Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №3

По теме «Синтаксический анализатор»

Выполнил:
студент гр. 953501
Кореневский С. А.

Проверил:
ст. преподаватель Шиманский В. В.

Минск 2022

Содержание

1 Постановка задачи

В ходе синтаксического анализа исходный текст программы проверяется на на соответствие синтаксическим нормам языка с построением дерево разбора (синтаксическое дерево), которое отражает синтаксическую структуру входной последовательности и удобно для дальнейшего использования, а также в случае несоответствия – позволяет вывести сообщения об ошибках.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

Таким образом на основе анализа выражений, состоящих из литералов, операторов и круглых скобок выполняется группирование токенов исходной программы в грамматические фразы, используемые для синтеза вывода.

Представление грамматических фраз исходной программы выполнить в виде дерева. Реализовать синтаксический анализатор с использованием одного из табличных методов (LL-, LR-метод, метод предшествования и пр.).

Разбор выражения *COST* = (*PRICE*+*TAX*)*0.98.

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

$$<ИД_1>=(<ИД_2>+<ИД_3>)*<ИД_4>.$$

По этой цепочке необходимо выполнить следующие действия:

1)$<ИД_3>$ прибавить к $<ИД_2>$;

2)результат (1) умножить на $<ИД_4>$;

3)результат (2) поместить в ячейку, резервированную для $<ИД_1>$. Этой последовательности соответствует дерево, изображенное на рис. 1.

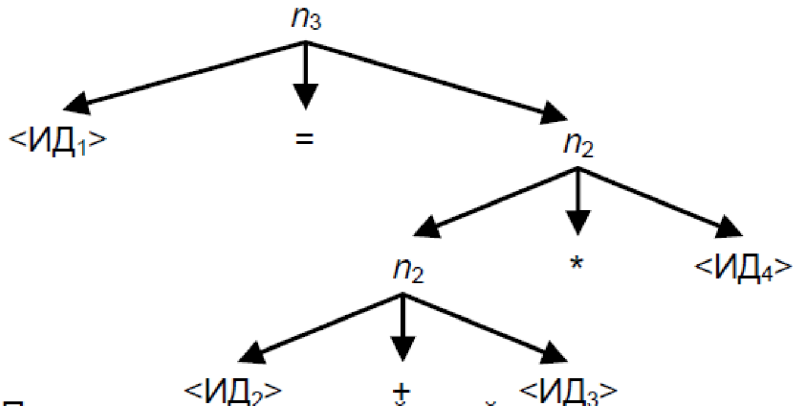

Рис. 1. – Последовательность действий при вычислении выражения

Т.е. мы имеем последовательность шагов в виде помеченного дерева.

Внутренние вершины представляют те действия, которые можно выполнять. Прямые потомки каждой вершины либо представляют аргументы, к которым нужно применять действие (если соответствующая вершина помечена идентификатором или является внутренней), либо помогают определить, каким должно быть это действие, в частности, знаки «+», «*» и «=». Скобки отсутствуют, т.к. они только определяют порядок действий.

LL- и LR- методы позволят обнаружить ошибки на самых ранних стадиях, т.е. когда разбор потока токенов от лексического анализатора в соответствии с грамматикой языка становится невозможен.

Можно использовать нисходящий (англ. top-down parser) со стартового символа, до получения требуемой последовательности токенов. Для этих целей применим метод рекурсивного спуска либо LL-анализатор. Или использовать восходящий (англ. bottom-up parser) - продукции восстанавливаются из правых частей, начиная с токенов и кончая стартовым символом - LR-анализатор и проч.

2 Теория

Синтаксический анализатор – это часть программы, преобразующей входные данные в структурированный формат; процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой, обычно применяемый совместно с лексическим анализом. Парсер выполняет синтаксический анализ текста.

В данной лабораторной работе необходимо реализовать простейший вариант синтаксического анализатора, поэтому будет достаточно использовать однопроходное абстрактное дерево.

При реализации синтаксического анализатора надо все символы, которые могут встретиться в обрабатываемом тексте разбить на группы таким образом, чтобы все символы группы вызывали одинаковую реакцию синтаксического анализатора, то есть это и есть лабораторная работа №1 по разбиению на токены. Затем необходимо выделить состояния синтаксического анализатора. Состояние определяет, какие символы в данный момент могут быть на входе синтаксического анализатора, и какова будет реакция на этот символ.

Одно состояние является начальным. Именно с него начинается работа синтаксического анализатора, и одно или несколько состояний должны быть конечными. Далее строиться таблица, которая определяет реакцию синтаксического анализатора на входные символы в зависимости от состояния. Реакция обычно заключается в смене состояния синтаксического анализатора и ещё каких-то действиях.

Если выражение записано верно, то в результате работы синтаксического анализатора должен появиться список лексем (список). Поскольку элементы списка должны иметь одинаковый тип, надо выбрать такую структуру, с помощью которой можно представить все возможные лексемы. Это можно сделать, если, например, каждый элемент списка представляет собой структуру, состоящую из двух полей: тип лексемы и её номер в списке лексем этого типа.

Часть лексем в арифметических и логических выражениях состоят из одного символа, но имена переменных и функций и константы в общем случае состоят из нескольких символов. Поэтому входные символы надо записывать во временную переменную, и когда данная лексема заканчивается, проверять, что она из себя представляет.

3 Пример работы программы

3.1 Исходный код программы

```
void mergeSort(int data[], int lenD)
{
  if (lenD>1) {
    int middle = lenD/2;
    int rem = lenD-middle;
    int *L = (int) middle;
    int *R = (int) rem;
    for (int i=0;i<lenD;i++){
      if (i<middle){
        L[i] = data[i];
      }
      else {
        R[i-middle] = data[i];
      }
    }
    mergeSort(L,middle);
    mergeSort(R,rem);
    merge(data, lenD, L, middle, R, rem);
  }
}

void merge(int merged[], int lenD, int L[], int lenL, int R[],
int lenR){
  int i = 0;
  int j = 0;
  while(i<lenL||j<lenR){
    if (i<lenL & j<lenR){
      if (L[i]<=R[j]){
        merged[i+j] == L[i];
        i++;
      }
      else {
        merged[i+j] = R[j];
        j++;
      }
    }
    else if (i<lenL){
      merged[i+j] == L[i];
      i++;
    }
    else if (j<lenR){
      merged [ i+j ] = R[ j ];
      j++;
    }
  }
}
```

3.2 Дерево программы

```
FileAST:
  FuncDef:
    Decl: mergeSort, [], [], []
      FuncDecl:
        ParamList:
          Decl: data, [], [], []
            ArrayDecl: []
              TypeDecl: data, []
                IdentifierType: ['int']
          Decl: lenD, [], [], []
            TypeDecl: lenD, []
              IdentifierType: ['int']
        TypeDecl: mergeSort, []
          IdentifierType: ['void']
    Compound:
      If:
        BinaryOp: >
          ID: lenD
          Constant: int, 1
        Compound:
          Decl: middle, [], [], []
            TypeDecl: middle, []
              IdentifierType: ['int']
            BinaryOp: /
              ID: lenD
              Constant: int, 2
          Decl: rem, [], [], []
            TypeDecl: rem, []
              IdentifierType: ['int']
            BinaryOp: -
              ID: lenD
              ID: middle
          Decl: L, [], [], []
            PtrDecl: []
              TypeDecl: L, []
                IdentifierType: ['int']
```

Рисунок 1 - Фрагмент дерева программы (часть 1)

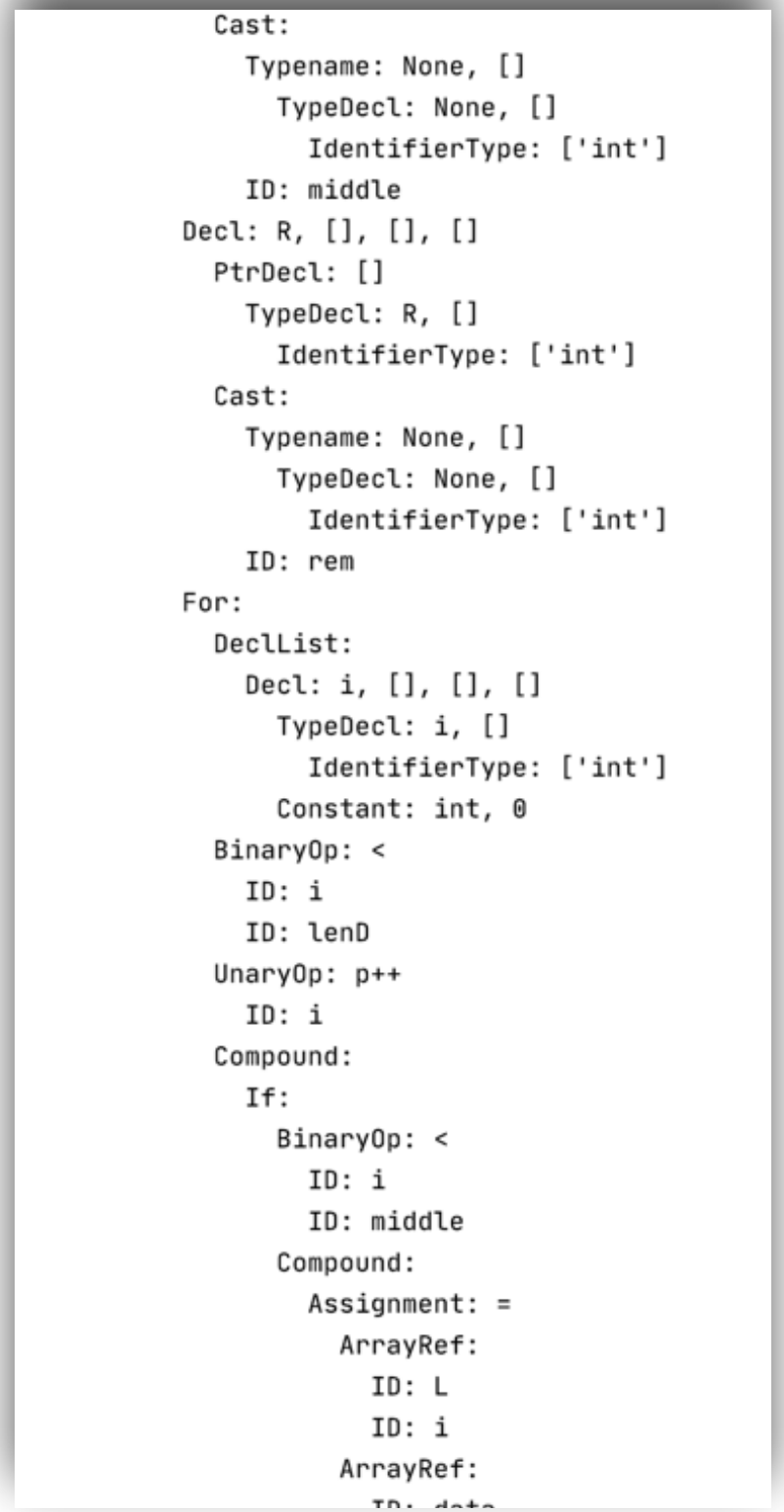

Рисунок 2 - Фрагмент дерева программы (часть 2)

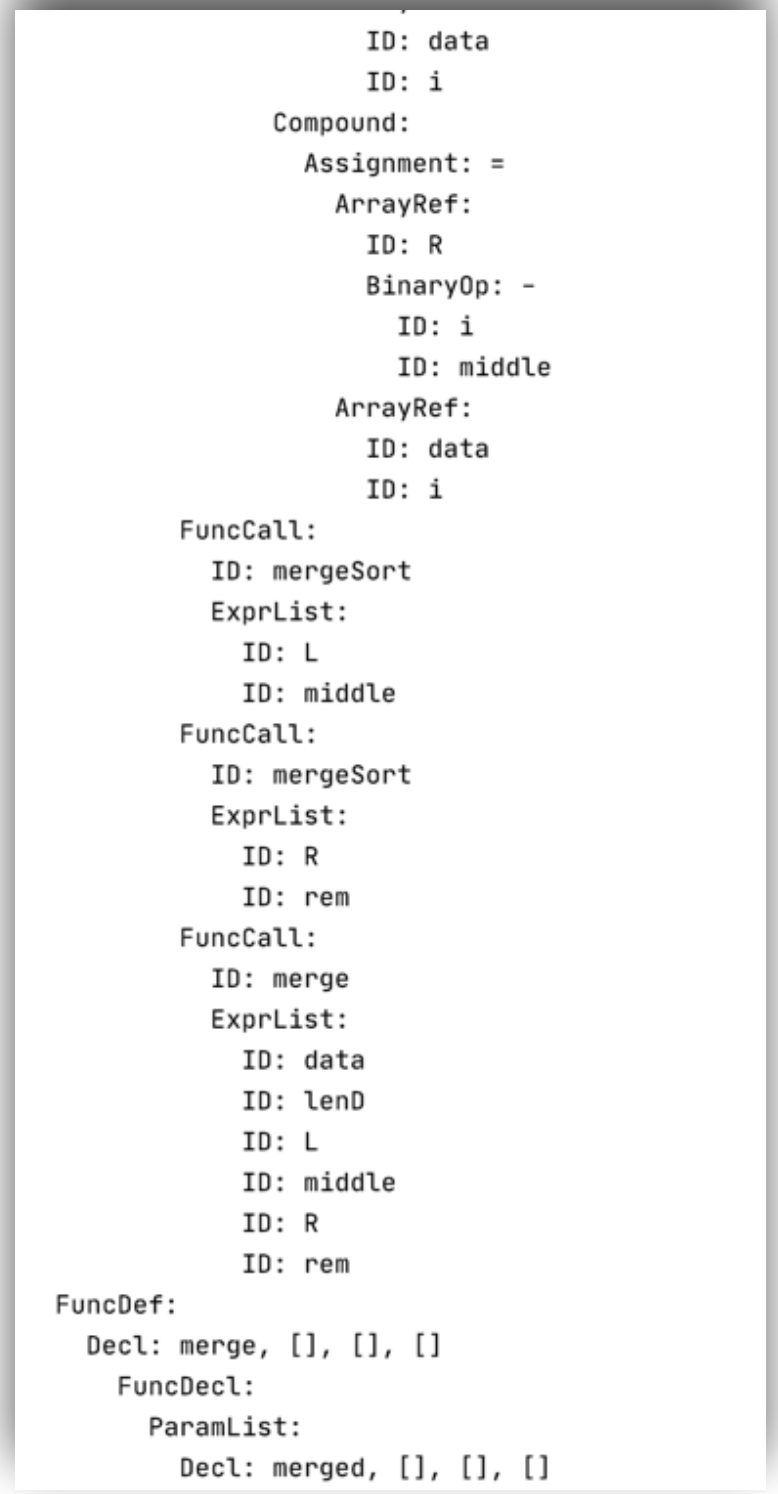

Рисунок 3 - Фрагмент дерева программы (часть 3)

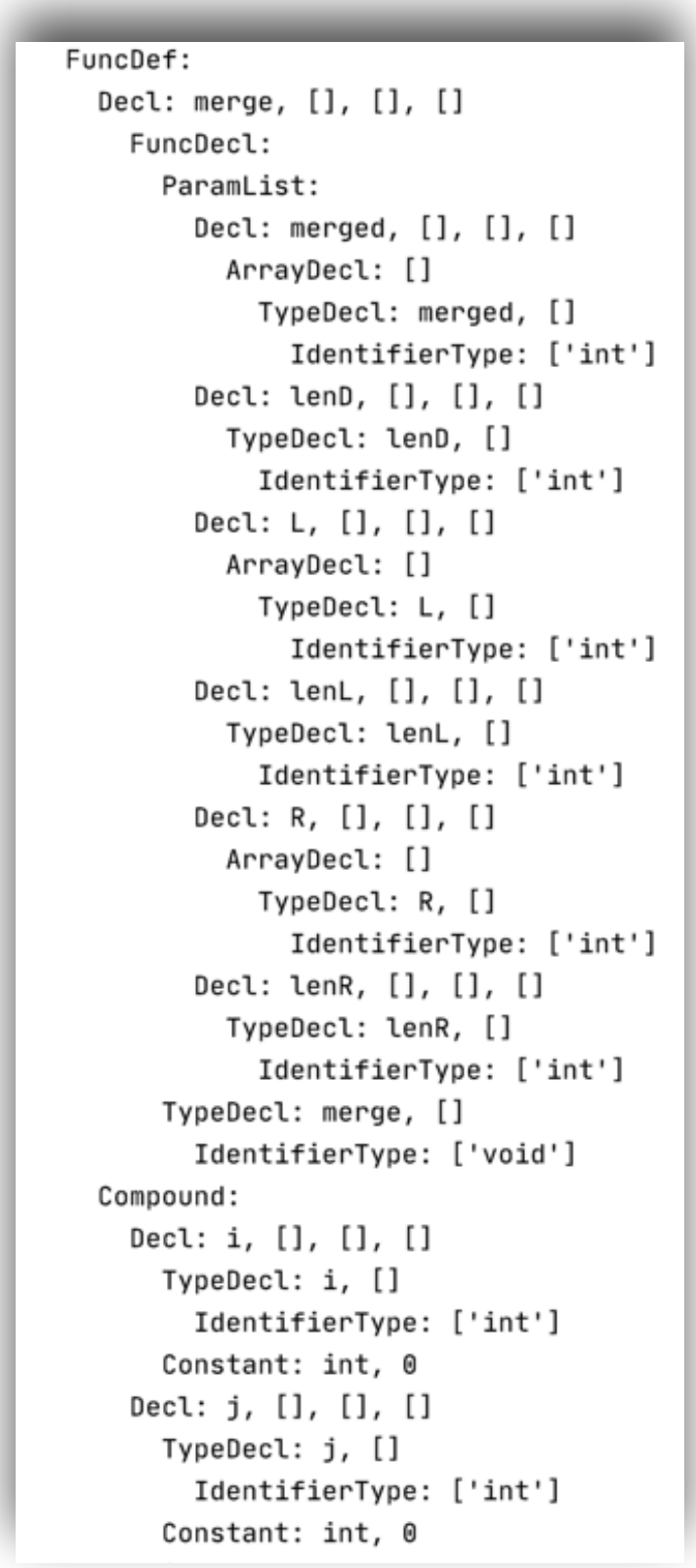

Рисунок 4 - Фрагмент дерева программы (часть 4)

```
While:
  BinaryOp: ||
    BinaryOp: <
      ID: i
      ID: lenL
    BinaryOp: <
      ID: j
      ID: lenR
  Compound:
    If:
      BinaryOp: &
        BinaryOp: <
          ID: i
          ID: lenL
        BinaryOp: <
          ID: j
          ID: lenR
      Compound:
        If:
          BinaryOp: <=
            ArrayRef:
              ID: L
              ID: i
            ArrayRef:
              ID: R
              ID: j
          Compound:
            BinaryOp: ==
              ArrayRef:
                ID: merged
                BinaryOp: +
                  ID: i
                  ID: j
              ArrayRef:
                ID: L
                ID: i
```

Рисунок 5 - Фрагмент дерева программы (часть 5)

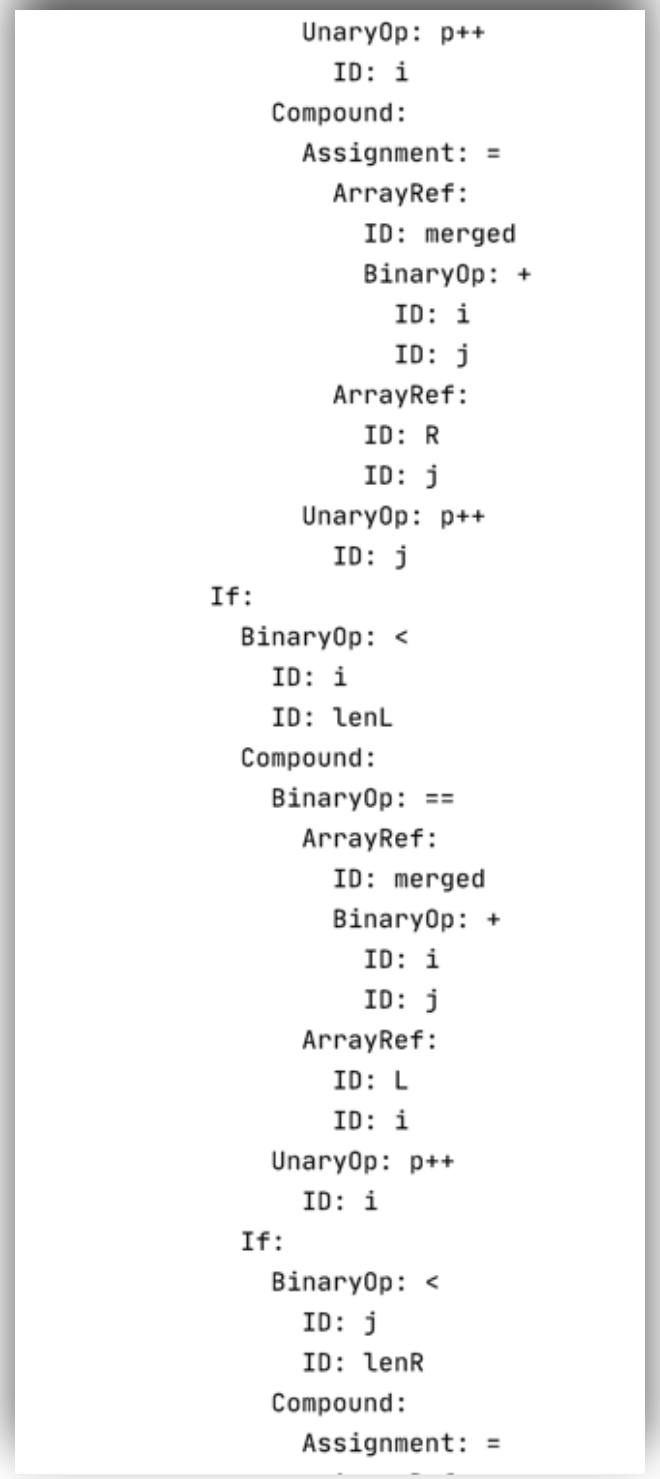

Рисунок 6 - Фрагмент дерева программы (часть 6)

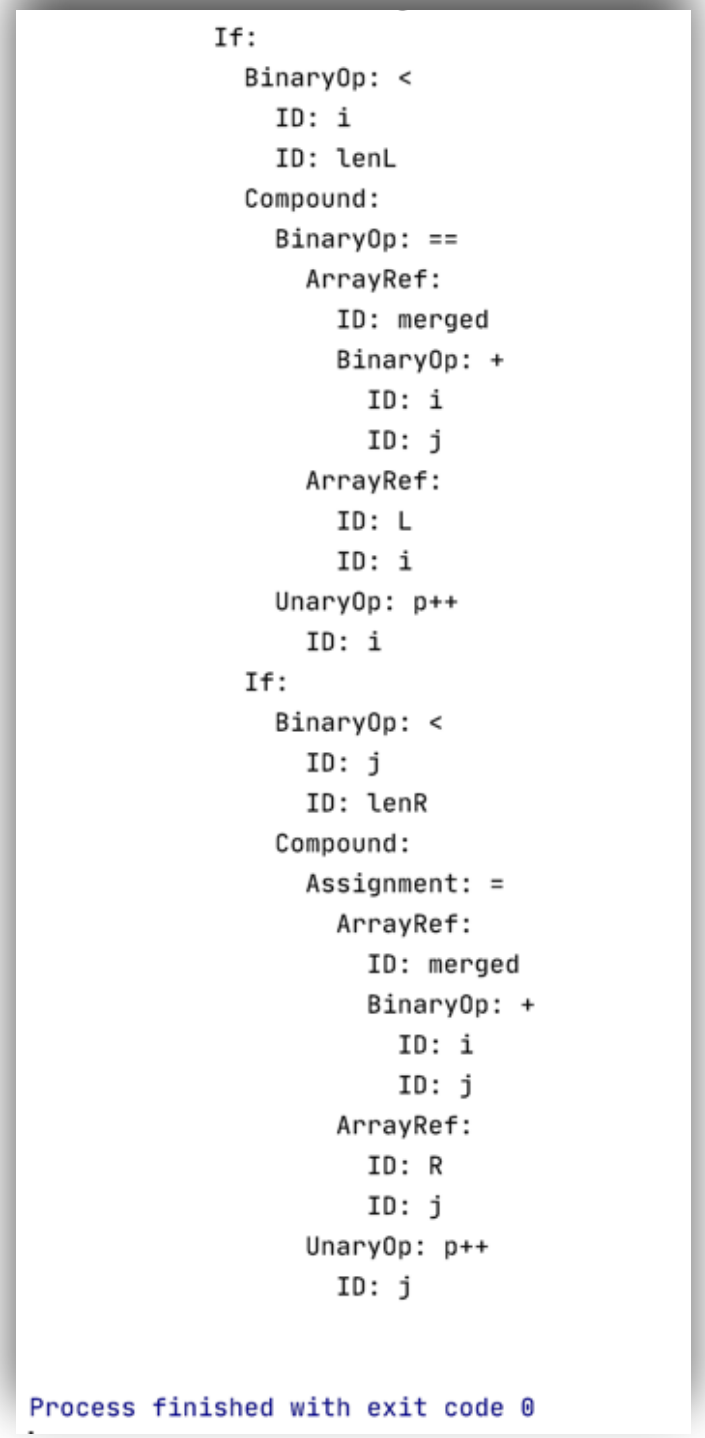

Рисунок 7 - Фрагмент дерева программы (часть 7)

3.3 Дерево программы в текстовом варианте

```
FileAST:
  FuncDef:
    Decl: mergeSort, [], [], []
      FuncDecl:
        ParamList:
          Decl: data, [], [], []
            ArrayDecl: []
              TypeDecl: data, []
                IdentifierType: ['int']
          Decl: lenD, [], [], []
            TypeDecl: lenD, []
              IdentifierType: ['int']
        TypeDecl: mergeSort, []
          IdentifierType: ['void']
    Compound:
      If:
        BinaryOp: >
          ID: lenD
          Constant: int, 1
        Compound:
          Decl: middle, [], [], []
            TypeDecl: middle, []
              IdentifierType: ['int']
            BinaryOp: /
              ID: lenD
              Constant: int, 2
          Decl: rem, [], [], []
            TypeDecl: rem, []
              IdentifierType: ['int']
            BinaryOp: -
              ID: lenD
              ID: middle
          Decl: L, [], [], []
            PtrDecl: []
              TypeDecl: L, []
                IdentifierType: ['int']
            Cast:
              Typename: None, []
                TypeDecl: None, []
                  IdentifierType: ['int']
              ID: middle
          Decl: R, [], [], []
            PtrDecl: []
```

```
    TypeDecl: R, []
      IdentifierType: ['int']
  Cast:
    Typename: None, []
      TypeDecl: None, []
        IdentifierType: ['int']
    ID: rem
For:
  DeclList:
    Decl: i, [], [], []
      TypeDecl: i, []
        IdentifierType: ['int']
      Constant: int, 0
  BinaryOp: <
    ID: i
    ID: lenD
  UnaryOp: p++
    ID: i
  Compound:
    If:
      BinaryOp: <
        ID: i
        ID: middle
      Compound:
        Assignment: =
          ArrayRef:
            ID: L
            ID: i
          ArrayRef:
            ID: data
            ID: i
      Compound:
        Assignment: =
          ArrayRef:
            ID: R
            BinaryOp: -
              ID: i
              ID: middle
          ArrayRef:
            ID: data
            ID: i
FuncCall:
  ID: mergeSort
  ExprList:
    ID: L
```

```
            ID: middle
          FuncCall:
            ID: mergeSort
            ExprList:
              ID: R
              ID: rem
          FuncCall:
            ID: merge
            ExprList:
              ID: data
              ID: lenD
              ID: L
              ID: middle
              ID: R
              ID: rem
FuncDef:
  Decl: merge, [], [], []
    FuncDecl:
      ParamList:
        Decl: merged, [], [], []
          ArrayDecl: []
            TypeDecl: merged, []
              IdentifierType: ['int']
        Decl: lenD, [], [], []
          TypeDecl: lenD, []
            IdentifierType: ['int']
        Decl: L, [], [], []
          ArrayDecl: []
            TypeDecl: L, []
              IdentifierType: ['int']
        Decl: lenL, [], [], []
          TypeDecl: lenL, []
            IdentifierType: ['int']
        Decl: R, [], [], []
          ArrayDecl: []
            TypeDecl: R, []
              IdentifierType: ['int']
        Decl: lenR, [], [], []
          TypeDecl: lenR, []
            IdentifierType: ['int']
      TypeDecl: merge, []
        IdentifierType: ['void']
  Compound:
    Decl: i, [], [], []
      TypeDecl: i, []
```

```
    IdentifierType: ['int']
  Constant: int, 0
Decl: j, [], [], []
  TypeDecl: j, []
    IdentifierType: ['int']
  Constant: int, 0
While:
  BinaryOp: ||
    BinaryOp: <
      ID: i
      ID: lenL
    BinaryOp: <
      ID: j
      ID: lenR
  Compound:
    If:
      BinaryOp: &
        BinaryOp: <
          ID: i
          ID: lenL
        BinaryOp: <
          ID: j
          ID: lenR
      Compound:
        If:
          BinaryOp: <=
            ArrayRef:
              ID: L
              ID: i
            ArrayRef:
              ID: R
              ID: j
          Compound:
            BinaryOp: ==
              ArrayRef:
                ID: merged
                BinaryOp: +
                  ID: i
                  ID: j
              ArrayRef:
                ID: L
                ID: i
            UnaryOp: p++
              ID: i
          Compound:
```

```
      Assignment: =
        ArrayRef:
          ID: merged
          BinaryOp: +
            ID: i
            ID: j
        ArrayRef:
          ID: R
          ID: j
      UnaryOp: p++
        ID: j
If:
  BinaryOp: <
    ID: i
    ID: lenL
  Compound:
    BinaryOp: ==
      ArrayRef:
        ID: merged
        BinaryOp: +
          ID: i
          ID: j
      ArrayRef:
        ID: L
        ID: i
    UnaryOp: p++
      ID: i
  If:
    BinaryOp: <
      ID: j
      ID: lenR
    Compound:
      Assignment: =
        ArrayRef:
          ID: merged
          BinaryOp: +
            ID: i
            ID: j
        ArrayRef:
          ID: R
          ID: j
      UnaryOp: p++
        ID: j
```

4 Синтаксические ошибки

Изменим исходный код программы, сознательно сделав несколько синтаксических ошибок:

1) незакрытая строка

```
SYNTAX ERROR (missing ;)
 row 42 pos 7
  merged [ i+j ] = R[ j ]
      j++;

Process finished with exit code 0
```

Рисунок 8 – Синтактическая ошибка 1

2) Введем лишний оператор «+»:

```
SYNTAX ERROR (лишний оператор +)
 row 30 pos 4
            if (L[i]<=R[j]){
                merged[i+j] == L[i];
                i+++;

Process finished with exit code 0
```

Рисунок 9 – Синтактическая ошибка 2

3) Не закроем скобку:

```
SYNTAX ERROR (не закрыта скобка)
 row 9 pos 22
      for (int i=0;i<lenD;i++{
            if (i<middle){
                L[i] = data[i];

Process finished with exit code 0
```

Рисунок 10 – Синтактическая ошибка 3

4) Введем лишний оператор «-»:

```
SYNTAX ERROR (лишний оператор -)
 row 42 pos 10
      else if (j<lenR){
            merged [ i+-j ] = R[ j ];
            j++;
```

Рисунок 11 – Синтактическая ошибка 4

5) Добавим число в неожиданное место без оператора:

```
SYNTAX ERROR (лишняя скобочка закрывающаюся)
 row 37 pos 17
     else if (i<lenL)){
            merged[i+j] == L[i];
            i++;

Process finished with exit code 0
```

Рисунок 12 – Синтактическая ошибка 5

5 Выводы

Сложность синтаксического анализа в доработке лексического анализатора и проверке на вложенность.

Синтаксический анализатор представляет собой вторую фазу компилятора, его основная задача состоит из создания программы, способной создавать синтаксическое дерево, состоящее из списка (list) на языке Python. Это дерево в дальнейшем необходимо для создания виртуальной машины или, другими словами, интерпретатора.

Приложение 1. Текст программы

```
from .ply import yacc

from . import c_ast
from .c_lexer import CLexer
from .plyparser import PLYParser, ParseError, parameterized, template
from .ast_transforms import fix_switch_cases, fix_atomic_specifiers

@template
class CParser(PLYParser):
    def __init__(
            self,
            lex_optimize=True,
            lexer=CLexer,
            lextab='pycparser.lextab',
            yacc_optimize=True,
            yacctab='pycparser.yacctab',
            yacc_debug=False,
            taboutputdir=''):

        self.clex = lexer(
            error_func=self._lex_error_func,
            on_lbrace_func=self._lex_on_lbrace_func,
            on_rbrace_func=self._lex_on_rbrace_func,
            type_lookup_func=self._lex_type_lookup_func)

        self.clex.build(
            optimize=lex_optimize,
            lextab=lextab,
            outputdir=taboutputdir)
        self.tokens = self.clex.tokens

        rules_with_opt = [
            'abstract_declarator',
            'assignment_expression',
            'declaration_list',
            'declaration_specifiers_no_type',
            'designation',
            'expression',
            'identifier_list',
            'init_declarator_list',
            'id_init_declarator_list',
            'initializer_list',
            'parameter_type_list',
            'block_item_list',
            'type_qualifier_list',
            'struct_declarator_list'
```

```
    ]

    for rule in rules_with_opt:
      self._create_opt_rule(rule)

    self.cparser = yacc.yacc(
      module=self,
      start='translation_unit_or_empty',
      debug=yacc_debug,
      optimize=yacc_optimize,
      tabmodule=yacctab,
      outputdir=taboutputdir)

    self._scope_stack = [dict()]

    # Keeps track of the last token given to yacc (the lookahead token)
    self._last_yielded_token = None

  def parse(self, text, filename='', debug=False):
    """ Parses C code and returns an AST.

      text:
        A string containing the C source code

      filename:
        Name of the file being parsed (for meaningful
        error messages)

      debug:
        Debug flag to YACC
    """
    self.clex.filename = filename
    self.clex.reset_lineno()
    self._scope_stack = [dict()]
    self._last_yielded_token = None
    return self.cparser.parse(
        input=text,
        lexer=self.clex,
        debug=debug)


  def _push_scope(self):
    self._scope_stack.append(dict())

  def _pop_scope(self):
    assert len(self._scope_stack) > 1
    self._scope_stack.pop()
```

```
def _add_typedef_name(self, name, coord):
    """ Add a new typedef name (ie a TYPEID) to the current scope
    """
    if not self._scope_stack[-1].get(name, True):
        self._parse_error(
            "Typedef %r previously declared as non-typedef "
            "in this scope" % name, coord)
    self._scope_stack[-1][name] = True

def _add_identifier(self, name, coord):
    """ Add a new object, function, or enum member name (ie an ID) to the
        current scope
    """
    if self._scope_stack[-1].get(name, False):
        self._parse_error(
            "Non-typedef %r previously declared as typedef "
            "in this scope" % name, coord)
    self._scope_stack[-1][name] = False

def _is_type_in_scope(self, name):
    """ Is *name* a typedef-name in the current scope?
    """
    for scope in reversed(self._scope_stack):
        # If name is an identifier in this scope it shadows typedefs in
        # higher scopes.
        in_scope = scope.get(name)
        if in_scope is not None: return in_scope
    return False

def _lex_error_func(self, msg, line, column):
    self._parse_error(msg, self._coord(line, column))

def _lex_on_lbrace_func(self):
    self._push_scope()

def _lex_on_rbrace_func(self):
    self._pop_scope()

def _lex_type_lookup_func(self, name):
    """ Looks up types that were previously defined with
        typedef.
        Passed to the lexer for recognizing identifiers that
        are types.
    """
    is_type = self._is_type_in_scope(name)
    return is_type

def _get_yacc_lookahead_token(self):
```

```
    """ We need access to yacc's lookahead token in certain cases.
        This is the last token yacc requested from the lexer, so we
        ask the lexer.
    """
    return self.clex.last_token

def _type_modify_decl(self, decl, modifier):
    """ Tacks a type modifier on a declarator, and returns
        the modified declarator.

        Note: the declarator and modifier may be modified
    """
    #~ print '****'
    #~ decl.show(offset=3)
    #~ modifier.show(offset=3)
    #~ print '****'

    modifier_head = modifier
    modifier_tail = modifier

    # The modifier may be a nested list. Reach its tail.
    while modifier_tail.type:
        modifier_tail = modifier_tail.type

    # If the decl is a basic type, just tack the modifier onto it.
    if isinstance(decl, c_ast.TypeDecl):
        modifier_tail.type = decl
        return modifier
    else:
        # Otherwise, the decl is a list of modifiers. Reach
        # its tail and splice the modifier onto the tail,
        # pointing to the underlying basic type.
        decl_tail = decl

        while not isinstance(decl_tail.type, c_ast.TypeDecl):
            decl_tail = decl_tail.type

        modifier_tail.type = decl_tail.type
        decl_tail.type = modifier_head
        return decl

# Due to the order in which declarators are constructed,
# they have to be fixed in order to look like a normal AST.
#
# When a declaration arrives from syntax construction, it has
# these problems:
# * The innermost TypeDecl has no type (because the basic
#   type is only known at the uppermost declaration level)
```

```
# * The declaration has no variable name, since that is saved
#   in the innermost TypeDecl
# * The typename of the declaration is a list of type
#   specifiers, and not a node. Here, basic identifier types
#   should be separated from more complex types like enums
#   and structs.
#
# This method fixes these problems.
def _fix_decl_name_type(self, decl, typename):
    """ Fixes a declaration. Modifies decl.
    """
    # Reach the underlying basic type
    #
    type = decl
    while not isinstance(type, c_ast.TypeDecl):
        type = type.type

    decl.name = type.declname
    type.quals = decl.quals[:]

    # The typename is a list of types. If any type in this
    # list isn't an IdentifierType, it must be the only
    # type in the list (it's illegal to declare "int enum ..")
    # If all the types are basic, they're collected in the
    # IdentifierType holder.
    for tn in typename:
        if not isinstance(tn, c_ast.IdentifierType):
            if len(typename) > 1:
                self._parse_error(
                    "Invalid multiple types specified", tn.coord)
            else:
                type.type = tn
                return decl

    if not typename:
        # Functions default to returning int
        #
        if not isinstance(decl.type, c_ast.FuncDecl):
            self._parse_error(
                    "Missing type in declaration", decl.coord)
        type.type = c_ast.IdentifierType(
                ['int'],
                coord=decl.coord)
    else:
        # At this point, we know that typename is a list of IdentifierType
        # nodes. Concatenate all the names into a single list.
        #
        type.type = c_ast.IdentifierType(
```

```
            [name for id in typename for name in id.names],
            coord=typename[0].coord)
    return decl

def _add_declaration_specifier(self, declspec, newspec, kind, append=False):
    """ Declaration specifiers are represented by a dictionary
        with the entries:
        * qual: a list of type qualifiers
        * storage: a list of storage type qualifiers
        * type: a list of type specifiers
        * function: a list of function specifiers
        * alignment: a list of alignment specifiers

        This method is given a declaration specifier, and a
        new specifier of a given kind.
        If `append` is True, the new specifier is added to the end of
        the specifiers list, otherwise it's added at the beginning.
        Returns the declaration specifier, with the new
        specifier incorporated.
    """
    spec = declspec or dict(qual=[], storage=[], type=[], function=[], alignment=[])

    if append:
        spec[kind].append(newspec)
    else:
        spec[kind].insert(0, newspec)

    return spec

def _build_declarations(self, spec, decls, typedef_namespace=False):
    """ Builds a list of declarations all sharing the given specifiers.
        If typedef_namespace is true, each declared name is added
        to the "typedef namespace", which also includes objects,
        functions, and enum constants.
    """
    is_typedef = 'typedef' in spec['storage']
    declarations = []

    # Bit-fields are allowed to be unnamed.
    if decls[0].get('bitsize') is not None:
        pass

    # When redeclaring typedef names as identifiers in inner scopes, a
    # problem can occur where the identifier gets grouped into
    # spec['type'], leaving decl as None.  This can only occur for the
    # first declarator.
    elif decls[0]['decl'] is None:
        if len(spec['type']) < 2 or len(spec['type'][-1].names) != 1 or \
```

```
                not self._is_type_in_scope(spec['type'][-1].names[0]):
            coord = '?'
            for t in spec['type']:
                if hasattr(t, 'coord'):
                    coord = t.coord
                    break
            self._parse_error('Invalid declaration', coord)

        # Make this look as if it came from "direct_declarator:ID"
        decls[0]['decl'] = c_ast.TypeDecl(
            declname=spec['type'][-1].names[0],
            type=None,
            quals=None,
            align=spec['alignment'],
            coord=spec['type'][-1].coord)
        # Remove the "new" type's name from the end of spec['type']
        del spec['type'][-1]

    # A similar problem can occur where the declaration ends up looking
    # like an abstract declarator.  Give it a name if this is the case.
    elif not isinstance(decls[0]['decl'], (
            c_ast.Enum, c_ast.Struct, c_ast.Union, c_ast.IdentifierType)):
        decls_0_tail = decls[0]['decl']
        while not isinstance(decls_0_tail, c_ast.TypeDecl):
            decls_0_tail = decls_0_tail.type
        if decls_0_tail.declname is None:
            decls_0_tail.declname = spec['type'][-1].names[0]
            del spec['type'][-1]

    for decl in decls:
        assert decl['decl'] is not None
        if is_typedef:
            declaration = c_ast.Typedef(
                name=None,
                quals=spec['qual'],
                storage=spec['storage'],
                type=decl['decl'],
                coord=decl['decl'].coord)
        else:
            declaration = c_ast.Decl(
                name=None,
                quals=spec['qual'],
                align=spec['alignment'],
                storage=spec['storage'],
                funcspec=spec['function'],
                type=decl['decl'],
                init=decl.get('init'),
                bitsize=decl.get('bitsize'),
```

```
                coord=decl['decl'].coord)

            if isinstance(declaration.type, (
                c_ast.Enum, c_ast.Struct, c_ast.Union,
                c_ast.IdentifierType)):
              fixed_decl = declaration
            else:
              fixed_decl = self._fix_decl_name_type(declaration, spec['type'])

            # Add the type name defined by typedef to a
            # symbol table (for usage in the lexer)
            if typedef_namespace:
              if is_typedef:
                self._add_typedef_name(fixed_decl.name, fixed_decl.coord)
              else:
                self._add_identifier(fixed_decl.name, fixed_decl.coord)

            fixed_decl = fix_atomic_specifiers(fixed_decl)
            declarations.append(fixed_decl)

        return declarations

    def _build_function_definition(self, spec, decl, param_decls, body):
        """ Builds a function definition.
        """
        if 'typedef' in spec['storage']:
            self._parse_error("Invalid typedef", decl.coord)

        declaration = self._build_declarations(
            spec=spec,
            decls=[dict(decl=decl, init=None)],
            typedef_namespace=True)[0]

        return c_ast.FuncDef(
            decl=declaration,
            param_decls=param_decls,
            body=body,
            coord=decl.coord)

    def _select_struct_union_class(self, token):
        """ Given a token (either STRUCT or UNION), selects the
            appropriate AST class.
        """
        if token == 'struct':
            return c_ast.Struct
        else:
            return c_ast.Union
```

```
##
## Precedence and associativity of operators
##
# If this changes, c_generator.CGenerator.precedence_map needs to change as
# well
precedence = (
   ('left', 'LOR'),
   ('left', 'LAND'),
   ('left', 'OR'),
   ('left', 'XOR'),
   ('left', 'AND'),
   ('left', 'EQ', 'NE'),
   ('left', 'GT', 'GE', 'LT', 'LE'),
   ('left', 'RSHIFT', 'LSHIFT'),
   ('left', 'PLUS', 'MINUS'),
   ('left', 'TIMES', 'DIVIDE', 'MOD')
)

##
## Grammar productions
## Implementation of the BNF defined in K&R2 A.13
##

# Wrapper around a translation unit, to allow for empty input.
# Not strictly part of the C99 Grammar, but useful in practice.
def p_translation_unit_or_empty(self, p):
   """ translation_unit_or_empty   : translation_unit
                                   | empty
   """
   if p[1] is None:
      p[0] = c_ast.FileAST([])
   else:
      p[0] = c_ast.FileAST(p[1])

def p_translation_unit_1(self, p):
   """ translation_unit    : external_declaration
   """
   # Note: external_declaration is already a list
   p[0] = p[1]

def p_translation_unit_2(self, p):
   """ translation_unit    : translation_unit external_declaration
   """
   p[1].extend(p[2])
   p[0] = p[1]

# Declarations always come as lists (because they can be
# several in one line), so we wrap the function definition
```

```
    # into a list as well, to make the return value of
    # external_declaration homogeneous.
    def p_external_declaration_1(self, p):
        """ external_declaration    : function_definition
        """
        p[0] = [p[1]]

    def p_external_declaration_2(self, p):
        """ external_declaration    : declaration
        """
        p[0] = p[1]

    def p_external_declaration_3(self, p):
        """ external_declaration    : pp_directive
                                    | pppragma_directive
        """
        p[0] = [p[1]]

    def p_external_declaration_4(self, p):
        """ external_declaration    : SEMI
        """
        p[0] = []

    def p_external_declaration_5(self, p):
        """ external_declaration    : static_assert
        """
        p[0] = p[1]

    def p_static_assert_declaration(self, p):
        """ static_assert           : _STATIC_ASSERT LPAREN constant_expression COMMA
    unified_string_literal RPAREN
                                    | _STATIC_ASSERT LPAREN constant_expression RPAREN
        """
        if len(p) == 5:
            p[0] = [c_ast.StaticAssert(p[3], None, self._token_coord(p, 1))]
        else:
            p[0] = [c_ast.StaticAssert(p[3], p[5], self._token_coord(p, 1))]

    def p_pp_directive(self, p):
        """ pp_directive  : PPHASH
        """
        self._parse_error('Directives not supported yet',
                          self._token_coord(p, 1))

    def p_pppragma_directive(self, p):
        """ pppragma_directive      : PPPRAGMA
                                    | PPPRAGMA PPPRAGMASTR
        """
```

```
        if len(p) == 3:
            p[0] = c_ast.Pragma(p[2], self._token_coord(p, 2))
        else:
            p[0] = c_ast.Pragma("", self._token_coord(p, 1))

    # In function definitions, the declarator can be followed by
    # a declaration list, for old "K&R style" function definitios.
    def p_function_definition_1(self, p):
        """ function_definition : id_declarator declaration_list_opt compound_statement
        """
        # no declaration specifiers - 'int' becomes the default type
        spec = dict(
            qual=[],
            alignment=[],
            storage=[],
            type=[c_ast.IdentifierType(['int'],
                                       coord=self._token_coord(p, 1))],
            function=[])

        p[0] = self._build_function_definition(
            spec=spec,
            decl=p[1],
            param_decls=p[2],
            body=p[3])

    def p_function_definition_2(self, p):
        """ function_definition : declaration_specifiers id_declarator declaration_list_opt
compound_statement
        """
        spec = p[1]

        p[0] = self._build_function_definition(
            spec=spec,
            decl=p[2],
            param_decls=p[3],
            body=p[4])
```