**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Чувашский государственный университет имени И.Н. Ульянова»**

**Факультет информатики и вычислительной техники**

**Кафедра вычислительной техники**

***ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ***

**Расчетно-графическая работа (часть 2)**

**Синтаксически управляемая трансляция**

**Выполнил:**

студент группы ИВТ-41-21

Фамилия И.О.

**Руководитель:**

доцент Павлов Л.А.

Чебоксары 2024

**Оглавление**

[1. Контекстно-свободная грамматика языка 5](#_Toc174701594)

[2. Синтаксически управляемое определение и схема трансляции 6](#_Toc174701595)

[2.1. Семантический анализ (проверка типов) 6](#_Toc174701596)

[2.2. Генерация промежуточного кода 7](#_Toc174701597)

[2.3. Разработка СУО 8](#_Toc174701598)

[2.4. Разработка СУТ 9](#_Toc174701599)

[3. Структуры данных и алгоритмы 20](#_Toc174701600)

[4. Программная реализация СУ-трансляции 22](#_Toc174701601)

[Заключение 25](#_Toc174701602)

[Список использованной литературы 26](#_Toc174701603)

Задание к РГР

1. Проработать все теоретические вопросы, связанные с восходящей синтаксически управляемой трансляцией языка *DemoLang*, разработанного в 1-й части РГР, которые включают в себя разработку синтаксически управляемых определений и схем трансляции для реализации семантического анализа (проверки типов) и генерации промежуточного кода.

2. Разработать программный модуль восходящей синтаксически управляемой трансляции.

**Внимание!** В примере оформления отчета желтым цветом выделены различные мои примечания и рекомендации, которые не нужно включать в отчет.

Введение

***Цель работы*** – изучение методов синтаксически управляемой трансляции (СУ-трансляция), включающей в себя фазы синтаксического и семантического анализа и генерации промежуточного кода; получение практических навыков построения моделей СУ-трансляции; разработка СУ-транслятора в соответствии с заданным вариантом.

Основой СУ-трансляции является включение *семантических действий* в процесс синтаксического анализа. Часто применяемыми моделями СУ-трансляции являются синтаксически управляемые определения и синтаксически управляемые схемы трансляции.

*Синтаксически управляемое определение* (СУО) представляет собой контекстно-свободную грамматику, дополненную атрибутами и семантическими правилами.

Для нетерминалов выделяют два типа атрибутов: синтезируемые и наследуемые.

Терминал может иметь только синтезируемый атрибут, который является атрибутом токена, передаваемого лексическим анализатором.

*Синтаксически управляемая схема трансляции* (СУТ) представляет собой контекстно-свободную грамматику, дополненную программными фрагментами (*семантическими действиями*), вставленными в правые части продукций. Отличие СУТ от СУО заключается в том, что в СУТ явно определен порядок вычисления семантических правил, задаваемый порядком обхода дерева разбора, который, в свою очередь, определяется используемым методом синтаксического анализа.

В процессе разработки СУТ обычно приходится решать следующие задачи:

* выделить конструкции заданного языка, обрабатываемые на этапе синтаксического анализа, и определять для них соответствующую контекстно-свободную грамматику;
* определить принадлежность контекстно-свободной грамматики к классу *LL*(1)- или *LR*(1)-грамматик;
* при необходимости выполнить эквивалентные преобразования грамматики в *LL*(1)- или *LR*(1)-форму;
* построить синтаксически управляемую схему трансляции, включив в продукции грамматики семантические правила, реализующие проверку типов и генерацию промежуточного кода;
* построить таблицу разбора для реализации синтаксически управляемой трансляции;
* разработать структуру СУ-транслятора, определить его функции и алгоритмы и программно его реализовать.

# Контекстно-свободная грамматика языка

В соответствии с реализованным в 1-й части РГР множеством лексических классов синтаксис языка *DemoLang* в виде РБНФ, где в качестве терминалов выступают токены, можно представить следующим образом:

1. Программа = "**prog**" "**id**" "**;**" Блок.
2. Блок = "**var**" ОписПеременных "**start**" ПоследОператоров "**stop**".
3. ОписПеременных = ТипПеременных "**;**" { ТипПеременных "**;**" }.
4. ТипПеременных = СписокИмен "**:**" "**id**".
5. СписокИмен = "**id**" { "**,**" "**id**" }.
6. ПоследОператоров = Оператор "**;**" { Оператор "**;**" }.
7. Оператор = "**id**" "**:=**" Выражение.
8. Выражение = Терм { "**+**" Терм }.
9. Терм = Фактор { "**\***" Фактор }.
10. Фактор = "**num**" **|** "**id**" **|** "**(**" Выражение "**)**".

Построим по данной РБНФ формальную контекстно-свободную грамматику *G* = (*VT*, *VN*, *P*, *S*). Здесь *VT* – конечное множество *терминалов*; *VN* – конечное множество *нетерминалов*; *P* – конечное множество *продукций* вида *A* → β, где *A* – нетерминал левой части продукции, β – правая часть – строка, такая, что β ∈ (*VT* ∪ *VN*)\*; *S* ∈ *VN* – *начальный символ* грамматики.

Напомним приемы построения формальной грамматики по РБНФ при использовании таких метасимволов, как квадратные скобки «[» и «]» (условное вхождение, т. е. заключенная в них конструкция может отсутствовать) и фигурные скобки «{» и «}» (повторение, т. е. нуль или более повторений заключенной в них конструкции).

|  |  |  |
| --- | --- | --- |
| *Метасимволы* | *РБНФ* | *Формальная грамматика* |
| Условное вхождение | *S* = *A* [ *B* ] | *S* → *A* **|** *A B* |
| Повторение без разделителя | *L = A* { *A* } | *L* → *A* **|** *L* *A* для *LR*-грамматик  *L* → *A* **|** *A* *L* для *LL*-грамматик |
| Повторение с разделителем (запятая) | *L = A* { "**,**" *A* } | *L* → *A* **|** *L* **,** *A* для *LR*-грамматик  *L* → *A* **|** *A* **,** *L* для *LL*-грамматик |

Используем следующие соглашения об обозначениях. Терминалы представляются символами операций и пунктуации, а также строками, выделенными жирным шрифтом. Нетерминалы – прописными буквами или строками букв, выделенных курсивом. Чтобы не возникала неоднозначность в последовательностях букв, обозначающих терминалы или нетерминалы, для разделения символов в продукции используются пробелы. Символом ε обозначается пустая строка.

Учитывая принятые обозначения и выполнив ряд операций замены вхождений нетерминалов, можно построить следующий вариант формальной грамматики языка:

1. *S* → **prog** **id** **;** *Block*
2. *Block* → **var** *LstVar* **start** *LstStmt* **stop**
3. *LstVar* → *DeclVar* **;** **|** *LstVar* *DeclVar* **;**
4. *DeclVar* → *LstId* **: id**
5. *LstId* → **id** **|** *LstId* **,** **id**
6. *LstStmt* → *Stmt* **;** **|** *LstStmt* *Stmt* **;**
7. *Stmt* → **id** **ass** *Expr*
8. *Expr* → *Term* **|** *Expr* **+** *Term*
9. *Term* → *Factor* **|** *Term* **\*** *Factor*
10. *Factor* → **num** **|** **id** **|** **(** *Expr* **)**

Хотя данная грамматика и является *SLR*(1)-грамматикой и ориентирована на восходящий разбор, она не подходит для реализации синтаксически управляемой трансляции.

Продукции

4) *DeclVar* → *LstId* **: id**

5) *LstId* → **id** **|** *LstId* **,** **id**

не позволяют получить *L*-атрибутное СУО, поскольку идентификаторы порождаются нетерминалом *LstId*, но тип, определяемый именем типа **id**, в поддереве *LstId* дерева разбора еще не известен. Эта проблема решается соответствующим эквивалентным преобразованием этих продукций:

4) *DeclVar* → **id** *LstId*

5) *LstId* → **,** **id** *LstId* | **: id**

Теперь тип можно рассматривать как синтезируемый атрибут нетерминала *LstId*, который можно внести в таблицу символов каждого идентификатора, порождаемого *LstId*.

Для восходящего анализа непустой список *L* элементов лучше задавать продукциями *L* → *a* **|** *La* (в отличие от восходящего анализа, когда список задается как *L* → *a* **|** *aL*). Очевидно, что эти грамматики эквивалентны (формальный метод устранения левой рекурсии в продукциях *L* → *a* **|** *La* даст нам как результат продукции *L* → *a* **|** *aL*), однако порядок разбора для них отличается.

# Синтаксически управляемое определение и схема трансляции

## Семантический анализ (проверка типов)

***Семантические соглашения***.

1. Никакой идентификатор объекта (переменная и т.п.) в программном блоке не должен быть объявлен более одного раза.

2. Определяющим вхождениям идентификатора должны соответствовать их использующие вхождения.

3. Язык строго типизированный, отсутствует неявное преобразование типов. Эквивалентность типов – именная, т.е. два типа эквивалентны, если они объявлены с помощью одного и того же имени типа.

4. Предопределенные типы: *integer* (4), *real* (8). В скобках указаны размеры памяти в байтах для представления объекта соответствующего типа (размер типа). Адрес размещения объекта в памяти должен быть кратен размеру типа. Выделение памяти требуется только для переменных.

5. В продукции *S* → **prog** **id** **;** *Block* (строка 1) имя программы **id**, очевидно, не должно встречаться ни в какой другой конструкции языка.

6. В продукциях *DeclVar* → **id** *LstId* **;** и *LstId* → **,** **id** *LstId* (строки 4 и 5) в качестве идентификатора **id** может быть только имя переменной.

7. В продукции *LstId* → **: id** (строка 5) в качестве идентификатора **id** может быть только имя типа.

8. В продукции *Stmt* → **id** **ass** *Expr* **;** (строка 7) для оператора присваивания в качестве идентификатора **id** может быть только имя переменной.

9. Во всех продукциях, описывающих бинарные операции (**+**, **\***), типы операндов должны совпадать.

***Выражения типов***.

Типы имеют структуру, представленную с использованием выражения типов:

а) фундаментальный тип является выражением типа, фундаментальные типы включают предопределенные типы *integer*, *real*, а также *void* (отсутствие типа);

б) выражение типа может быть образовано путем применения конструктора типа к выражению типа, конструктор типа предназначен для создания нового типа на основе другого типа.

В языке *DemoLang* все типы простые (предопределенные), поэтому нет необходимости в конструкторах типа. У Вас будет свой конструктор типа (массив, запись, процедура или функция) в зависимости от варианта задания.

Как уже рассматривалось в 1-й части РГР, в языке *DemoLang* идентификатор может быть именем программы, именем типа или именем переменной. Поскольку предполагается, что идентификаторы и числа будут храниться в одной таблице символов, определены следующие обозначения категорий: *catProgName* – имя программы, *catTypeName* – имя типа, *catVarName* – имя переменной, *catConst* – число, *catNoCat* – отсутствие категории.

В работе реализуется структурная эквивалентность типов.

Если Вы хотите применить именную эквивалентность типов, то для любых объявлений вида

**var**

MyArr: **array**[1..20] **of** integer;

MyPnt: ^Integer;

надо будет при трансляции реализовать создание неявных имен типов, чтобы эти объявления трактовалось следующим образом (T1 и T2 – неявные имена типов, создаваемые компилятором, чтобы тип задавался только именем):

**type**

T1 = **array**[1..20] **of** integer;

T2 = ^Integer;

**var**

MyArr: T1;

MyPnt: T2;

Другой способ – определить синтаксис языка так, чтобы в разделе объявления типов переменных (например, раздел **var**) тип можно было задавать только именем типа. Тогда разделу **var** обязательно должен предшествовать раздел для именования всех используемых типов (например, раздел **type**).

## Генерация промежуточного кода

В качестве языка промежуточного представления выбран трехадресный код, реализуемый в виде четверок (тетрад). Четверка представляет собой запись с полями *op*, *arg*1, *arg*2 и *result*. Поле *op* содержит код операции. Поля *arg*1, *arg*2 и *result* содержат указатели на соответствующие записи таблицы символов. Временные имена вносятся в таблицу символов при их создании.

Рекомендуется для вычисления логических выражений реализовать подход, основанный на использовании команд условных и безусловных переходов, которые в зависимости от логического условия передают управление в ту или иную позицию кода (само значение не вычисляется, в коде отсутствуют логические операции). Такой подход позволяет достаточно легко реализовать сокращенное вычисление логических выражений. Для реализации однопроходной трансляции логических выражений и управляющих операторов (в языке *DemoLang* их нет) используйте метод обратных поправок. Реализуется инкрементная трансляция, которая заключается в том, что формируется единый поток генерируемых трехадресных команд в некотором глобальном массиве или файле.

Перечень трехадресных команд:

1. Команда присваивания вида *x* := *y* *op* *z*, где *op* – бинарная арифметическая операция,  
*y* и *z* – переменные или числа, *x* – переменная. Код операции *op* (атрибут токена **+** или **\***) размещается в поле *op* тетрады, *y* в *arg*1, *z* в *arg*2 и *x* в *result*. Коды операций определены в 1-й части РГР: *opAdd* (сложение), *opSub* (вычитание), *opMult* (умножение), *opDiv* (деление).

2. Команда копирования вида *x* := *y*, где *y* – переменная или число, *x* – переменная. Код операции копирования (обозначим *opAss*) размещается в поле *op* тетрады, *y* в *arg*1 и *x* в *result*, *arg*2 не используется.

Вы должны определить свой перечень необходимых трехадресных команд. У Вас обязательно добавятся команды условного и безусловного перехода, присваивания вида *x* := **true** и *x* := **false**, возможно, и другие.

## Разработка СУО

Сначала рассмотрим объекты, процедуры и функции, которые будут использоваться в семантических правилах.

***Глобальные объекты*:**

*TblIde* – таблица символов (односвязный список элементов), запись (узел) таблицы имеет следующую структуру полей:

*Lex* – лексема идентификатора или числа;

*Cat* – категория символа (*catProgName*, *catTypeName*, *catVarName*, *catConst*, *catNoCat*);

*Type* – тип, в свою очередь состоит из полей:

*TypeCode* – код типа данных (*typeVoid*, *typeInt*, *typeReal*),

*Width* – размер типа (*wInt* = 4, *wReal* = 8).

*Addr* – адрес размещения объекта в памяти (для переменных);

*NextEl* – указатель на следующий элемент таблицы символов.

Целью такого разбиения поля *Type* на подполя является сокращение числа атрибутов при реализации синтаксически управляемой трансляции (вместо нескольких атрибутов будет один атрибут *type*). Более того, для разных типов могут понадобиться и другие сведения о типе. Для типа диапазон нужны значения *Low* (нижняя граница диапазона) и *High* (верхняя граница диапазона). Для типа массив надо знать число размерностей, данные о каждой размерности. Для типа запись понадобится ссылка на данные о полях записи и т. д. Представление всех этих сведений в качестве отдельных атрибутов существенно усложнит семантические правила.

По сравнению со структурой, рассмотренной при проектировании сканера, для языка *DemoLang* для реализации трансляции понадобилось добавление только одного поля *Addr*.

Для удобства память разбита на две части:

1) память данных (*DataMem*), в которой размещаются переменные и константы, по сути это массив байтов;

2) память команд (*InstrMem*), в которой размещаются трехадресные команды при инкрементной трансляции, по сути это массив трехадресных команд.

*NextAddr* – адрес очередной свободной ячейки в памяти данных.

*NextInstr* – адрес очередной генерируемой команды в памяти команд.

***Процедуры и функции*:**

Процедура *AddType*(*pnt*, *Cat*, *Type*) устанавливает категорию *Cat* и тип *Type* в записи таблицы символов, на которую указывает *pnt*, а также размещает переменные в памяти с выравниванием адресов. Для простых типов назначаются адреса, которые кратны размеру в байтах для типа данных.

**procedure** *AddType* (*pnt*, *Cat*, *Type*)

**begin**

*pnt***.***Cat* := *Cat* //установка категории

*pnt***.***Type*:= *Type* //установка типа

**if** *Cat* = *catVarName* **then**

**begin** //размещение в памяти

**while** *NextAddr* **mod** *Type***.***Width* ≠ 0 **do** //выравнивание адреса

*NextAddr* := *NextAddr* + 1

*pnt***.***Addr*:= *NextAddr* //установка адреса памяти

*NextAddr* := *NextAddr* + *Type***.***Width* //адрес очередной свободной памяти

**end else** *pnt***.***Addr*:= –1

**end**

Здесь в качестве примера приведен только алгоритм процедуры *AddType*. Вы должны привести алгоритмы всех процедур и функций (не забудьте про конструкторы типов), за исключением простейших, которые только возвращают или устанавливают значение какого-либо поля, или реализуют операции для типовых структур данных типа стеков и очередей. Можно также не раскрывать процедуры формирования сообщений об ошибках (в данном примере это процедура *Type\_Error*).

В языке *DemoLang* только простые типы. У Вас будут производные типы массив или запись (структура). Для массива достаточно выровнять первый элемент, а для записи должно выравниваться каждое поле.

Функция *NewTemp*(*Type*) создает новую временную переменную, добавляет ее в таблицу символов, устанавливает тип (*Type*), категорию (*catVarName*) и возвращает указатель на соответствующую запись.

Вам для реализации обратных поправок при генерации кода понадобятся:

Функция *MakeList*(*i*) создает новый одноэлементный список, состоящий только из *i* (индекс в массиве команд), возвращает указатель на созданный список.

Функция *Merge*(*p*1, *p*2) объединяет списки, на которые указывают *p*1 и *p*2, возвращает указатель на объединенный список

Процедура *BackPatch*(*p*, *i*) устанавливает *i* в качестве целевой метки в каждую команду из списка, на который указывает *p*, после этого время жизни списка завершается.

Процедура *Gen*(*op*, *arg*1, *arg*2, *result*) формирует трехадресную команду в виде четверки, устанавливая соответствующие значения полей, а также инкрементно добавляет новую команду к последовательности сформированных к данному моменту команд (инкрементирует *NextInstr*). Отсутствие какого-либо параметра обозначается как –1.

Процедура *Type\_Error* формируют соответствующие сообщения об ошибках:

*Type\_Error*(1) – повторное объявление идентификатора (соглашение №1);

*Type\_Error*(2) – идентификатор не является именем типа (соглашение №7);

*Type\_Error*(3) – идентификатор не является именем переменной (соглашения №6 и №8);

*Type\_Error*(4) – несовместимость типов (соглашение №9).

*Type\_Error*(5) – не объявлен идентификатор (соглашение №2).

Предполагается, что после обнаружения ошибки и выдачи соответствующего сообщения процесс трансляции немедленно прекращается (для компактности записи в семантических действиях это не отражено).

***Атрибуты*:**

Нетерминал *LstId* имеет синтезируемый атрибут *type* (тип). Нетерминалы *Expr*, *Term*, *Factor* имеют синтезируемые атрибуты *type* (тип), *addr* (указатель на запись в таблице символов).

Атрибут *pnt* терминалов **id** и **num** указывает на соответствующую запись в таблице символов, является атрибутом токена, формируемого сканером. Атрибутом *op* терминалов **+** и **\*** является код операции (также формируемый сканером как атрибут соответствующего токена).

Разработанное СУО представлено в табл. 1 (первые два столбца).

В приведенном СУО у нетерминалов *LstStmt* и *Stmt* нет атрибутов типа *nextlist* (указатель на список команд переходов), необходимых для метода обратных поправок, поскольку в языке *DemoLang* нет управляющих операторов. У вас они будут.

## Разработка СУТ

СУТ представляет собой контекстно-свободную грамматику, дополненную программными фрагментами (семантическими действиями), вставленными в правые части продукций в виде специальных символов действий. Позиция выполняемого действия обычно указывается фигурными скобками. Наше СУО является *L*-атрибутным, поэтому преобразование в СУТ заключается в размещении действий для вычисления наследуемых атрибутов перед соответствующим нетерминалом, а синтезируемых атрибутов – в конце продукции. Поскольку СУТ ориентирована на восходящий синтаксический анализ (действия должны выполняться в процессе свертки правой части продукции), вместо вставленных действий по вычислению наследуемых атрибутов следует добавить в СУТ нетерминалы-маркеры с соответствующими ε-продукциями, при свертке которых и будут выполняться соответствующие семантические действия.

Важным является назначение памяти для хранения значений атрибутов в процессе трансляции. Можно разместить атрибуты в стеке синтаксического анализатора (расширив соответствующим образом структуру элемента стека) или использовать специальный стек или несколько стеков (например, отдельные стеки для синтезируемых и наследуемых атрибутов) для хранения значений атрибутов в течение времени их жизни.

Выбран вариант размещения атрибутов в специальном стеке атрибутов (глобальная переменная *AttrSt*). Обозначения используемых стековых операций: процедура *Push*(*x*, *S*) размещает значение *x* в стеке *S*; функция *Pop*(*S*) исключает элемент из вершины стека *S* и возвращает его значение; функция *StackTop*(*S*) возвращает значение элемента из вершины стека *S* без его исключения. Результаты детализации представлены в табл. 1 (столбцы 3 и 4).

Таблица 1. СУО и СУТ для проверки типов и трансляции в трехадресный код языка *DemoLang*

| Продукция с действиями | Семантические правила | Семантические действия | Символ  действия |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| *S* → **prog** **id** {*A*1} **;** *Block* | **if id.***pnt***.***Cat* ≠ *catNoCat* **then** *Type\_Error*(1)  *AddType*(**id.***pnt, catProgName*, *void*) | **if id.***pnt***.***Cat* ≠ *catNoCat* **then** *Type\_Error*(1)  *AddType*(**id.***pnt, catProgName*, *void*) | *A*1 |
| *Block* → **var** *LstVar* **start** *LstStmt* **stop** |  |  |  |
| *LstVar* → *DeclVar* **;** |  |  |  |
| *LstVar* → *LstVar*1 *DeclVar* **;** |  |  |  |
| *DeclVar* → **id** {*A*2}*LstId* {*A*3} | **if id.***pnt***.***Cat* ≠ *catNoCat* **then** *Type\_Error*(1) | **if id.***pnt***.***Cat* ≠ *catNoCat* **then** *Type\_Error*(1)  *Push*(**id.***pnt*, *AttrSt*) //атрибут **id.***pnt* | *A*2 |
| *AddType*(**id.***pnt, catVarName*, *LstId***.***type*) | *t*1 := *Pop*(*AttrSt*) //атрибут *LstId***.***type*  *t*2 := *Pop*(*AttrSt*) //атрибут **id.***pnt*  *AddType*(*t*2*, catVarName*, *t*1) | *A*3 |
| *LstId* → **,** **id** {*A*2} *LstId*1 {*A*4} | **if id.***pnt***.***Cat* ≠ *catNoCat* **then** *Type\_Error*(1) | **if id.***pnt***.***Cat* ≠ *catNoCat* **then** *Type\_Error*(1)  *Push*(**id.***pnt*, *AttrSt*) //атрибут **id.***pnt* | *A*2 |
| *AddType* (**id.***pnt, catVarName*, *LstId*1**.***type*)  *LstId***.***type* := *LstId*1**.***type* | *t*1 := *Pop*(*AttrSt*) //атрибут *LstId*1**.***type*  *t*2 := *Pop*(*AttrSt*) //атрибут **id.***pnt*  *AddType*(*t*2*, catVarName*, *t*1)  *Push*(*t*1, *AttrSt*) //атрибут *LstId***.***type* | *A*4 |
| *LstId* → **:** **id** {*A*5} | **if** **id.***pnt***.***Cat* ≠ *catTypeName* **then** *Type\_Error*(2)  *LstId***.***type* := **id.***pnt***.***Type* | **if** **id.***pnt***.***Cat* ≠ *catTypeName* **then** *Type\_Error*(2)  *Push*(**id.***pnt***.***Type*, *AttrSt*) //атрибут *LstId***.***type* | *A*5 |
| *LstStmt* → *Stmt* **;** |  |  |  |
| *LstStmt* → *LstStmt*1 *Stmt* **;** |  |  |  |
| *Stmt* → **id** {*A*6} **ass** *Expr* {*A*7} | **if id.***pnt***.***Cat* = *catNoCat* **then** *Type\_Error*(5)  **if** **id.***pnt***.***Cat* ≠ *catVarName* **then** *Type\_Error*(3) | **if id.***pnt***.***Cat* = *catNoCat* **then** *Type\_Error*(5)  **if** **id.***pnt***.***Cat* ≠ *catVarName* **then** *Type\_Error*(3)  *Push*(**id.***pnt*, *AttrSt*) //атрибут **id.***pnt* | *A*6 |
| **if** **id.***pnt***.***Type* ≠ *Expr***.***type* **then** *Type\_Error*(4)  *Gen*(*opAss*, *Expr***.***addr*, –1, **id.***pnt*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Expr***.***addr*  *t*2 := *Pop*(*AttrSt*) //атрибут *Expr***.***type*  *t*3 := *Pop*(*AttrSt*) //атрибут **id.***pnt*  **if** *t*3**.***Type* ≠ *t*2 **then** *Type\_Error*(4)  *Gen*(*opAss*, *t*1, –1, *t*3) | *A*7 |
| *Expr* → *Term* | *Expr***.***type* := *Term***.***type*  *Expr***.***addr* := *Term***.***addr* | правила копирования, в верхней части стека уже находятся соответствующие атрибуты, действия со стеком не нужны |  |
| *Expr* → *Expr*1 **+** {*A*8} *Term* {*A*9} |  | *Push*(**+.***op*, *AttrSt*) //атрибут **+.***op* | *A*8 |
| **if** *Expr*1**.***type* ≠ *Term***.***type* **then** *Type\_Error*(4)  *Expr***.***type* := *Term***.***type*  *Expr***.***addr* := *NewTemp*(*Term***.***type*)  *Gen*(**+.***op*, *Expr*1**.***addr*, *Term***.***addr*, *Expr***.***addr*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Term***.***addr*  *t*2 := *Pop*(*AttrSt*) //атрибут *Term***.***type*  *t*3 := *Pop*(*AttrSt*) //атрибут **+.***op*  *t*4 := *Pop*(*AttrSt*) //атрибут *Expr*1**.***addr*  *t*5 := *Pop*(*AttrSt*) //атрибут *Expr*1**.***type*  **if** *t*5 ≠ *t*2 **then** *Type\_Error*(4)  *Push*(*t*2, *AttrSt*) //атрибут *Expr***.***type*  *t*6 := *NewTemp*(*t*2)  *Push*(*t*6, *AttrSt*) //атрибут *Expr***.***addr*  *Gen*(*t*3, *t*4, *t*1, *t*6) | *A*9 |
| *Term* → *Factor* | *Term***.***type* := *Factor***.***type*  *Term***.***addr* := *Factor***.***addr* | правила копирования, в верхней части стека уже находятся соответствующие атрибуты, действия со стеком не нужны |  |
| *Term* → *Term*1 **\*** {*A*10} *Factor* {*A*11} |  | *Push*(**\*.***op*, *AttrSt*) //атрибут **\*.***op* | *A*10 |
| **if** *Term*1**.***type* ≠ *Factor***.***type* **then** *Type\_Error*(4)  *Term***.***type* := *Factor***.***type*  *Term***.***addr* := *NewTemp*(*Factor***.***type*, *Factor***.***addr***.***Width*)  *Gen*(**\*.***op*, *Term*1**.***addr*, *Factor***.***addr*, *Term***.***addr*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Factor***.***addr*  *t*2 := *Pop*(*AttrSt*) //атрибут *Factor***.***type*  *t*3 := *Pop*(*AttrSt*) //атрибут **\*.***op*  *t*4 := *Pop*(*AttrSt*) //атрибут *Term*1**.***addr*  *t*5 := *Pop*(*AttrSt*) //атрибут *Term*1**.***type*  **if** *t*5 ≠ *t*2 **then** *Type\_Error*(4)  *Push*(*t*2, *AttrSt*) //атрибут *Term***.***type*  *t*6 := *NewTemp*(*t*2)  *Push*(*t*6, *AttrSt*) //атрибут *Term***.***addr*  *Gen*(*t*3, *t*4, *t*1, *t*6) | *A*11 |
| *Factor* → **num** {*A*12} | *Factor***.***type* := **num.***pnt***.***Type***.***type*  *Factor***.***addr* := **num.***pnt* | *Push*(**num.***pnt***.***Type*, *AttrSt*) //атрибут *Factor***.***type*  *Push*(**num.***pnt*, *AttrSt*) //атрибут *Factor***.***addr* | *A*12 |
| *Factor* → **id** {*A*13} | **if id.***pnt***.***Cat* = *catNoCat* **then** *Type\_Error*(5)  **if** **id.***pnt***.***Cat* ≠ *catVarName* **then** *Type\_Error*(3)  *Factor***.***type* := **id.***pnt***.***Type*  *Factor***.***addr* := **id.***pnt* | **if id.***pnt***.***Cat* = *catNoCat* **then** *Type\_Error*(5)  **if** **id.***pnt***.***Cat* ≠ *catVarName* **then** *Type\_Error*(3)  *Push*(**id.***pnt***.***Type*, *AttrSt*) //атрибут *Factor***.***type*  *Push*(**id.***pnt*, *AttrSt*) //атрибут *Factor***.***addr* | *A*13 |
| *Factor* → **(** *Expr* **)** | *Factor***.***type* := *Expr***.***type*  *Factor***.***addr* := *Expr***.***addr* | правила копирования, в верхней части стека уже находятся соответствующие атрибуты, действия со стеком не нужны |  |

Анализ выполняемых семантических действий в таблице показывает, что действия *A*8 и *A*10 отличаются только тем, что в *A*8 заносится в стек **+.***op*, а в *A*10 – **\*.***op*. А это не что иное, как атрибут соответствующего токена. Поэтому для реализации этих действий достаточно одной процедуры (назовем ее *A*8). Чтобы была видна семантика этих действий, их нумерация сохранена, т. е. в схеме трансляции действие *A*10 остается, но для его выполнения будет вызываться *A*8.

Аналогично действия *A*9 и *A*11 отличаются только токенами (в *A*9 атрибут токена **+.***op*, а в *A*11 – **\*.***op*). Поэтому также достаточно одной процедуры *A*9, которая будет вызываться и для реализации действия *A*11.

**Комментарии к табл. 1.**

В продукции *S* → **prog** **id** {*A*1} **;** *Block* **stop** символ действия *A*1 можно было бы разместить в конце правой части, но тогда для последующего использования пришлось бы сохранить **id***.pnt* в стеке атрибутов и разместить соответствующий символ действия именно в той позиции, где сейчас размещен символ *A*1. Поэтому перенос *A*1 в конец правой части продукции не имеет смысла.

Для ряда продукций семантические правила разбиты на 2 группы, чтобы реализовать действия в разных позициях правой части продукции. Например, в продукции

*Stmt* → **id** {*A*6} **ass** *Expr* {*A*7}

действие *A*6 проверяет, является ли **id** именем переменной, если нет, то это семантическая ошибка. В случае обнаружения ошибки следует прекратить последующий разбор нетерминала *Expr*, т. е. действие *A*6 выполнить сразу после получения **id** от сканера. Заметим, что даже если не проверять на наличие ошибки (перенести проверку в действие *A*7), все равно в этой позиции должно быть действие по сохранению в стеке атрибутов указателя **id.***pnt* на соответствующую запись в таблице символов.

Подобные комментарии к табл. 1 можете не включать в отчет по РГР. Главное, чтобы Вы могли обосновать размещение символов действий при защите РГР.

Получена следующая СУТ:

*S* → **prog** **id** {*A*1} **;** *Block*

*Block* → **var** *LstVar* **start** *LstStmt* **stop**

*LstVar* → *DeclVar* **;** **|** *LstVar* *DeclVar* **;**

*DeclVar* → **id** {*A*2} *LstId* {*A*3}

*LstId* → **,** **id** {*A*2} *LstId* {*A*4} | **: id** {*A*5}

*LstStmt* → *Stmt* **;** **|** *LstStmt* *Stmt* **;**

*Stmt* → **id** {*A*6} **ass** *Expr* {*A*7}

*Expr* → *Term* **|** *Expr* **+** {*A*8} *Term* {*A*9}

*Term* → *Factor* **|** *Term* **\*** {*A*10} *Factor* {*A*11}

*Factor* → **num** {*A*12} **|** **id** {*A*13} **|** **(** *Expr* **)**

Поскольку для восходящей трансляции все действия должны находиться в конце продукции, построенная СУТ не подходит. Следует добавить нетерминалы-маркеры вместо каждого вставленного действия [1; 6; 11]. В результате получим СУТ, которая может быть использована для реализации СУО в процессе восходящего синтаксического анализа:

1. *S* → **prog** **id** *A* **;** *Block*
2. *Block* → **var** *LstVar* **start** *LstStmt* **stop**
3. *LstVar* → *DeclVar* **;**
4. *LstVar* → *LstVar* *DeclVar* **;**
5. *DeclVar* → **id** *B* *LstId* {*A*3}
6. *LstId* → **,** **id** *B* *LstId* {*A*4}
7. *LstId* → **: id** {*A*5}
8. *LstStmt* → *Stmt* **;**
9. *LstStmt* → *LstStmt* *Stmt* **;**
10. *Stmt* → **id** *C* **ass** *Expr* {*A*7}
11. *Expr* → *Term*
12. *Expr* → *Expr* **+** *D* *Term* {*A*9}
13. *Term* → *Factor*
14. *Term* → *Term* **\*** *E* *Factor* {*A*11}
15. *Factor* → **num** {*A*12}
16. *Factor* → **id** {*A*13}
17. *Factor* → **(** *Expr* **)**
18. *A* → ε {*A*1}
19. *B* → ε {*A*2}
20. *C* → ε {*A*6}
21. *D* → ε {*A*8}
22. *E* → ε {*A*10}

*SLR*(1)-метод требует вычисления функции

*Follow*(*X*) = {*a*⎪}, *a*∈*VT*, α∈*VT*\*, β∈(*VT* ∪ *VN*)\*,

для каждого нетерминала *X* грамматики. Эта функция определяет множество терминалов, которые могут следовать непосредственно за нетерминалом *X* в какой-либо сентенциальной форме, выводимой из начального нетерминала *S*. Формальный процесс вычисления функции *Follow* подробно изложен в [5] при рассмотрении *LL*(1)-грамматик.

Структура и алгоритм построения *LR*-таблицы разбора рассмотрены в [3; 4]. Таблица разбора представляет собой прямоугольную матрицу, состоящую из столбцов для каждого терминала и нетерминала (символам действий столбцы не сопоставляются) и строк, соответствующих каждому состоянию, в котором может находиться анализатор. Комбинация состояния в вершине стека анализатора и текущего входного символа используется в качестве индекса таблицы разбора и по ее элементу определяется дальнейшее действие. Элементы таблицы бывают четырех типов:

1. *Элементы переноса*. Будем записывать в виде *Si*, что означает: поместить в стек анализатора состояние *i* и перейти в состояние *i*. Если входной символ является терминалом, принять его.

2. *Элементы свертки*. Будем записывать в виде *Rj* (если нет символа действий) или *Rj*{*A*} (если есть действие *A*). Запись означает: выполнить свертку для продукции с номером *j*, т. е. допустив, что *n* есть число символов в правой части *j*-й продукции, удалить *n* элементов из стека анализатора и перейти к состоянию, находящемуся в вершине стека. Если в конце правой части продукции есть символ действия, выполнить действие. Нетерминал в левой части *j*-й продукции считать входным символом на следующем шаге разбора.

3. *Элементы остановки* (stop), которые указывают на успешное завершение разбора входной строки.

4. *Элементы ошибок*. Это пустые ячейки в таблице разбора и соответствуют синтаксическим ошибкам. Семантика ошибки легко определяется состоянием анализатора.

Для вычисления множеств *Follow*, построения *SLR*(1)-автомата (табл. 2) и соответствующей *SLR*(1)-таблицы разбора (табл. 3) использован генератор таблиц разбора (приложение TblGenerator.exe).

Перед использованием генератора в исходную грамматику не забудьте в первую продукцию добавить маркер конца ввода **eof** как терминал!

Таблица 2. *SLR*(1)-автомат

| № сост | Конфигурация | Символ перехода (свертки) | Состояние- преемник | Свертка {действие} |
| --- | --- | --- | --- | --- |
| 1 | S = • "prog" "id" A ";" Block "eof" | prog | 2 |  |
| 2 | S = "prog" • "id" A ";" Block "eof" | id | 3 |  |
| 3 | S = "prog" "id" • A ";" Block "eof" | A | 4 |  |
|  | A = {A1} • | ; |  | R18 {A1} |
| 4 | S = "prog" "id" A • ";" Block "eof" | ; | 5 |  |
| 5 | S = "prog" "id" A ";" • Block "eof" | Block | 6 |  |
|  | Block = • "var" LstVar "start" LstStmt "stop" | var | 7 |  |
| 6 | S = "prog" "id" A ";" Block • "eof" | eof | stop |  |
| 7 | Block = "var" • LstVar "start" LstStmt "stop" | LstVar | 8 |  |
|  | LstVar = • DeclVar ";" | DeclVar | 9 |  |
|  | LstVar = • LstVar DeclVar ";" | LstVar | 8 |  |
|  | DeclVar = • "id" B LstId {A3} | id | 10 |  |
| 8 | Block = "var" LstVar • "start" LstStmt "stop" | start | 11 |  |
|  | LstVar = LstVar • DeclVar ";" | DeclVar | 12 |  |
|  | DeclVar = • "id" B LstId {A3} | id | 10 |  |
| 9 | LstVar = DeclVar • ";" | ; | 13 |  |
| 10 | DeclVar = "id" • B LstId {A3} | B | 14 |  |
|  | B = {A2} • | ,, : |  | R19 {A2} |
| 11 | Block = "var" LstVar "start" • LstStmt "stop" | LstStmt | 15 |  |
|  | LstStmt = • Stmt ";" | Stmt | 16 |  |
|  | LstStmt = • LstStmt Stmt ";" | LstStmt | 15 |  |
|  | Stmt = • "id" C "ass" Expr {A7} | id | 17 |  |
| 12 | LstVar = LstVar DeclVar • ";" | ; | 18 |  |
| 13 | LstVar = DeclVar ";" • | id, start |  | R3 |
| 14 | DeclVar = "id" B • LstId {A3} | LstId | 19 |  |
|  | LstId = • "," "id" B LstId {A4} | , | 20 |  |
|  | LstId = • ":" "id" {A5} | : | 21 |  |
| 15 | Block = "var" LstVar "start" LstStmt • "stop" | stop | 22 |  |
|  | LstStmt = LstStmt • Stmt ";" | Stmt | 23 |  |
|  | Stmt = • "id" C "ass" Expr {A7} | id | 17 |  |
| 16 | LstStmt = Stmt • ";" | ; | 24 |  |
| 17 | Stmt = "id" • C "ass" Expr {A7} | C | 25 |  |
|  | C = {A6} • | ass |  | R20 {A6} |
| 18 | LstVar = LstVar DeclVar ";" • | id, start |  | R4 |
| 19 | DeclVar = "id" B LstId {A3} • | ; |  | R5 {A3} |
| 20 | LstId = "," • "id" B LstId {A4} | id | 26 |  |
| 21 | LstId = ":" • "id" {A5} | id | 27 |  |
| 22 | Block = "var" LstVar "start" LstStmt "stop" • | eof |  | R2 |
| 23 | LstStmt = LstStmt Stmt • ";" | ; | 28 |  |
| 24 | LstStmt = Stmt ";" • | id, stop |  | R8 |
| 25 | Stmt = "id" C • "ass" Expr {A7} | ass | 29 |  |
| 26 | LstId = "," "id" • B LstId {A4} | B | 30 |  |
|  | B = {A2} • | ,, : |  | R19 {A2} |
| 27 | LstId = ":" "id" {A5} • | ; |  | R7 {A5} |
| 28 | LstStmt = LstStmt Stmt ";" • | id, stop |  | R9 |
| 29 | Stmt = "id" C "ass" • Expr {A7} | Expr | 31 |  |
|  | Expr = • Term | Term | 32 |  |
|  | Expr = • Expr "+" D Term {A9} | Expr | 31 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "\*" E Factor {A11} | Term | 32 |  |
|  | Factor = • "num" {A12} | num | 34 |  |
|  | Factor = • "id" {A13} | id | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
| 30 | LstId = "," "id" B • LstId {A4} | LstId | 37 |  |
|  | LstId = • "," "id" B LstId {A4} | , | 20 |  |
|  | LstId = • ":" "id" {A5} | : | 21 |  |
| 31 | Stmt = "id" C "ass" Expr {A7} • | ; |  | R10 {A7} |
|  | Expr = Expr • "+" D Term {A9} | + | 38 |  |
| 32 | Expr = Term • | ;, +, ) |  | R11 |
|  | Term = Term • "\*" E Factor {A11} | \* | 39 |  |
| 33 | Term = Factor • | ;, +, \*, ) |  | R13 |
| 34 | Factor = "num" {A12} • | ;, +, \*, ) |  | R15 {A12} |
| 35 | Factor = "id" {A13} • | ;, +, \*, ) |  | R16 {A13} |
| 36 | Factor = "(" • Expr ")" | Expr | 40 |  |
|  | Expr = • Term | Term | 32 |  |
|  | Expr = • Expr "+" D Term {A9} | Expr | 40 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "\*" E Factor {A11} | Term | 32 |  |
|  | Factor = • "num" {A12} | num | 34 |  |
|  | Factor = • "id" {A13} | id | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
| 37 | LstId = "," "id" B LstId {A4} • | ; |  | R6 {A4} |
| 38 | Expr = Expr "+" • D Term {A9} | D | 41 |  |
|  | D = {A8} • | id, num, ( |  | R21 {A8} |
| 39 | Term = Term "\*" • E Factor {A11} | E | 42 |  |
|  | E = {A10} • | id, num, ( |  | R22 {A10} |
| 40 | Factor = "(" Expr • ")" | ) | 43 |  |
|  | Expr = Expr • "+" D Term {A9} | + | 38 |  |
| 41 | Expr = Expr "+" D • Term {A9} | Term | 44 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "\*" E Factor {A11} | Term | 44 |  |
|  | Factor = • "num" {A12} | num | 34 |  |
|  | Factor = • "id" {A13} | id | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
| 42 | Term = Term "\*" E • Factor {A11} | Factor | 45 |  |
|  | Factor = • "num" {A12} | num | 34 |  |
|  | Factor = • "id" {A13} | id | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
| 43 | Factor = "(" Expr ")" • | ;, +, \*, ) |  | R17 |
| 44 | Expr = Expr "+" D Term {A9} • | ;, +, ) |  | R12 {A9} |
|  | Term = Term • "\*" E Factor {A11} | \* | 39 |  |
| 45 | Term = Term "\*" E Factor {A11} • | ;, +, \*, ) |  | R14 {A11} |
| SLR(1)-грамматика | | | | |
| **Множества Follow нетерминалов** | | | | |
| Follow(S)={}; Follow(A)={;}; Follow(Block)={eof}; Follow(LstVar)={id,start}; Follow(LstStmt)={id,stop}; Follow(DeclVar)={;}; Follow(B)={,,:}; Follow(LstId)={;}; Follow(Stmt)={;}; Follow(C)={ass}; Follow(Expr)={;,+,)}; Follow(Term)={;,+,\*,)}; Follow(D)={id,num,(}; Follow(Factor)={;,+,\*,)}; Follow(E)={id,num,(}; | | | | |

Таблица 3. *SLR*(1)-таблица разбора

| № сост | A | Block | LstVar | LstStmt | DeclVar | B | LstId | Stmt | C | Expr | Term | D | Factor | E | "prog" | "id" | ";" | "eof" | "var" | "start" | "stop" | "," | ":" | "ass" | "+" | "\*" | "num" | "(" | ")" |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S3 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 | S4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R18{A1} |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S5 |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  | S6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S7 |  |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | stop |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  |  | S8 |  | S9 |  |  |  |  |  |  |  |  |  |  | S10 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  | S12 |  |  |  |  |  |  |  |  |  |  | S10 |  |  |  | S11 |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S13 |  |  |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  | S14 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R19{A2} | R19{A2} |  |  |  |  |  |  |
| 11 |  |  |  | S15 |  |  |  | S16 |  |  |  |  |  |  |  | S17 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S18 |  |  |  |  |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R3 |  |  |  | R3 |  |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  | S19 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S20 | S21 |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |  | S23 |  |  |  |  |  |  |  | S17 |  |  |  |  | S22 |  |  |  |  |  |  |  |  |
| 16 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S24 |  |  |  |  |  |  |  |  |  |  |  |  |
| 17 |  |  |  |  |  |  |  |  | S25 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R20{A6} |  |  |  |  |  |
| 18 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R4 |  |  |  | R4 |  |  |  |  |  |  |  |  |  |
| 19 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R5{A3} |  |  |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S26 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 21 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S27 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 22 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R2 |  |  |  |  |  |  |  |  |  |  |  |
| 23 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S28 |  |  |  |  |  |  |  |  |  |  |  |  |
| 24 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R8 |  |  |  |  | R8 |  |  |  |  |  |  |  |  |
| 25 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S29 |  |  |  |  |  |
| 26 |  |  |  |  |  | S30 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R19{A2} | R19{A2} |  |  |  |  |  |  |
| 27 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R7{A5} |  |  |  |  |  |  |  |  |  |  |  |  |
| 28 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R9 |  |  |  |  | R9 |  |  |  |  |  |  |  |  |
| 29 |  |  |  |  |  |  |  |  |  | S31 | S32 |  | S33 |  |  | S35 |  |  |  |  |  |  |  |  |  |  | S34 | S36 |  |
| 30 |  |  |  |  |  |  | S37 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S20 | S21 |  |  |  |  |  |  |
| 31 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R10{A7} |  |  |  |  |  |  |  | S38 |  |  |  |  |
| 32 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R11 |  |  |  |  |  |  |  | R11 | S39 |  |  | R11 |
| 33 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R13 |  |  |  |  |  |  |  | R13 | R13 |  |  | R13 |
| 34 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R15{A12} |  |  |  |  |  |  |  | R15{A12} | R15{A12} |  |  | R15{A12} |
| 35 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R16{A13} |  |  |  |  |  |  |  | R16{A13} | R16{A13} |  |  | R16{A13} |
| 36 |  |  |  |  |  |  |  |  |  | S40 | S32 |  | S33 |  |  | S35 |  |  |  |  |  |  |  |  |  |  | S34 | S36 |  |
| 37 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R6{A4} |  |  |  |  |  |  |  |  |  |  |  |  |
| 38 |  |  |  |  |  |  |  |  |  |  |  | S41 |  |  |  | R21{A8} |  |  |  |  |  |  |  |  |  |  | R21{A8} | R21{A8} |  |
| 39 |  |  |  |  |  |  |  |  |  |  |  |  |  | S42 |  | R22{A10} |  |  |  |  |  |  |  |  |  |  | R22{A10} | R22{A10} |  |
| 40 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | S38 |  |  |  | S43 |
| 41 |  |  |  |  |  |  |  |  |  |  | S44 |  | S33 |  |  | S35 |  |  |  |  |  |  |  |  |  |  | S34 | S36 |  |
| 42 |  |  |  |  |  |  |  |  |  |  |  |  | S45 |  |  | S35 |  |  |  |  |  |  |  |  |  |  | S34 | S36 |  |
| 43 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R17 |  |  |  |  |  |  |  | R17 | R17 |  |  | R17 |
| 44 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R12{A9} |  |  |  |  |  |  |  | R12{A9} | S39 |  |  | R12{A9} |
| 45 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | R14{A11} |  |  |  |  |  |  |  | R14{A11} | R14{A11} |  |  | R14{A11} |

Если таблицы, представляющие характеристический автомат и таблицу разбора, имеют большой размер, то их можно оформить как приложения к записке с соответствующей ссылкой на них.

Код лексического класса (код токена) определяет столбец таблицы разбора. Поскольку генератор таблиц разбора (TblGenerator.exe) формирует таблицу разбора независимо от кодировки токенов, то код токена не будет совпадать с номером соответствующего столбца. Поэтому следует предусмотреть специальную функцию для перекодировки кода токена в соответствующий номер столбца таблицы разбора.

Семантика синтаксических ошибок легко определяется по состоянию анализатора. Предполагается наличие специальной процедуры *Syntax\_Error*, формирующей сообщения об ошибках по состояниям анализатора. В случае возникновения любых ошибок в процессе СУ-трансляции после выдачи соответствующего сообщения трансляция немедленно прекращается.

# Структуры данных и алгоритмы

Вопросы выбора структур данных и алгоритмов, связанных с синтаксически управляемой трансляцией подробно рассмотрены в разделах 2 и 3, поскольку они тесно связаны с разработкой СУО и СУТ.

**Таблица разбора**.

Таблица разбора *LRTbl* представляет собой прямоугольную матрицу, состоящую из столбцов для каждого терминала и нетерминала (символам действий столбцы не сопоставляются) и строк, соответствующих каждому состоянию, в котором может находиться анализатор. Комбинация состояния в вершине стека анализатора и текущего входного символа используется в качестве индекса таблицы разбора и по ее элементу определяется дальнейшее действие. Таблица статическая, ее размеры известны.

Элемент таблицы *ElemLR* представляется следующими полями:

* *ElType* – тип элемента (элемент ошибки *lrErr* = 0, элемент переноса *lrShift* = 1, элемент свертки *lrReduct* = 2, элемент останова *lrStop* = 3);
* *ElPar* – параметр элемента (для элемента переноса – это номер состояния, который следует поместить в стек анализатора и реализовать переход в это состояние; для элемента свертки – это число элементов, исключаемых из стека анализатора, которое равно длине правой части соответствующей продукции без учета символов действий; для элементов ошибки и останова поле не используется);
* *Left* – нетерминал (точнее его внутренний код) левой части продукции, по которой выполняется свертка (имеет нулевое значение для всех типов элементов кроме элементов свертки), этот нетерминал будет являться входным символом на следующем этапе анализа после свертки;
* *Act* – код символа действия, по которому определяется процедура, реализующая это действие (нулевое значение поля означает отсутствие действия).

Для сопоставления токена и номера столбца таблицы разбора предусмотрен специальный массив *LexCodeLR*, которая заполняется при загрузке таблицы разбора из xml-файла. Индексом массива является код токена, т. е. *LexCodeLR*[*Token***.***Code*] – номер столбца таблицы разбора для токена *Token***.***Code*.

Формально в таблице разбора в качестве параметра элемента указывается номер соответствующей продукции (это сделано для наглядности). Поскольку при практической реализации свертки важно знать не столько саму продукцию, а сколько длину ее правой части, нетерминал ее левой части и наличие или отсутствие действия, лучше всю эту информацию хранить в элементе таблицы разбора. В противном случае, если в качестве параметра использовать номер продукции, пришлось бы в анализаторе иметь всю грамматику в виде пронумерованных продукций. В процессе свертки пришлось бы по номеру продукции перейти к соответствующей продукции, получить из нее данные о длине ее правой части, нетерминале левой части и наличии или отсутствии у нее символа действия. Предложенный же вариант предполагает наличие в анализаторе только таблицы разбора.

Алгоритм СУ-трансляции по таблице разбора:

*ClearStack*(*AttrSt*) //стек атрибутов пуст

*ClearStack*(*ParserSt*) //стек анализатора пуст

*NextAddr* := 0 //адрес очередной свободной ячейки в памяти данных *DataMem*

*NextInstr* := 0 //адрес очередной генерируемой команды в памяти команд *InstrMem*

*AddBaseTypes* //добавить в таблицу символов базовые типы *integer*, *real*

*Push*(1, *ParserSt*) //начальное состояние в стек анализатора

*Token* := *Scaner*(); //получение токена от сканера

*PredToken* := *Token*

**if** *Token***.***Code* = *lcErr* **then** *Lex\_Error* //лексическая ошибка

*Sym* := *LexCodeLR*[*Token***.***Code*] //*Sym* – номер столбца таблицы разбора

**while** *ElemLR***.***ElType* <> *lrStop* **do** //пока не элемент останова stop

**begin**

*State* := *StackTop*(*ParserSt*) //текущее состояние анализатора

*ElemLR* := *LRTbl* [*State*, *Sym*]

**case** *ElemLR***.***ElType* **of**

*lrErr*: //синтаксическая ошибка

*Syntax\_Error*(*State*)

*lrShift*: //элемент переноса

**begin**

*Push*(*ElemLR***.***ElPar*, *ParserSt*) //состояние в стек анализатора

**if** *Sym* терминал **then** //принять терминал

**begin**

*PredToken* := *Token* //сохранить токен как предшествующий очередному

*Token* := *Scaner*(); //получение следующего токена от сканера

**if** *Token***.***Code* = *lcErr* **then** *Lex\_Error* //лексическая ошибка

*Sym* := *LexCodeLR*[*Token***.***Code*] //*Sym* – номер столбца таблицы разбора

**end else** *Sym* := *LexCodeLR*[*Token***.***Code*] //текущий токен как входной символ

**end**

*lrReduct*: //элемент свертки

**begin**

**for** *i* := 1 **to** *ElemLR***.***ElPar* **do**

*tmp* := *Pop*(*ParserSt*) //удаление верхних элементов стека

*Sym* := *ElemLR***.***Left* //нетерминал левой части как новый входной символ

**if** *ElemLR*.*Act* <> 0 **then** //есть действие

*Semantic\_Action*(*ElemLR***.***Act*, *PredToken*) //выполнить действие

**end**

**end** //case

**end** //while

Переменная *PredToken* служит для сохранения предшествующего токена перед получением от сканера очередного входного токена. Необходимость такого сохранения связана со следующим. Для примера рассмотрим в таблице разбора состояние 35, в которое осуществлен переход из состояния 29 по прочитанному токену **id**. В состоянии 35 должна быть выполнена свертка *R*16 для продукции *Factor* → **id** {*A*13} и выполнено действие *A*13. В соответствии с алгоритмом анализатор принимает решение о свертке после получения очередного входного токена, принадлежащему множеству {**;**, **+**, **\***, **)** }. В результате предшествующий токен **id** будет утерян, хотя для реализации действия *A*13 он нужен.

Процедура *Semantic\_Action*(*ElemLR***.***Act*, *PredToken*) по коду действия (параметр *ElemLR***.***Act*) выбирает соответствующую подпрограмму, реализующую действие; для некоторых действий необходим токен (параметр *PredToken*).

**Перечень ошибок**.

Ошибки, обнаруживаемые в процессе СУ-трансляции, и их коды представлены в табл. 5. Ошибки типа BndErr не отражены в СУТ, поскольку они возникает при переполнении различных таблиц, представленных массивами, и обнаруживаются при добавлении новых элементов. Предусмотрено визуальное выделение ошибочной лексемы, а также указание номера строки и позиции в строке этой лексемы.

При желании можно дополнить сообщение о синтаксической ошибке списком допустимых в этой позиции токенов (типа «Допустим только **id**» или «Допустимы только **id**, **ass**»). Такой список легко можно сформировать по таблице разбора. Состояние (строка таблицы разбора), в котором обнаружена ошибка, известно, достаточно рассмотреть в этой строке только элементы, не являющиеся элементами ошибок, в столбцах, соответствующих терминалам, и сформировать список допустимых терминалов.

Таблица 5

Перечень ошибок, обнаруживаемых в процессе СУ-трансляции, и их кодов

|  |  |
| --- | --- |
| Код ошибки | Название ошибки |
|  | ***Ошибки переполнения*** |
| BndErr\_1 | Переполнение памяти данных (не более *MaxDataMem*) |
| BndErr\_2 | Переполнение памяти команд (не более *MaxInstrMem*) |
| BndErr\_3 | Превышен максимальный размер таблицы разбора (*MaxStateLR* × *MaxSymLR*) |
|  | ***Семантические ошибки*** |
| TypeErr\_1 | Повторное объявление идентификатора |
| TypeErr\_2 | Идентификатор не является именем типа |
| TypeErr\_3 | Идентификатор не является именем переменной |
| TypeErr\_4 | Несовместимость типов |
| TypeErr\_5 | Не объявлен идентификатор |
|  | ***Синтаксические ошибки*** |
| SynErr | Недопустимый токен (отображаются допустимые в данной позиции токены) |

# Программная реализация СУ-трансляции

Выбор среды разработки и языка программирования рассмотрен в РГР №1. Назначение полей и перечень их возможных значений подробно рассмотрены в разделах 2, 3, 4. Здесь представлена только их программная реализация. Ряд структур данных и подпрограмм рассмотрены в 1-й части РГР, некоторые из них дополнены новыми элементами. Здесь представлены только те структуры данных и подпрограммы, которые необходимы для реализации трансляции.

***Структуры данных*:** (студентам рекомендуется определить соответствующие классы ООП)

*Лексические классы* (*коды токенов*) (без изменений из 1-й части РГР):

tLexCode = (lcProg, lcStart, lcStop, lcVar, lcSemi, lcColon, lcComma, lcOpPar, lcClPar, lcAss,

lcAdd, lcMult, lcId, lcNum, lcEof, lcErr).

*Категории идентификаторов* *или число* (без изменений из 1-й части РГР):

tCatId = (catNoCat, catProgName, catTypeName, catVarName, catConst).

*Коды* *типов данных*:

tTypesCode = (typeVoid, typeInt, typeReal).

*Операции для лексических классов lcAdd*, *lcMult* и коды операций трехадресных команд:

tOperCode = (opAdd, opSub, opMult, opDiv, opAss).

По сравнению с 1-й частью РГР добавлена операция *opAss* для трехадресной команды копирования вида *x* := *y*.

*Размеры предопределенных типов* представлены именованными константами:

wInt = 4; //целый

wReal = 8; //вещественный

*Элемент типа данных*

tType = **record**

TypeCode: tTypesCode; //код типа данных

Width: integer; //размер типа

**end**

*Таблица символов* TblIde представляет собой односвязный список элементов со следующей структурой узла (таблица задается указателем на первый элемент списка):

tElemTblIde = **record**

Lex: string; //лексема идентификатора или числа

Cat: tCatId; //категория идентификатора или число

Typ: tType; //тип (**type** – ключевое слово, поэтому поле названо Typ)

Addr: integer; //адрес размещения в памяти

NextEl: tPntElTblIde; //след. элемент таблицы символов

**end**

По сравнению с 1-й частью РГР добавлено поле *Addr.*

*Токен*:

tToken = **record**

Code: tLexCode; //код токена (лексического класса)

Attr: Pointer; //атрибут токена (нетипизированный указатель)

Name: string; //обозначение токена (для визуализации)

LexBeg, LexEnd: Word; //для локализации ошибок

**end**

*Типы элементов LR-таблицы разбора* (*поле ElType*) как именованные константы:

lrErr = 0; //элемент ошибки

lrShift = 1; //элемент переноса

lrReduct = 2; //элемент свертки

lrStop = 3; //элемент останова

*Элемент LR-таблицы разбора*:

tElemLR = **record**

ElType: Byte; //тип элемента (lrErr, lrShift, lrReduct, lrStop)

ElPar: Byte; //параметр элемента (для элемента переноса – это номер состояния,

для элемента свертки – длина правой части свертываемой продукции

Left: Byte; //код левого нетерминала свертываемой продукции

Act: Byte; //код символа действия (0, если действия нет)

**end**

*Трехадресная команда* (*четверка*):

tInst = **record**

Op: tOperCode; //код операции

Arg1, Arg2, Res: Pointer; //поля для аргументов и результата

**end**

*Память данных*:

DataMem: **array** [0 .. MaxDataMem] **of** Byte

*Память трехадресных команд*:

InstrMem: **array** [0 .. MaxInstrMem] **of** tInst

*Таблица разбора*:

LRTbl: **array** [1 .. MaxStateLR, 1 .. MaxSymLR] **of** tElemLR

*Соответствие кодов токена и номеров столбцов таблицы разбора*:

LexCodeLR: **array** [tLexCode] **of** integer

*Названия столбцов таблицы разбора*:

SymsLR: **array** [1 .. MaxSymLR] **of** string

Элементы 1 .. NumNeTerm соответствуют нетерминалам, а элементы NumNeTerm+1 .. NumColLR соответствуют терминалам. Заполнение массива и вычисление значений переменных NumNeTerm (число нетерминалов грамматики) и NumColLR (число столбцов в таблице разбора) производится в процессе загрузки таблицы разбора из xml-файла.

***Процедуры и функции*:**

**function** SUT\_LR: Boolean реализует СУ-трансляцию, возвращает **true**, если СУ-трансляция выполнена успешно, и **false** в противном случае.

Процедуры формирования текстов об ошибках:

**procedure** Bound\_Error (Code: Byte; Token: tToken);

**procedure** Type\_Error(Code: Byte; Token: tToken);

**procedure** Syntax\_Error(State: Byte; Token: tToken).

В первых двух процедурах входной параметр *Code* являются кодом ошибки, входной параметр *State* процедуры *Syntax\_Error* указывает состояние анализатора, по которому можно сформировать детальное сообщение о соответствующей ошибке. Параметр *Token* всех трех процедур – это токен, с которым связана ошибка.

**function** Semantic\_Action (ActionCode: Byte; Token: tToken): Boolean реализует выполнение семантических действий, возвращает **true**, если действия выполнены успешно, и **false** в противном случае. Данная функция по коду действия (параметр *ActionCode*) выбирает соответствующую подпрограмму, реализующую действие; для некоторых действий необходим токен (параметр *Token*). Подпрограммы реализации семантических действий определены следующим образом:

**function** NewTemp(Typ: tType): tPntElTblIde;

**procedure** AddType(Pnt: tPntElTblIde; Cat: tCatId; Typ: tType);

**procedure** Gen(Op: tOperCode; Arg1, Arg2, Res: Pointer).

В функции NewTemp в качестве префикса генерируемой лексемы временной переменной выбран символ «$».

Семантические действия *A*1, …, *A*13 определены следующим образом:

а) действия, в которых возможно обнаружение ошибок, реализованы функциями, возвращающими **true**, если действие выполнено успешно (ошибки не обнаружены), и **false** в противном случае, например, **function** A1 (Token: tToken): Boolean;

б) действия, в которых нет обнаружения ошибок, но требуется токен, реализованы процедурами, например, **procedure** A8 (Token: tToken);

в) действия, в которых нет обнаружения ошибок и нет входных параметров, реализованы процедурами, например, **procedure** A3;

Для отображения результатов трансляции и распределения памяти реализованы подпрограммы:

**function** OutInstr(Ind: integer): string отображает трехадресную команду с адресом Ind.

**procedure** OutDataMem отображение памяти данных.

***Инструкция пользователя и результаты тестирования и отладки****.*

Разработанное приложение работает под ОС Windows, не требует какой-либо специальной установки и представляет собой исполняемый файл **SUT.exe**. После запуска приложения формируется подпапка **TESTS,** которая является папкой по умолчанию для загрузки и сохранения исходных текстов программ как текстовых файлов.

Интерфейс и результаты работы приложения на тестовом примере представлены на рис. 1 (успешная трансляция) и рис. 2 (пример обнаружения семантической ошибки). Группа элементов «Текст тестовой программы» включает в себя область ввода и редактирования текста программы и кнопки для загрузки и сохранения текста программы в текстовый файл. Нажатие кнопок откроет стандартное окно диалога для выбора папок и имен файлов. В статусной строке группы показывается позиция курсора в области редактирования (номер строки и позиция в строке) и информация о том, были ли внесены в грамматику изменения. Кнопка «Загрузить ТР» предназначена для импорта *SLR*(1)-таблицы разбора из xml-файла, сформированного генератором таблиц разбора.

Для запуска СУ-трансляции служит кнопка «СУТ». Результаты трансляции отражаются в элементе «Трехадресные команды», а распределение памяти – в элементе «Распределение памяти данных», в элементе «Обнаруженные ошибки» формируется сообщение об успехе или неуспехе трансляции. Если при трансляции обнаружена ошибка, в элементе «Обнаруженные ошибки» отображается сообщение об ошибке, ее код и визуально выделяется ошибочная лексема. Транслятор прекращает работу сразу же после первой обнаруженной ошибки.

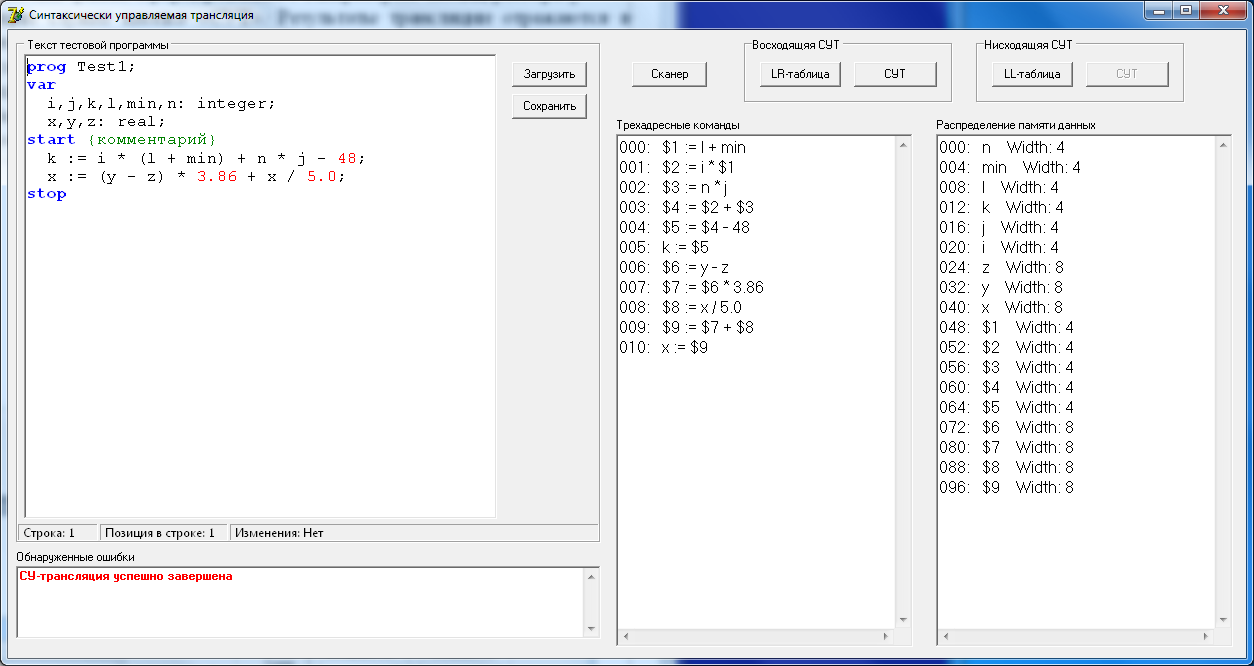


Рис. 1. Интерфейс приложения и пример успешной СУ-трансляции

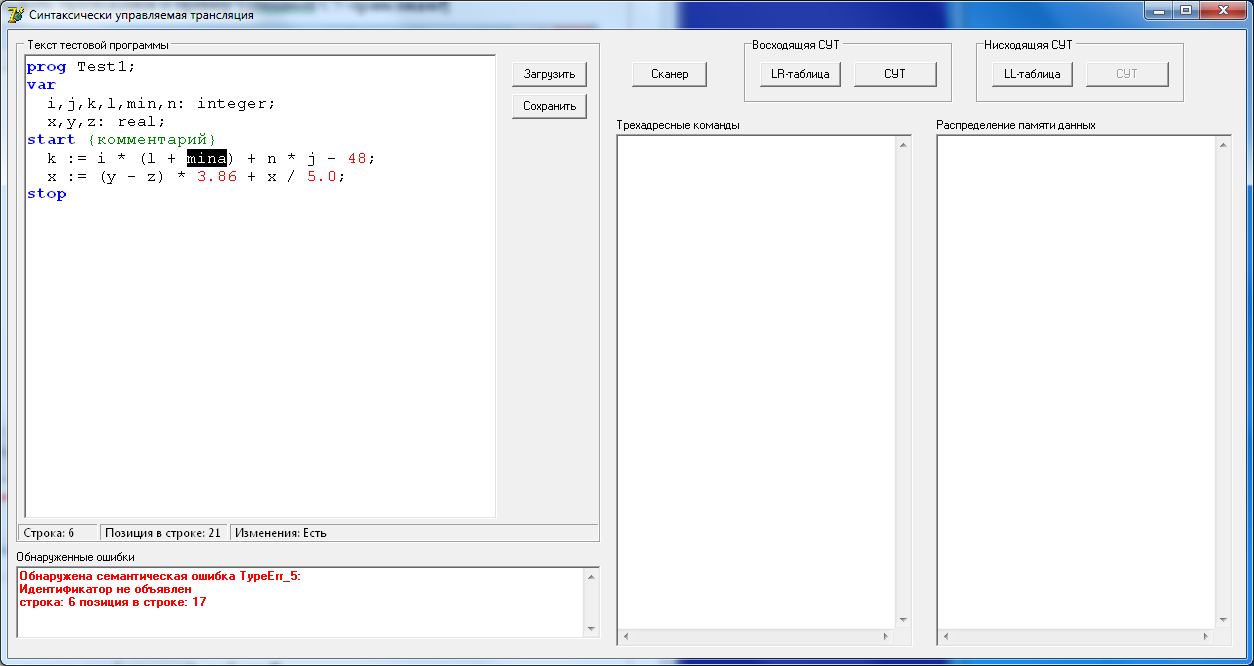


Рис. 2. Пример обнаружения семантической ошибки

Заключение

В процессе выполнения расчетно-графической работы

* изучены методы синтаксически управляемой трансляции;
* получены практические навыки построения моделей СУ-трансляции (СУО и СУТ);
* разработана СУТ, ориентированная на восходящий разбор;
* разработаны структуры данных для представления таблиц СУ-транслятора;
* получены практические навыки разработки модуля СУ-трансляции.

Список использованной литературы

1. *Ахо, А*. Компиляторы: принципы, технологии и инструментарий / А. Ахо, М. Лам, Р. Сети, Д. Ульман. – 2-е изд.– М.: Вильямс, 2008. – 1184 с.

2. *Опалева*, *Э*.*А*. Языки программирования и методы трансляции / Э.А. Опалева, В.П. Самойленко.– СПб.: БХВ-Петербург, 2005.– 480 с.

3. *Павлов, Л.А*. Восходящий синтаксический анализ: конспект лекций / Л.А. Павлов.– Чебоксары: Изд-во Чуваш. ун-та, 2004. – 44 с.

4. *Павлов, Л.А*. Лексический и синтаксический анализ: учеб. пособие / Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2022. – 152 с.

5. *Павлов, Л.А*. Нисходящий синтаксический анализ: конспект лекций / Л.А. Павлов.– Чебоксары: Изд-во Чуваш. ун-та, 2003. – 48 с.

6. *Павлов, Л.А*. Синтаксически управляемая трансляция: учеб. пособие / Л.А. Павлов. Чебоксары: Изд-во Чуваш. ун-та, 2017. – 60 с.

7. *Павлов, Л.А*. Структуры и алгоритмы обработки данных: учеб. пособие / Л.А. Павлов, Н.В. Первова. – 2-е  
изд. – Чебоксары: Изд-во Чуваш. ун-та, 2018. – 254 с.

8. *Павлов, Л.А*. Структуры и алгоритмы обработки данных: учебник / Л.А. Павлов, Н.В. Первова. – 3-е изд. – СПб.: Лань, 2021. – 256 с.

9. *Свердлов*, *С.З*. Языки программирования и методы трансляции: учеб. пособие / С.З. Свердлов.– СПб.: Питер, 2007.– 638 с.

10. *Свердлов*, *С.З*. Языки программирования и методы трансляции: учеб. пособие / С.З. Свердлов. – 2-е изд. – СПб.: Лань, 2019.– 564 с.

11. Теория языков программирования и методы трансляции: Метод. указания к выполнению расчетно-графических работ / Сост. Л.А. Павлов. – Чебоксары: Изд-во Чуваш. ун-та, 2015. – 48 с.