИ ЯЗЫКА C»
2. по дисциплине «Формальные грамматики и теория компиляторов»

Выполнили

студенты гр. 33656/1 Климов А. М.

1. *<подпись>*

Федичев А. В.

*<подпись>*

Проверил

старший преподаватель Семьянов П. В.

*<подпись>*

Санкт-Петербург

2019

1. **Цель работы**

Изучение грамматики языка C, закрепление навыков работы с лексером и парсером «yacc» и «lex».

1. **Задача работы**

При помощи утилит yacc и lex описать грамматику языка C, разработать программу, на вход которой подается файл с исходным кодом данного языка, а на выходе выводится информация о том, является ли данный код корректным. В случае если код некорректный, выводится номер строки с ошибкой.

1. **Ход работы**

***ДОБАВИТЬ ПРО ОГРАНИЧЕНИЯ И ШИФТЫ***

Реализованный парсер языка C имеет следующие ограничения:

* Не поддерживается препроцессор языка C;
* Выражения с typedef считаются корректными, только в следующем виде:

*typedef тип имя;*

1. Были описаны все существующие в языке лексемы, ***НАПИСАТЬ ЧТО-НИБУДЬ ПРО ЛЕКСЕР***
2. Далее была описана грамматика языка. Любая программа состоит из следующих элементов:

* Объявления функций/переменных;
* Typedef;
* Функции.

1. Объявления функций/переменных.

***ДОБАВИТЬ ОПСАНИЕ ГРАММАТИКИ ОБЪЯВЛЕНИЯ ПЕРЕМЕННЫХ И ФУНКЦИЙ***

1. Typedef:

С помощью ключевого слова *typedef* язык C позволяет определять имена новых типов данных, при этом новый тип данных не создается, а определяется новое имя существующему типу. Стандартный вид оператора typedef следующий:

*typedef тип имя;*

где тип — это любой существующий тип данных, а имя – это новое имя для данного типа. Новое имя определяется в дополнение к существующему имени типа, а не замещает его.

Синтаксис конструкции typedef следующий:

*typedef →TYPEDEF typedef\_oldtype typedef\_newtypes,* где *typedef\_oldtype –* тип данных для которого определяется новое имя, а *typedef\_newtypes –* новое имя типа или имена через запятую. Грамматика данных нетерминалов представлена ниже:

*typedef\_oldtype → predefinitor*

*typedef\_newtypes →*

*typedef\_newtype*

*| typedef\_newtypes ',' typedef\_newtype*

*| typedef\_newtypes ',' type\_qualifier typedef\_newtype*

При реализации в курсовой работе конструкции *typedef* главной проблемой оказалось (тип сложно было придумать алгоритм, в яке нет каких-то средств, позволяющих на лету генерить/изменять тип терминала, т.к. необходимо отличить тип от идентификатора)

Проблема была решена следующим образом: был создан односвязный список, элементами которого является структура *typedef\_list,* в которой хранится строка – имя типа. При появлении токена, совпадающего с регулярным выражением [\_a-zA-Z][\_a-zA-Z0-9]\* сначала происходит проверка на наличие данного токена в списке имен типов, а затем если такой токен был найден, лексером возвращается результат CUSTOM\_TYPE, что свидетельствует о том, что данный токен является именем типа, иначе – возвращается NAME, т.е. токен является идентификатором.

Добавление элементов в список происходит при обработке парсером правила typedef.

1. Функции.

Любую функцию в C можно описать следующим образом:

*<type> function\_name <arguments> { <function\_body> }*

Здесь *<type>* – тип функции, *<arguments>* – аргументы функции, *<function\_body>* – тело функции.

Описание грамматики типов были рассмотрены выше. Аргументы функции описываются следующим образом:

***ДОБАВИТЬ ОПСАНИЕ ГРАММАТИКИ АРГУМЕНТОВ***

Тело функции представляет собой набор различных операторов и объявлений. Также, тело функции может быть пустым. Грамматика объявлений уже была рассмотрена, рассмотрим грамматику операторов.

Операторы (*statement*) в C могут быть нескольких типов:

* пустой оператор (*;*);
* метки (*label*);
* выражения (*expression;*);
* условные операторы (*conditional\_statement*);
* циклы (*loop\_statement*);
* переходы (*jump\_statement*);
* список операторов ( *{ statement\_list }* ).

Метки описываются следующим образом:

*label → NAME : statement*

Каждая метка имеет имя *NAME*, после имени ставится знак «*:*», а после него идет оператор (описание см. выше).

Выражения в C имеют следующий вид:

1. *expression →*

*assignment\_expression*

*| expression ',' assignment\_expression*

Выражения в C могут быть перечислены в строчку через запятую. Каждое отдельное выражение – выражение присваивания (assignment\_expression).

1. *assignment\_expression →*

*conditional\_expression*

*| prefix\_expression assignment\_operator*

*assignment\_expression*

Выражение присваивания может содержать различные операторы присваивания (*assignment\_operator → =, +=, -= и так далее*). На уровне ниже располагается условное выражение.

1. *conditional\_expression →*

*logical\_expression*

*| logical\_expression '?' expression ':' conditional\_expression*

Условное выражение имеет следующий вид:

*Условие ? Выражение1 : Выражение2;*

1. *logical\_expression →*

*compare\_expression*

*| logical\_expression logical\_operator compare\_expression*

Далее идет логическое выражение, которое может содержать логические операторы (*logical\_operator → &, &&, |, || и так далее*). Каждое такое выражение состоит из выражений сравнения.

1. *compare\_expression →*

*shift\_expression*

*| compare\_expression compare\_operator shift\_expression*

Выражения сравнения включают в себя операторы сравнения (*compare\_operator → >, ≥, <, ≤ и так далее*).

1. *shift\_expression →*

*simple\_expression*

*| shift\_expression SHIFT\_LEFT simple\_expression*

*| shift\_expression SHIFT\_RIGHT simple\_expression*

Выражения сдвигов включают в себя операторы побитовых сдвигов >> и <<.

1. *simple\_expression →*

*cast*

*| simple\_expression calc\_operator cast*

На данном этапе определяются обычные арифметические операции (*calc\_operator → +, -, \*, / и так далее*). На уровне ниже находятся операторы приведения типов.

1. *cast →*

*prefix\_expression*

*| '(' type\_specifier ')' cast*

*| '(' type\_specifier pointer\_id')' cast*

Приведение типов состоит из указания типа (в скобках) и выражения (дальше по цепочке вниз).

1. *prefix\_expression →*

*postfix\_expression*

*| INC prefix\_expression*

*| DEC prefix\_expression*

*| unary\_operator cast*

*| SIZEOF prefix\_expression*

*| SIZEOF '(' type\_specifier ')'*

На данном этапе описываются всевозможные префиксы для выражений (инкремент, декремент и так далее).

1. *postfix\_expression →*

*primary\_expression*

*| postfix\_expression '[' expression ']'*

*| postfix\_expression '(' ')'*

*| postfix\_expression '(' arguments\_without\_type ')'*

*| postfix\_expression '.' NAME*

*| postfix\_expression PTR NAME*

*| postfix\_expression INC*

*| postfix\_expression DEC*

Аналогично *prefix\_expression*, только здесь инкремент и декремент располагаются после основного выражения. Также здесь располагаются идентификаторы массивов.

1. *primary\_expression →*

*STRING*

*| NAME*

*| INTEGER*

*| FLOATING*

*| '(' expression ')'*

Самая низкая ступень дерева разложения грамматики. Здесь указываются базовые члены выражений (целые числа, числа с плавающей точкой, переменные и так далее).

Условные операторы в C могут быть двух типов:

* Оператор if;
* Оператор switch.

Данные типы условных операторов могут быть описаны следующим образом:

*conditional\_statement →*

*IF '(' expression ')' statement*

*| IF '(' expression ')' statement ELSE statement*

*| SWITCH '(' expression ')' switch\_statement*

Оператор *switch\_statement* отличается от обычного *statement* только тем, что в switch\_statement могут быть метки типа «*CASE expression : statement*» и «*DEFAULT : statement*»

Циклы в языке С можно описать следующим способом:

*loop\_statement →*

*WHILE '(' expression ')' statement*

*| DO statement WHILE '(' expression ')' ';'*

*| FOR '(' expression\_for\_loop ';' expression\_for\_loop ')'*

*statement*

*| FOR '(' expression\_for\_loop ';' expression\_for\_loop ';'*

*expression ')' statement*

Здесь выражения для задания цикла *expression\_for\_loop* описывается так:

*expression\_for\_loop →*

*predefinitor definitor*

*| expression*

Таким образом, первые два аргумента для цикла for можно задать либо объявив переменную прямо «на месте», либо сослаться на уже существующую.

Операторы переходов представляют собой операторы, которые позволяют немедленно перейти к какой-либо части кода. Описать такие операторы можно следующим образом:

*jump\_statement →*

*GOTO NAME ';'*

*| CONTINUE ';'*

*| BREAK ';'*

*| RETURN ';'*

*| RETURN expression ';'*

Здесь находятся такие ключевые слова, как *goto*, *return*, *break* и так далее.

Наконец, список операторов представляет собой набор операторов и определений переменных, заключенных в фигурные скобки:

*statement\_list →*

*statement*

*| definition*

*| statement\_list statement*

*| statement\_list definition*

1. ***ДОБАВИТЬ ЧТО-НИБУДЬ???***
2. **Результаты работы**

В результате работы была написана программа, на вход которой подается файл с исходным кодом на языке C. Программа выполняет проверку синтаксиса кода. На выходе программа сообщает, является ли данный код грамматически корректным или нет. Если в коде была обнаружена ошибка, программа указывает эту ошибку и строчку, в которой она расположена.

В ходе написания программы часто возникали конфликты сдвига/свертки. Некоторые такие конфликты решались способами, рассмотренными на лекциях по дисциплине «Формальные грамматики и теория компиляторов». Например, грамматика выражения:

E → E + E

была преобразована в:

E → T | E + T

T → …

Еще одним способом разрешения таких конфликтов было указание приоритетов некоторых терминалов.

Наиболее сложным в данной работе оказалось описание типов, т.к. средствами yacc тяжело правильно описать существующие в языке C простые типы, например, если *long long int*, *long float*, *unsigned*, *long* – корректные типы, то *long long char*, *int* *char*, *unsigned double* и т.д. – некорректные. Также, кроме того, что в каждой конструкции допустимо указывать лишь один класс хранения, в различных случаях возможно использование лишь определенных спецификаторов класса хранения: объявления глобальной видимости могут содержать только классы *extern* и *static*, аргументы – *register*, при описании полей структур и объединений и в конструкциях приведения типов нельзя указывать класс хранения. Для решения данной проблемы была создана вспомогательная структура:

*struct \_technical\_variables*

*{*

*TypeBasic type\_basic;*

*TypeAdditional type\_additional;*

*TypeSign type\_sign;*

*TypeStorage type\_storage;*

*SpecificationType specification\_type;*

*} technical\_variables;*

Все переменные в данной структуре принадлежат к перечислимому типу. В таблице 1 приведено описании вспомогательных перечислений и их значений.

*Таблица 1 – Описание реализованных вспомогательных типов*

|  |  |  |
| --- | --- | --- |
| Название перечислимого типа | Описание значений | Применение |
| *enum \_SpecificationType* | *SpecificationTypeStorageDenied* – определение классов хранения запрещено | Используется для определения типа конструкции относительно возможности указания определенных классов хранения. |
| *SpecificationTypeGlobal* – только классы хранения доступные для конструкций глобальной области видимости |
| *SpecificationTypeArgument* – аргументы функций |
| *SpecificationTypeNone* – значение по умолчанию, можно использовать любые спецификаторы классов хранения |
| *enum \_TypeBasic* | *TypeBasicInt*, *TypeBasicVoid*, *TypeBasicDouble*, *TypeBasicFloat*, *TypeBasicChar* – типы *int*, *void*, *double*, *float*, *char* соответственно | Используется для определения указанного явно типа данных. Использование переменной данного типа позволяет обнаружить многочисленное указание типа, а также позволяет согласовать использование дополнительных спецификаторов (*unsigned*, *singed*, *long* и т.д.) |
| *TypeBasicCompound* – составной тип |
| *TypeBasicNone* – тип не был указан явно |
| *enum \_TypeAdditional* | *TypeAdditionalLongLong*, *TypeAdditionalLong*, *TypeAdditionalShort* – указан спецификатор *long long*, *long* или *short* соответственно | Используется для определения указанного дополнительного спецификатора. Позволяет обнаружить многочисленное указание спецификаторов данного типа и позволяет согласовать использование этих спецификаторов с различными типами |
| *TypeAdditionalNone* – дополнительных спецификаторов не указано |
| *enum \_TypeSign* | *TypeSignSigned*, *TypeSignUnsigned* – спецификатор signed или unsigned | Используется для обнаружения некорректного сочетания спецификаторов |
| *TypeSignNone* – спецификаторы signed или unsigned не используются |
| *enum \_TypeStorage* | *TypeStorageAuto*, *TypeStorageExtern*, *TypeStorageStatic*, *TypeStorageRegister* – спецификаторы класса auto, extern, static, register соответственно | Используется для определения использования нескольких спецификаторов класса хранения и для обнаружения указания недопустимого спецификатора |
| *TypeStorageNone* – спефикатор хранения не указан |

1. **Выводы**

В ходе выполнения курсовой работы была изучена грамматика языка C, изучены принципы работы утилит «yacc» и «lex», а также закреплены навыки работы с данными утилитами.