1. Санкт-Петербургский политехнический университет Петра Великого
2. Институт прикладной математики и механики
3. **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 А. М. Климов

1. *<подпись>*

А.В. Федичев

*<подпись>*

Руководитель

старший преподаватель П. В. Семьянов

*<подпись>*

1. «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 201\_\_ г.

Санкт-Петербург

2019

1. **Цель работы**

Исследование возможности применения программ «yacc» и «lex» для написания парсера реального языка.

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**

Реализованный парсер языка C имеет следующие ограничения:

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

*‘typedef’ <тип> <имя>;*

* Глобальные объявления требуют указания типа;
* Использование конструкции typedef возможно только в глобальной области видимости;
* Не поддерживаются функции с переменным числом параметров;
* Не поддерживается использование оператора sizeof без скобок;
* Не поддерживается объявление указателей на функции;
* Не поддерживается объявление функций не в глобальной области видимости.

1. Была описана грамматика языка. Любая программа состоит из следующих элементов:

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

1. Объявления функций/переменных.

Объявления переменных были описаны следующим образом:

*[<спецификатор>] <тип> <имя> <размерность> {‘,’ <имя> <размерность>}*

Спецификаторы могут быть следующих типов:

* Спецификатор типа: int, char, double, float, составной тип;
* Спецификатор класса хранения: auto, register, extern, static;
* Квалификатор типа: volatile, const;
* Дополнительный спецификатор типа: long, short, long long.

Спецификаторы могут описываться в любой последовательности, однако нельзя указывать более одного спецификатора типа и дополнительного спецификатора типа, более одного спецификатора класса хранения. Каким образом проверяются спецификаторы написано в результатах работы.

Грамматика объявления переменных выглядит следующим образом:

*definition →*

*predefinitor ';'*

*| predefinitor definitors ';'*

Правило *predefinitor* определяется как правило указания типа вместе с спецификаторами:

*predefinitor →*

*storage\_class\_specifier*

*| type\_qualifier*

*| type\_specifier*

*| predefinitor type\_specifier*

*| predefinitor storage\_class\_specifier*

*| predefinitor type\_qualifier*

Правила *storage\_class\_specifier*, *type\_qualifier*, *type\_specifier* – возможные спецификаторы. Правило *definitors* – возможные идентификаторы:

*definitors →*

*definitor*

*| definitors ',' definitor*

*definitor →*

*definitor\_identificator*

*| definitor\_identificator '=' expression*

*| definitor\_identificator '=' initializer\_list*

Правило *definitor\_identificator* определяет возможный идентификатор:

*definitor\_identificator →*

*array\_or\_id*

*| pointer\_id type\_qualifier array\_or\_id*

*| pointer\_id array\_or\_id*

Функции объявляются аналогичным образом:

*function →*

*func\_prototype '{' statement\_list '}'*

*| func\_prototype '{' '}'*

*| func\_prototype ';'*

*func\_prototype →*

*prefuction '(' ')'*

*| prefuction '(' arguments ')'*

*| prefuction '(' arguments\_without\_type ')' definitions*

Правило *prefuction* описываетзаголовок функции до открывающейся круглой скобки:

*prefuction→*

*predefinitor func\_id*

Правило *func\_id –* возможное имя функции:

*func\_id →*

*NAME*

*| pointer\_id NAME*

Правило *arguments –* список формальных параметров функции*:*

*arguments →*

*predefinitor definitor*

*| arguments ',' predefinitor definitor*

*| predefinitor*

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 являлось то, что в lex нет подходящего средства, позволяющего отличить во время исполнения тип токена.

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.

Для работы с структурой *typedef\_list* были разработаны следующие функции:

* *existstype(char const \*xtype)* – возвращает положительное значение если имя типа (*xtype*) содержится в списке токенов, иначе возвращается нуль;
* *get\_type(const char \*token)* – в зависимости от того является ли *token* идентификатором или типом, определенным с помощью typedef, возвращается CUSTOM\_TYPE или NAME соответственно;
* *addtype(char const \*newtype)* – добавляет тип, определенный токеном *newtype* в список.

Исходный код разработанных функций приведен в приложении.

1. Функции.

Функция в реализованной грамматике была описана следующим образом:

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

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

Описание грамматики типов и аргументов функции были рассмотрены выше.

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:

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

Оператор «;» относится к оператору выражения.

Метки были описаны следующим образом:

*label → NAME : statement*

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

Выражения в разработанной грамматике имеют следующий вид:

*expressions →*

*expression*

*| expressions ',' expression*

*expression →*

*conditional\_expression*

Таким образом, в реализованной грамматике, тернарные выражения могут быть внутри обычных выражений. Тернарные операторы имеют вид:

*conditional\_expression →*

*simple\_expression*

*| simple\_expression '?' expressions ':' expression*

При написании грамматики тернарной операции сначала не было учтено, что возможно использование конструкции тернарной операции как аргумента при вызове функций, или как операнда выражений. Данная проблема была решена добавлением тернарного оператора в правило *expression.*

Где simple\_expression – обычные арифметические выражения:

*simple\_expression →*

*cast*

*| simple\_expression operator cast*

Также в реализованной грамматике оператор приведения типов cast тоже был добавлен как член выражения. Оператор cast имеет вид:

*cast →*

*prefix\_expression*

*| '(' caster ')' cast*

Где правилом caster описываются всевозможные типы данных. Правило *prefix\_expression* описывает возможные префиксы (инкремент, декремент), а правило *postfix\_expression* описывает возможные постфиксы (инкремент, декремент, вызов функции, обращение к элементу массива и др.):

*prefix\_expression →*

*postfix\_expression*

*| INC prefix\_expression*

*| INC casts prefix\_expression*

*| DEC prefix\_expression*

*| DEC casts prefix\_expression*

*| unary\_operator cast*

*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*

С помощью правила primary\_expression были описаны возможные конечные члены выражения (целые числа, вещественные числа, оператор sizeof и др.), а также выражения в скобках:

*primary\_expression →*

*'(' expression ')'*

*| NAME*

*| INTEGER*

*| FLOATING*

*| STRING*

*| sizeof*

Таким образом, в разработанной грамматике lvalue и rvalue смешаны. Данная ситуация в начале разработки была воспринята как проблема семантики, а не синтаксиса. Но, так как эта проблема была обнаружена на поздних стадиях разработки парсера, попытка исправить грамматику выражений с учетом lvalue и rvalue привела к значительному увеличению количества конфликтов.

Условные операторы в C могут быть двух типов:

* Оператор if;
* Оператор switch.

Данные типы условных операторов были 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 ';'*

*expression\_for\_loop ')' statement*

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

*expression\_for\_loop →*

*/\* empty \*/*

*| 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. Такие операторы были описаны следующим образом:

*jump\_statement →*

*GOTO NAME ';'*

*| RETURN ';'*

*| RETURN expression ';'*

Так как операторы переходов *break* и *continue* не могут располагаться в любом месте кода, были добавлены дополнительные правила, которые учитывают этот момент. Так, циклах, помимо обычных операторов перехода, описанных в правиле *statement* теперь могут располагаться операторы *break* и *continue*, а в операторах ветвления *switch* могут располагаться только *break*.

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. **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 определенных спецификаторов класса хранения: объявления глобальной видимости могут содержать только классы *extern* и *static*, аргументы – *register*, при описании полей структур, 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* – спецификатор хранения не указан |

Также присутствуют функции для установления значения полей в *technical\_variables.* Когда встречается токен-спецификатор или имя типа, устанавливается значение в соответствующее поле, при этом сравнивается, корректно ли выражение с получаемым сочетанием спецификаторов, если нет, то в консоль выводится сообщение об этом. Когда конструкция описания типа заканчивается, происходит очистка значений полей *technical\_variables*. Код разработанных функций установки значений полей в структуру *technical\_variables* представлен в приложении*.*

1. **Вывод**

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. Были получены навыки обработки входных файлов с учетом семантики. Также была разработана программа, выполняющая синтаксический анализ исходного кода на языке C. С помощью вставки кода на языке C в код парсера в данной работе была реализована поддержка *typedef* и проверка правильности сочетания спецификаторов типа.

Исходя из результатов работы можно сделать вывод, что с помощью программ yacc и lex можно разработать парсер реального языка, который будет содержать ограничения. Однако, можно эти ограничения можно избежать путем добавления большого количества правил или дополнительного кода на языке C, что может привести к появлению различного рода конфликтов.

Описание грамматики реального языка требует учета всех тонкостей языка и продумывания всех грамматических конструкций до непосредственной разработки, что является достаточно трудной задачей и приводит к возникновению ряда различных конфликтов, ограничений и допущений.

1. **Приложение**

Значимая часть кода для работы с typedef:

struct typedef\_list

{

char \*type\_id;

struct typedef\_list \*next;

} \*g\_typedefs;

int existstype(char const \*xtype)

{

struct typedef\_list \*tmp;

tmp = g\_typedefs;

while (tmp)

{

if (0 == strcmp(xtype, tmp->type\_id))

{

return 1;

}

tmp = tmp->next;

}

return 0;

}

int get\_type(const char\* token)

{

if(existstype(token))

{

return CUSTOM\_TYPE;

}

return NAME;

}

struct typedef\_list \*addtype(char const \*newtype)

{

struct typedef\_list \*tmp = (struct typedef\_list\*)malloc(sizeof(struct typedef\_list));

tmp->type\_id = (char\*)malloc(strlen(newtype)+1);

strcpy(tmp->type\_id, newtype);

tmp->next = NULL;

if (!g\_typedefs)

{

g\_typedefs = tmp;

}

else

{

tmp->next = g\_typedefs;

g\_typedefs = tmp;

}

Значимая часть кода для работы с типами:

typedef enum \_SpecificationType

{

SpecificationTypeStorageDenied,

SpecificationTypeGlobal,

SpecificationTypeArgument,

SpecificationTypeNone,

SpecificationTypePrev

}SpecificationType;

typedef enum \_TypeBasic

{

TypeBasicInt,

TypeBasicVoid,

TypeBasicDouble,

TypeBasicFloat,

TypeBasicChar,

TypeBasicCompound,

TypeBasicNone

}TypeBasic;

typedef enum \_TypeAdditional

{

TypeAdditionalLongLong,

TypeAdditionalLong,

TypeAdditionalShort,

TypeAdditionalNone

}TypeAdditional;

typedef enum \_TypeSign

{

TypeSignSigned,

TypeSignUnsigned,

TypeSignNone

}TypeSign;

typedef enum \_TypeStorage

{

TypeStorageAuto,

TypeStorageExtern,

TypeStorageRegister,

TypeStorageStatic,

TypeStorageNone,

}TypeStorage;

struct \_technical\_variables

{

TypeBasic type\_basic;

TypeAdditional type\_additional;

TypeSign type\_sign;

TypeStorage type\_storage;

SpecificationType specification\_type;

SpecificationType prev\_storage;

unsigned int auto\_cntr, extern\_cntr, register\_cntr, static\_cntr;

} technical\_variables;

void set\_specification(SpecificationType specification\_type)

{

//printf(" %d specification: %d %d %d\n", yylineno, technical\_variables.prev\_storage, technical\_variables.specification\_type, specification\_type);

if(specification\_type == SpecificationTypePrev)

{

//SpecificationType tmp=technical\_variables.prev\_storage;

//technical\_variables.prev\_storage = technical\_variables.specification\_type;

technical\_variables.specification\_type = technical\_variables.prev\_storage;

return;

}

technical\_variables.prev\_storage = technical\_variables.specification\_type;

technical\_variables.specification\_type = specification\_type;

}

void technical\_variables\_clean\_all()

{

technical\_variables.type\_basic = TypeBasicNone;

technical\_variables.type\_additional = TypeAdditionalNone;

technical\_variables.type\_sign = TypeSignNone;

technical\_variables.type\_storage = TypeStorageNone;

}

//#define DGB\_PRINT printf("%s %d\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

#define DGB\_PRINT

// сначала может быть объявлен доп тип, потом базовый. чекнуть

void set\_type\_basic(TypeBasic new\_type\_basic)

{

if(technical\_variables.type\_basic != TypeBasicNone )

{

yyerror("More than one type specified");

//YYACCEPT;

return;

}

if(technical\_variables.type\_sign != TypeSignNone &&

(new\_type\_basic != TypeBasicInt && new\_type\_basic != TypeBasicChar && new\_type\_basic != TypeBasicNone)

)

{

yyerror("More than one type specified");

//YYACCEPT;

return;

}

switch(technical\_variables.type\_additional)

{

case TypeAdditionalNone:

technical\_variables.type\_basic = new\_type\_basic;

return;

break;

case TypeAdditionalShort:

if(technical\_variables.type\_basic != TypeBasicInt &&

technical\_variables.type\_basic != TypeBasicFloat &&

technical\_variables.type\_basic != TypeBasicDouble &&

technical\_variables.type\_basic != TypeBasicNone

)

{

yyerror("More than one type specified");

//YYACCEPT;

return;

}

technical\_variables.type\_basic = new\_type\_basic;

return;

break;

case TypeAdditionalLongLong:

if(technical\_variables.type\_basic != TypeBasicInt &&

technical\_variables.type\_basic != TypeBasicDouble &&

technical\_variables.type\_basic != TypeBasicNone

)

{

yyerror("More than one type specified");

//YYACCEPT;

return;

}

technical\_variables.type\_basic = new\_type\_basic;

return;

break;

case TypeAdditionalLong:

if(technical\_variables.type\_basic != TypeBasicInt &&

technical\_variables.type\_basic != TypeBasicFloat &&

technical\_variables.type\_basic != TypeBasicDouble &&

technical\_variables.type\_basic != TypeBasicNone

)

{

yyerror("More than one type specified");

//YYACCEPT;

return;

}

technical\_variables.type\_basic = new\_type\_basic;

return;

break;

}

}

void set\_type\_additional(TypeAdditional new\_type\_additional)

{

if(technical\_variables.type\_additional != TypeAdditionalNone)

{

yyerror("Incorrect combination of specificators");

//YYACCEPT;

return;

}

else if(technical\_variables.type\_basic != TypeBasicInt &&

technical\_variables.type\_basic != TypeBasicFloat &&

technical\_variables.type\_basic != TypeBasicDouble &&

technical\_variables.type\_basic != TypeBasicNone)

{

printf("%d\n", TypeBasicNone);

yyerror("Incorrect combination of specificators");

//YYACCEPT;

return;

}

switch(technical\_variables.type\_basic)

{

case TypeBasicInt:

technical\_variables.type\_additional = new\_type\_additional;

return;

break;

case TypeBasicFloat:

if(new\_type\_additional != TypeAdditionalLong)

{

yyerror("Incorrect combination of specificators");

//YYACCEPT;

return;

}

technical\_variables.type\_additional = new\_type\_additional;

return;

break;

case TypeBasicDouble:

if(new\_type\_additional != TypeAdditionalLong &&

new\_type\_additional != TypeAdditionalLongLong

)

{

yyerror("Incorrect combination of specificators");

//YYACCEPT;

return;

}

technical\_variables.type\_additional = new\_type\_additional;

return;

break;

case TypeBasicNone:

technical\_variables.type\_additional = new\_type\_additional;

return;

break;

}

}

void set\_type\_sign(TypeSign new\_type\_sign)

{

if(technical\_variables.type\_sign == new\_type\_sign)

{

return;

}

else if(technical\_variables.type\_sign != TypeSignNone)

{

yyerror("Incorrect combination of specificators");

//YYACCEPT;

return;

}

if (technical\_variables.type\_basic != TypeBasicInt &&

technical\_variables.type\_basic != TypeBasicChar &&

technical\_variables.type\_basic != TypeBasicNone)

{

yyerror("Incorrect combination of specificators");

//YYACCEPT;

return;

}

else

{

technical\_variables.type\_sign = new\_type\_sign;

return;

}

}

void set\_type\_storage(TypeStorage new\_type\_storage)

{

SpecificationType specification\_type = technical\_variables.specification\_type;

if(technical\_variables.type\_storage != TypeStorageNone)

{

yyerror("More than one storage class specified");

//YYACCEPT;

return;

}

switch(specification\_type)

{

case SpecificationTypeArgument:

if(new\_type\_storage != TypeStorageRegister)

{

yyerror("Incorrect storage class specified");

//YYACCEPT;

return;

}

technical\_variables.type\_storage = new\_type\_storage;

return;

break;

case SpecificationTypeGlobal:

if(new\_type\_storage != TypeStorageStatic &&

new\_type\_storage != TypeStorageExtern)

{

yyerror("Incorrect storage class specified");

//YYACCEPT;

return;

}

technical\_variables.type\_storage = new\_type\_storage;

return;

break;

case SpecificationTypeStorageDenied:

yyerror("Storage class denied in this construction");

//YYACCEPT;

return;

break;

case SpecificationTypeNone:

technical\_variables.type\_storage = new\_type\_storage;

return;

break;

}

}