**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](#_Toc174702724)

[2. Синтаксически управляемое определение и схема трансляции 6](#_Toc174702725)

[2.1. Семантический анализ (проверка типов) 6](#_Toc174702726)

[2.2. Генерация промежуточного кода 7](#_Toc174702727)

[2.3. Разработка СУО 8](#_Toc174702728)

[2.4. Разработка СУТ 9](#_Toc174702729)

[3. Структуры данных и алгоритмы 18](#_Toc174702730)

[4. Программная реализация СУ-трансляции 20](#_Toc174702731)

[Заключение 23](#_Toc174702732)

[Список использованной литературы 23](#_Toc174702733)

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

Данная грамматика не подходит для реализации синтаксически управляемой трансляции.

Продукции

4) *DeclVar* → *LstId* **: id**

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

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 нисходящего анализа такая грамматика не подходит, требуется *LL*(1)-грамматика. Здесь можно использовать два подхода. При первом подходе сначала грамматика преобразуется к требуемому классу, затем разрабатывается СУО, а затем СУО преобразуется в СУТ. При втором подходе сначала для грамматики (неважно, относится она к требуемому классу или нет) разрабатывается СУО или СУТ, а затем выполняются все необходимые преобразования грамматики с учетом наличия в ней семантических действий.

Для нисходящего анализа исходную грамматику лучше представлять в более удобном для последующего преобразования в *LL*(1)-форму виде. В частности, непустой список *L* элементов a лучше задавать продукциями *L* → *a* **|** *aL* (в отличие от восходящего анализа, когда список задается как *L* → *a* **|** *La*). 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. Такой подход позволит свести до минимума число леворекурсивных продукций в грамматике. Если грамматика является основой для построения СУТ, то после включения в нее семантических действий, будет легче выполнить преобразование СУТ в *LL*(1)-форму.

# 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* **stop** (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

*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 простоты 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 должен выравниваться каждый элемент.

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 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 2. Схема трансляции (СУТ) для проверки типов и трансляции в трехадресный код языка *DemoLang*

| Продукция с действиями | Семантические правила | Семантические действия | Символ  действия |
| --- | --- | --- | --- |
| *S* → **prog** **id** {*A*1} **;** *Block* **stop** | **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* |  |  |  |
| *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*)  *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* **stop**

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

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

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

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

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

*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 это не *LL*(1)-грамматика, для нисходящей трансляции построенная СУТ не подходит. Следует преобразовать ее в *LL*(1)-форму.

**Процесс преобразования в *LL*(1)-форму**

В первую очередь необходимо устранить левую рекурсию в продукциях

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

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

Применив правило устранения левой рекурсии [4; 5], получим

*Expr* → *Term* *X*

*X* → ε **|** **+** {*A*8} *Term* {*A*9} *X*

*Term* → *Factor* *Y*

*Y* → ε **|** **\*** {*A*10} *Factor* {*A*11} *Y*

Теперь приступим к левой факторизации [4; 5]. При выполнении левой факторизации для наглядности будем показывать вынос за скобки общих префиксов продукций, к которым применяется левая факторизация. Применим скобки вида **〈** и **〉**, поскольку они не входят в алфавит языка.

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

*LstVar* → *DeclVar* **;** **〈**ε **|** *LstVar* **〉**

получим

*LstVar* → *DeclVar* **;** *A*

*A* → ε **|** *LstVar*

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

*LstStmt* → *Stmt* **;** **〈**ε **|** *LstStmt* **〉**

получим

*LstStmt* → *Stmt* **;** *B*

*B* → ε **|** *LstStmt*

**Примечание**. Рассмотренный выше процесс преобразований грамматики в *LL*(1)-форму приведен для иллюстрации. Вы можете процесс преобразований не включать в отчет по РГР, главное, чтобы Вы могли показать необходимые преобразования при защите РГР.

В результате получили *LL*(1)-грамматику, ориентированную на нисходящую синтаксически управляемую трансляцию:

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

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

*LstVar* → *DeclVar* **;** *A*

*A* → ε **|** *LstVar*

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

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

*LstStmt* → *Stmt* **;** *B*

*B* → ε **|** *LstStmt*

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

*Expr* → *Term* *X*

*X* → ε **|** **+** {*A*8} *Term* {*A*9} *X*

*Term* → *Factor* *Y*

*Y* → ε **|** **\*** {*A*10} *Factor* {*A*11} *Y*

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

Направляющие символы и схема грамматики приведены в табл. 2, а *LL*(1)-таблица разбора – в табл. 3. Для построения этих таблиц использован генератор таблиц разбора (приложение TblGenerator.exe).

Таблица 2. Направляющие символы и схема грамматики

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Продукция | Направляющие символы (DS) | Схема грамматики |
| 1 | S = "prog" "id" {A1} ";" Block "eof" | prog | 1 = 2 3 4 5 6 7 |
| 2 | Block = "var" LstVar "start" LstStmt "stop" | var | 8 = 9 10 11 12 13 |
| 3 | LstVar = DeclVar ";" A | id | 14 = 15 16 17 |
| 4 | A = | start | 18 = 20 |
| 5 | A = LstVar | id | 19 = 21 |
| 6 | DeclVar = "id" {A2} LstId {A3} | id | 22 = 23 24 25 26 |
| 7 | LstId = "," "id" {A2} LstId {A4} | , | 27 = 29 30 31 32 33 |
| 8 | LstId = ":" "id" {A5} | : | 28 = 34 35 36 |
| 9 | LstStmt = Stmt ";" B | id | 37 = 38 39 40 |
| 10 | B = | stop | 41 = 43 |
| 11 | B = LstStmt | id | 42 = 44 |
| 12 | Stmt = "id" {A6} "ass" Expr {A7} | id | 45 = 46 47 48 49 50 |
| 13 | Expr = Term X | id, num, ( | 51 = 52 53 |
| 14 | X = | ;, ) | 54 = 56 |
| 15 | X = "+" {A8} Term {A9} X | + | 55 = 57 58 59 60 61 |
| 16 | Term = Factor Y | id, num, ( | 62 = 63 64 |
| 17 | Y = | ;, +, ) | 65 = 67 |
| 18 | Y = "\*" {A10} Factor {A11} Y | \* | 66 = 68 69 70 71 72 |
| 19 | Factor = "num" {A12} | num | 73 = 76 77 |
| 20 | Factor = "id" {A13} | id | 74 = 78 79 |
| 21 | Factor = "(" Expr ")" | ( | 75 = 80 81 82 |
| LL(1)-грамматика | | | |

Таблица 3. *LL*(1)-таблица разбора

| № | Terminals | Jump | Accept | Stack | Action | Error |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | prog | 2 | false | false |  | true |
| 2 | prog | 3 | true | false |  | true |
| 3 | id | 4 | true | false |  | true |
| 4 |  | 5 | false | false | A1 | true |
| 5 | ; | 6 | true | false |  | true |
| 6 | var | 8 | false | true |  | true |
| 7 | eof | 0 | true | false |  | true |
| 8 | var | 9 | false | false |  | true |
| 9 | var | 10 | true | false |  | true |
| 10 | id | 14 | false | true |  | true |
| 11 | start | 12 | true | false |  | true |
| 12 | id | 37 | false | true |  | true |
| 13 | stop | 0 | true | false |  | true |
| 14 | id | 15 | false | false |  | true |
| 15 | id | 22 | false | true |  | true |
| 16 | ; | 17 | true | false |  | true |
| 17 | id, start | 18 | false | false |  | true |
| 18 | start | 20 | false | false |  | false |
| 19 | id | 21 | false | false |  | true |
| 20 | start | 0 | false | false |  | true |
| 21 | id | 14 | false | false |  | true |
| 22 | id | 23 | false | false |  | true |
| 23 | id | 24 | true | false |  | true |
| 24 |  | 25 | false | false | A2 | true |
| 25 | ,, : | 27 | false | true |  | true |
| 26 |  | 0 | false | false | A3 | true |
| 27 | , | 29 | false | false |  | false |
| 28 | : | 34 | false | false |  | true |
| 29 | , | 30 | true | false |  | true |
| 30 | id | 31 | true | false |  | true |
| 31 |  | 32 | false | false | A2 | true |
| 32 | ,, : | 27 | false | true |  | true |
| 33 |  | 0 | false | false | A4 | true |
| 34 | : | 35 | true | false |  | true |
| 35 | id | 36 | true | false |  | true |
| 36 |  | 0 | false | false | A5 | true |
| 37 | id | 38 | false | false |  | true |
| 38 | id | 45 | false | true |  | true |
| 39 | ; | 40 | true | false |  | true |
| 40 | id, stop | 41 | false | false |  | true |
| 41 | stop | 43 | false | false |  | false |
| 42 | id | 44 | false | false |  | true |
| 43 | stop | 0 | false | false |  | true |
| 44 | id | 37 | false | false |  | true |
| 45 | id | 46 | false | false |  | true |
| 46 | id | 47 | true | false |  | true |
| 47 |  | 48 | false | false | A6 | true |
| 48 | ass | 49 | true | false |  | true |
| 49 | id, num, ( | 51 | false | true |  | true |
| 50 |  | 0 | false | false | A7 | true |
| 51 | id, num, ( | 52 | false | false |  | true |
| 52 | id, num, ( | 62 | false | true |  | true |
| 53 | ;, +, ) | 54 | false | false |  | true |
| 54 | ;, ) | 56 | false | false |  | false |
| 55 | + | 57 | false | false |  | true |
| 56 | ;, ) | 0 | false | false |  | true |
| 57 | + | 58 | true | false |  | true |
| 58 |  | 59 | false | false | A8 | true |
| 59 | id, num, ( | 62 | false | true |  | true |
| 60 |  | 61 | false | false | A9 | true |
| 61 | ;, +, ) | 54 | false | false |  | true |
| 62 | id, num, ( | 63 | false | false |  | true |
| 63 | id, num, ( | 73 | false | true |  | true |
| 64 | ;, +, \*, ) | 65 | false | false |  | true |
| 65 | ;, +, ) | 67 | false | false |  | false |
| 66 | \* | 68 | false | false |  | true |
| 67 | ;, +, ) | 0 | false | false |  | true |
| 68 | \* | 69 | true | false |  | true |
| 69 |  | 70 | false | false | A10 | true |
| 70 | id, num, ( | 73 | false | true |  | true |
| 71 |  | 72 | false | false | A11 | true |
| 72 | ;, +, \*, ) | 65 | false | false |  | true |
| 73 | num | 76 | false | false |  | false |
| 74 | id | 78 | false | false |  | false |
| 75 | ( | 80 | false | false |  | true |
| 76 | num | 77 | true | false |  | true |
| 77 |  | 0 | false | false | A12 | true |
| 78 | id | 79 | true | false |  | true |
| 79 |  | 0 | false | false | A13 | true |
| 80 | ( | 81 | true | false |  | true |
| 81 | id, num, ( | 51 | false | true |  | true |
| 82 | ) | 0 | true | false |  | true |

*LL*(1)-таблица разбора представляет собой набор строк. Каждая строка содержит поля (столбцы):

а) список терминалов – *Terminals*,

б) поле переходов – *Jump*,

в) поле приема – *Aссept*,

г) поле стека – *Staсk*,

д) поле действия – *Action*,

е) поле ошибки – *Error*.

Область значений поля *Jump* – неотрицательные целые числа (номера строк таблицы), а область значений полей *Aссept*, *Staсk* и *Error* – {**true**, **false**}. По сравнению со структурой таблицы, рассмотренной в [4; 5], исключено поле *Return* из-за его избыточности, добавлено поле *Action* для хранения символа действия. Правила заполнения таблицы с учетом наличия в грамматике символов действий изложены в [6].

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 *LLTbl* представляет собой массив строк. Строка *ElemTR* содержит поля: *Terminals*, *Jump*, *Accept*, *Stack*, *Action*, *Error*. Назначение и правила заполнения полей подробно рассмотрены в разделе 3.

Алгоритм СУ-трансляции по *LL*(1)-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*(0, *ParserSt*) //начальное состояние в стек анализатора

*i* := 1 //номер 1-й строки таблицы разбора

*la* := **true**

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

*PredToken* := *Token*

**while** (*Token***.***Code* ≠ *lcEof*) **or** (*i* ≠ 0) **do**

**begin**

**if** (*LLTbl*[*i*]**.***Action* ≠ 0) **or** (*Token***.***Code* ∈ *LLTbl*[*i*]**.***Terminals*) **then**

**begin**

**if** *LLTbl*[*i*]**.***Action* ≠ 0 **then** //строка соответствует символу действия

*Semantic\_Action*(*LLTbl*[*i*]**.***Action*, *PredToken*) //выполнить действие

*la* := *LLTbl*[*i*]**.***Accept*

**if** *LLTbl*[*i*]**.***Jump* ≠ 0 **then**

**begin**

**if** *LLTbl*[*i*]**.***Stack* **then** *Push*(*i* + 1, *ParserSt*)

*i* := *LLTbl*[*i*]**.***Jump*

**end else** *i* :=*Pop*(*ParserSt*)

**end else**

**if not** *LLTbl*[*i*]**.***Error* **then**

**begin**

*i* := *i* + 1

*la* := **false**

**end else** *Syntax\_Error*(*i*) //синтаксическая ошибка

**if** *la* **then**

**begin**

*PredToken* := *Token*

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

**end**

**end**

**end** //while

Данный алгоритм является модификацией алгоритма из [4; 5]. Здесь переменная *lа* представляет собой логическое значение, которое определяет, надо ли считывать новый предварительно просматриваемый символ до обработки следующего элемента таблицы разбора. Например, *la* = **false**, когда предварительно просматриваемый символ не является направляющим для какой-либо конкретной продукции и требуется исследовать множество направляющих символов следующей альтернативной продукции. Если символ не содержится в текущем множестве направляющих символов и поле ошибки *Error* будет **true**, то выдается сообщение о синтаксической ошибке. Если поле стека обрабатываемой *i*-й строки таблицы разбора имеет значение **true**, то до перехода к адресу, задаваемому полем перехода, в стек помещается адрес следующей строки таблицы.

Служебный код токена *lcEof* обозначает маркер конца ввода.

Процедура *Syntax\_Error*(*i*) генерирует сообщение о синтаксической ошибке по номер *i* строки, в которой была зафиксирована ошибка.

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 строку 3, в которой осуществляется прием токена **id** (поле *Accept* = **true**), получение очередного токена (токен «точка с запятой») и реализуется переход к строке 4 (поле *Jump* = 4) для выполнения действие *A*1. В результате предшествующий токен **id** будет утерян, хотя для реализации действия *A*1 он нужен.

Процедура *Semantic\_Action*(*LLTbl*[*i*]**.***Action*, *PredToken*) по коду действия (параметр *LLTbl*[*i*]**.***Action*) 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.

**Примечание**. 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 (не более *MaxRowLL*) |
|  | ***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**

*Элемент LL-таблицы разбора*:

tElemLL = **record**

Jump: Word; //поле переходов

Accept: Boolean; //поле приема

Stack: Boolean; //поле стека

Action: Byte; //поле действия

Error: Boolean; //поле ошибки

Terminals: **set** **of** tLexCode; //множество кодов терминалов-токенов

**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*:

LLTbl: **array** [1 .. MaxRowLL] **of** tElemLL;

***Процедуры и функции*:**

**function** SUT\_LL: 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.

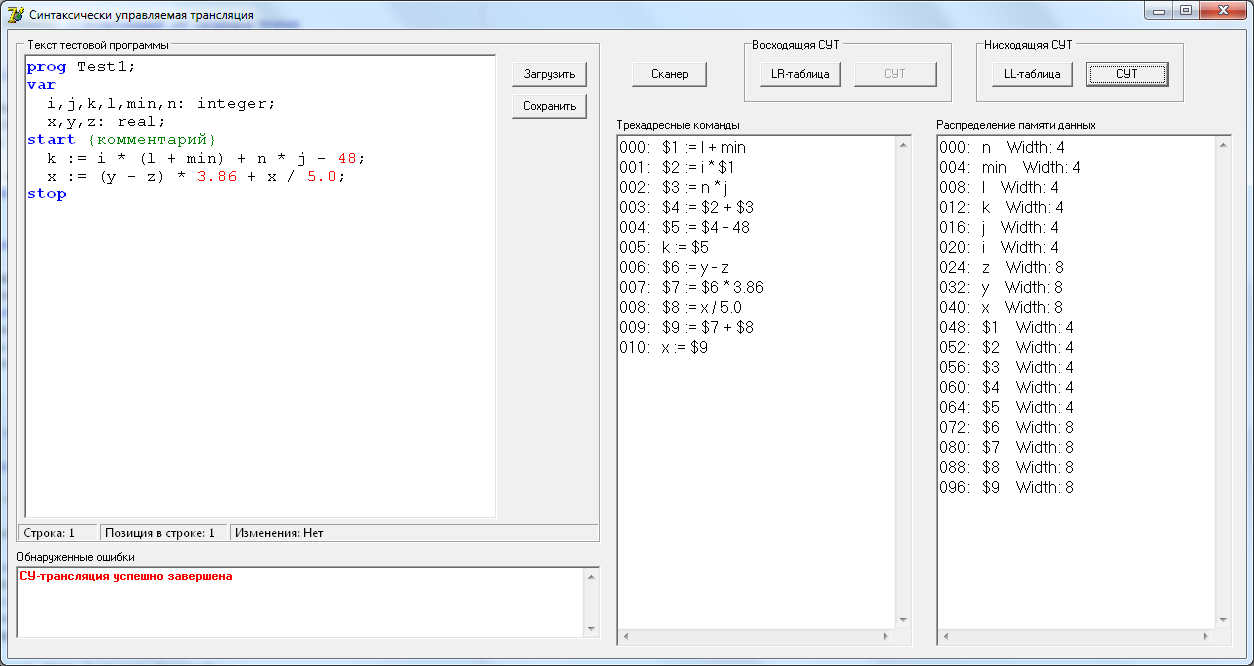


Рис. . Интерфейс приложения и пример успешной СУ-трансляции

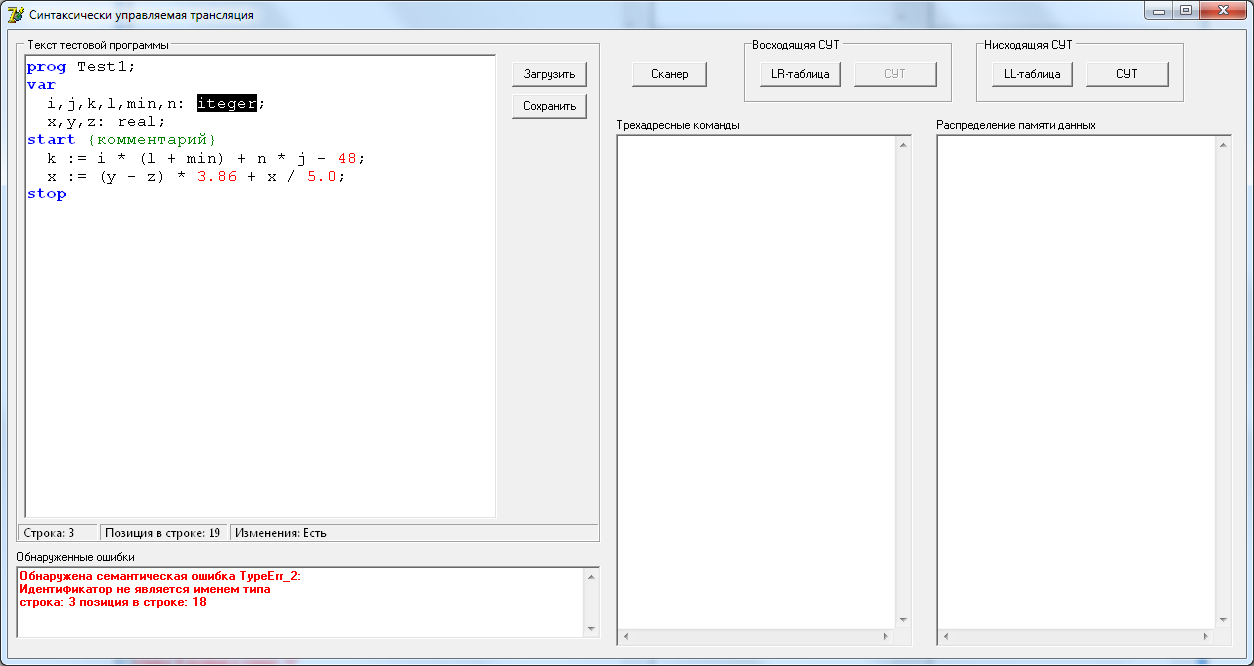


Рис. . Пример обнаружения семантической ошибки

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.