**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 №2**

**Синтаксически управляемая трансляция**

**Выполнил:**

студент группы ИВТ-41-13

Фамилия И.О.

**Руководитель:**

доцент Павлов Л.А.

Чебоксары 2021

**Оглавление**

[Задание к РГР 3](#_Toc431747928)

[Введение 4](#_Toc431747929)

[1. Контекстно-свободная грамматика языка 5](#_Toc431747930)

[2. Синтаксически управляемое определение 6](#_Toc431747931)

[3. Синтаксически управляемая схема трансляции 11](#_Toc431747932)

[4. Структуры данных и алгоритмы 23](#_Toc431747933)

[5. Программная реализация СУ-трансляции 24](#_Toc431747934)

[Заключение 28](#_Toc431747935)

[Список использованной литературы 28](#_Toc431747936)

Задание к РГР

 1. Проработать все теоретические вопросы, связанные с нисходящей синтаксически управляемой трансляцией языка *MyLang*, разработанного в РГР №1, 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 нисходящей синтаксически управляемой трансляции.

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 РГР №1 множеством лексических классов синтаксис языка *MyLang* в виде РБНФ, где в качестве терминалов выступают токены, можно представить следующим образом:

1. СхемаПрогр = "**scheme**" "**id**" "**;**" Блок "**endsch**".
2. Блок = СписОбъявл "**start**" ПоследОператоров "**;**" ЗаклОператор.
3. СписОбъявл = Объявление { Объявление }.
4. Объявление = [ Категория ] Тип СписокИдент "**;**".
5. Категория = "**const**" **|** "**functional**".
6. Тип = "**id**".
7. СписокИдент = "**id**" { "**,**" "**id**" }.
8. ПоследОператоров = Оператор { "**;**" Оператор }.
9. Оператор = ["**lbl**" "**:**"] ОпБезМетки.
10. ОпБезМетки = Присваивание **|** Условный.
11. Присваивание = "**id**" "**:=**" Терм ["**goto**" "**lbl**" ].
12. Терм = "**id**" [ "**(**" Аргументы "**)**" ].
13. Аргументы = Терм **|** Терм "**,**" Терм.
14. Условный = "**if**" Терм "**then**" "**lbl**" "**else**" "**lbl**".
15. ЗаклОператор = ["**lbl**" "**:**"] "**stop**".

 Построим по данной РБНФ классическую 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.

 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:

 *S* → **scheme** **id** **;** *Block***endsch**

 *Block* → *LstDecl* **start***LstStmt***;** *StopStmt*

 *LstDecl* → *Decl* **|**  *Decl* *LstDecl*

 *Decl* → **id***LstId* **;** **|** *Categ* **id***LstId* **;**

 *Categ* → **const** **| functional**

 *LstId* → **id** **| id** **,** *LstId*

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

 *Stmt* → *StmtNoLbl* **| lbl :***StmtNoLbl*

 *StmtNoLbl* → *AssStmt* **|** *TestStmt*

 *AssStmt* → **id** **ass** *Term* **| id** **ass** *Term***goto lbl**

 *Term* → **id | id (***Argum* **)**

 *Argum* → *Term***|***Term***,***Term*

 *TestStmt* → **if** *Term* **then** **lbl** **else** **lbl**

*StopStmt* → **lbl : stop****| stop**

 **Примечание для студента**. 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.

 **Еще одно примечание для студента**. 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

## 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), float (8), Boolean (1). 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 и констант.

 5. В продукции *S* → **scheme** **id** **;** *Block* **endsch** имя схемы **id** не должно встречаться ни в какой другой конструкции языка.

 6. Если терм соответствует продукции *Term* → **id**, то **id** может быть только именем переменной (не указана категория *Categ* в объявлении *Decl*) или константы (категория **const**). Тип терма определяется типом **id**.

 7. Если терм соответствует продукции *Term* → **id (***Argum* **)**, то **id** (префикс терма) может быть только функциональным идентификатором (категория **functional** в объявлении *Decl*), причем типы аргументов из *Argum* должны совпадать. Тип терма определяется типом функционального идентификатора.

 8. Имя константы (категория **const** в объявлении *Decl*) не должно встречаться в левой части оператора присваивания (левой частью может быть только имя переменной).

 9. Функциональный идентификатор (категория **functional** в объявлении *Decl*) может быть только префиксом терма с аргументами, не может быть аргументом терма и не может встречаться в левой части оператора присваивания (т. е. не может использоваться как переменная).

 10. В условном операторе (нетерминал *TestStmt*) в качестве терма *Term* может быть только терм типа Boolean.

 11. Для метки из правой части оператора, реализующей передачу управления, обязательно должен быть оператор, помеченный этой меткой. Не может быть двух операторов с одинаковыми метками.

 ***Выражения типов***.

 Типы имеют структуру, представленную с использованием выражения типов:

 а) фундаментальный тип является выражением типа, фундаментальные типы включают предопределенные типы *integer*, *float*, *Boolean*, 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.

 Поскольку в языке *MyLang* идентификатор может быть именем программы, функциональным идентификатором, именем типа, именем переменной или именем константы. Поэтому используется понятие категории идентификатора. Примем следующие обозначения категорий: *catProgName* – имя программы, *catFuncName* – функциональный идентификатор, *catTypeName* – имя типа, *catVarName* – имя переменной, *catConstName* – имя константы, *None* – категория идентификатора еще не установлена.

 В языке *MyLang* все типы простые, поэтому нет необходимости в конструкторах типа.

 **Примечание для студента**. У Вас будет свой конструктор типа (массив, запись, процедура или функция) в зависимости от варианта задания.

 Рассмотрим пригодность формальной грамматики для разработки СУО в части семантического анализа.

 1. Продукции для объявления типа для некоторого списка идентификаторов *LstId* (**id** – имя типа)

 *Decl* → *Categ* **id***LstId* **;**

 *Categ* → **const** **| functional**

потребуют введения для нетерминала *Categ* атрибута для определения категории идентификаторов *LstId*. Этих действий можно избежать соответствующим преобразованием грамматики, выполнив замену вхождений нетерминала *Categ* (исключаем нетерминал *Categ* из грамматики):

 *Decl* → **const** **id***LstId* **;** **|** **functional** **id***LstId* **;**

 Теперь тип, категорию и размер типа можно рассматривать как наследуемые атрибуты нетерминала *LstId*, которые можно внести в таблицу символов каждого идентификатора, порождаемого *LstId*, как это показано в СУО в табл. 1.

## 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 внутренний 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. Трехадресная команда *x* := *f*(*y*, *z*) представляется размещением кода функционального идентификатора *f* в *op*, *y* в *arg*1, *z* в *arg*2 и *x* в *result*. Инструкции с унарными действиями наподобие *x* := *f*(*y*) или *x* := *y* не используют *arg2*. Условные и безусловные переходы помещают в *result* целевую метку.

 Для вычисления логических выражений (термов тип Boolean) используется 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 выражений. Для реализации однопроходной трансляции логических выражений и управляющих операторов используется метод обратных поправок.

 Перечень трехадресных команд:

 1) команда присваивания вида *x* := *f*(*y*, *z*), где *f* – двухместный функциональный идентификатор (соответствует бинарной операции), *y* и *z* – переменные или константы, *x* – переменная;

 2) команда присваивания вида *x* := *f*(*y*), где *f* – одноместный функциональный идентификатор (соответствует унарной операции), *y* – переменная или константа, *x* – переменная;

 3) команда копирования вида *x* := *y*, где *y* – переменная или константа, *x* – переменная;

 4) безусловный переход **goto** *L*, после этой инструкции будет выполнена инструкция с меткой *L*;

 5) условный переход вида **if** *x* **goto** *L* (*x* – логическая переменная), если значение *х* истинно, следующей выполняется команда с меткой *L*, в противном случае выполняется следующая за условным переходом команда.

 Реализуется инкрементная трансляция, которая заключается в том, что формируется единый поток генерируемых трехадресных команд в некотором глобальном массиве или файле. Для реализации передачи управления в командах перехода используется метод обратных поправок.

 В качестве структурами данных для представления трехадресных команд используются четверки (тетрады). Четверка представляет собой запись с полями *op*, *arg*1, *arg*2 и *result*. Поле *op* содержит указатель на запись в таблице символов, соответствующую функциональному идентификатору. Поля *arg*1, *arg*2 и *result* содержат указатели на соответствующие записи таблицы символов. Команда вида *x* := *f*(*y*, *z*) представляется размещением *f* в *op*, *y* в *arg*1, *z* в *arg*2 и *x* в *result*. Команда вида *x* := *f*(*y*) не использует *arg*2. Команда копирования вида *x* := *y* не использует *op* и *arg*2. Команда безусловного перехода вида **goto** *L* представляется размещением кода операции **goto** в *op*, метки перехода *L* в *result* (*arg*1 и *arg*2 не используются). Команда условного перехода вида **if** *x* **goto** *L* представляется размещением кода операции **if** в *op*, *x* в *arg*1, метки перехода *L* в *result* (*arg*2 не используются). Для команды **stop** в *op* размещается код операции **stop** (остальные поля не используются).

 **Примечание для студента**. У Вас будет свой перечень трехадресных команд.

 Рассмотрим пригодность формальной грамматики для разработки СУО в части генерации промежуточного кода.

 1. Продукциях *Term* → **id (***Argum* **)** и *Argum* → *Term***|***Term***,** *Term* не подходят для генерации команд, поскольку команда включает в себя и **id**, и термы (один или два), т. е. придется формировать трехадресную команду по частям в разных продукциях. Поэтому эти продукции лучше преобразовать в следующие продукции (исключаем нетерминал *Argum* из грамматики):

 *Term* → **id (***Term* **)****| id (***Term***,***Term* **)**

## Разработка СУО

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

 ***Глобальные объекты*:**

 *Idents* – таблица идентификаторов (массив записей), запись таблицы состоит из следующих полей:

 *Lex* – лексема идентификатора (определено при проектировании сканера),

 *Cat* – категория идентификатора (определено при проектировании сканера),

 *Type* – тип идентификатора,

 *Size* – размер занимаемой памяти в байтах (для констант, имен типов и переменных),

 *Addr* – адрес размещения объекта в памяти (для констант и переменных).

 *Lbls* – таблица меток (массив записей), запись таблицы состоит из следующих полей:

 *Lex* – лексема метки (определено при проектировании сканера),

 *Index* – индекс (адрес) команды с данной меткой в едином потоке команд,

 *Lst* – указатель на список адресов команд перехода к этой метке.

 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 *TestLbls*() выполняет проверку, есть ли для метки из правой части оператора, реализующей передачу управления, оператор, помеченный этой меткой. При отсутствии такого оператора формирует сообщение *Type\_Error*(9).

 Функция *NewTemp*() создает новую временную переменную, добавляет ее в таблицу идентификаторов и возвращает указатель на соответствующую запись.

 Процедура *AddType*(*ns*, *Cat*, *Type*, *Size*) устанавливает категорию *Cat*, тип *Type* и размер типа *Size* в записи *ns* таблицы *Idents*, а также размещает переменные и константы в памяти.

 **procedure** *AddType* (*ns*, *Cat*, *Type*, *Size*)

 **begin**

 **if** *Idents*[*ns*].*Cat*≠ *None* **then** *Type\_Error*(1) //категория уже есть, повторное объявление

 *Idents*[*ns*].*Cat* := *Cat* //установка категории идентификатора

*Idents*[*ns*].*Type*:= *Type* //установка типа идентификатора

*Idents*[*ns*].*Size*:= *Size* //установка размера типа

 **if** *Cat* **in** [*catConstName*, *catVarName*] **then**

 **begin** //размещение в памяти

 **while** *NextAddr* **mod** *Size* ≠ 0 **do** *NextAddr* := *NextAddr* + 1 //выравнивание адреса

 *Idents*[*ns*].*Addr*:= *NextAddr* //установка адреса памяти

 *NextAddr* := *NextAddr* + *Size* //адрес очередной свободной памяти

 **end else** *Idents*[*ns*].*Addr*:= –1

 **end**

 В приведенной процедуре выполняется проверка на повторное определяющее вхождение идентификатора с помощью поля категории (поле *Cat*). Лексический анализатор после создания соответствующей записи устанавливает этому полю значение *None* (отсутствие категории). Если это поле не равно *None*, это означает, что идентификатор уже объявлялся, т.е. имеет место повторное объявление.

 Функция *Merge*(*p, i*) добавляет *i* в список команд, на который указывает *p* (*i* – адрес команды в массиве команд), 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 №2);

 *Type\_Error*(3) – не может быть двух операторов с одинаковыми метками (соглашение №11);

 *Type\_Error*(4) – идентификатор не является именем переменной (соглашение №8);

 *Type\_Error*(5) – несовместимость типов (соглашение №7);

 *Type\_Error*(6) – идентификатор не является именем переменной или именем константы (соглашение №6);

 *Type\_Error*(7) – должен быть функциональный идентификатор (соглашение №9);

 *Type\_Error*(8) – терм должен быть типа Boolean (соглашения №10);

 *Type\_Error*(9) – нет оператора, помеченного данной меткой (соглашения №11).

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

 **Примечание для студента**. Здесь в качестве примера приведен только алгоритм процедуры *AddType*. Вы должны привести алгоритмы всех процедур и функций, за исключением простейших, которые только возвращают или устанавливают значение какого-либо поля. Можно также не раскрывать процедуры формирования сообщений об ошибках (в данном примере это процедура *Type\_Error*).

 ***Атрибуты*:**

 Нетерминал *LstId* имеет наследуемые атрибуты *type* (тип), *cat* (категория типа), *size* (размер типа). Нетерминал *Term* имеет синтезируемые атрибуты *type* (тип терма) и *addr* (указатель на запись в таблице идентификаторов).

 Атрибут *ns* терминалов **id** и **lbl** указывает на соответствующую запись в таблице символов, является атрибутом токена, предоставляемого лексическим анализатором.

 Разработанное СУО представлено в табл. 1.

Таблица 1.

СУО для проверки типов и трансляции в трехадресный код языка *MyLang*

| Продукция | Семантические правила |
| --- | --- |
| *S* → **scheme** **id** **;** *Block* **endsch** | *AddType* (**id***.ns, catProgName*, *void*, 0) |
| *Block* → *LstDecl* **start***LstStmt***;***StopStmt* | *TestLbls*() |
| *LstDecl* → *Decl* |  |
| *LstDecl* →  *Decl* *LstDecl*1 |  |
| *Decl* → **id***LstId* **;** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*LstId*.*cat* := *catVarName**LstId*.*type* := *Idents*[**id**.*ns*].*Type**LstId*.*size* := *Idents*[**id**.*ns*].*Size* |
| *Decl* → **const** **id***LstId* **;** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*LstId*.*cat* := *catConstName**LstId*.*type* := *Idents*[**id**.*ns*].*Type**LstId*.*size* := *Idents*[**id**.*ns*].*Size* |
| *Decl* → **functional** **id***LstId* **;** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*LstId*.*cat* := *catFuncName**LstId*.*type* := *Idents*[**id**.*ns*].*Type**LstId*.*size* := *Idents*[**id**.*ns*].*Size* |
| *LstId* → **id** | *AddType*(**id**.*ns*, *LstId*.*cat*, *LstId*.*type*, *LstId*.*size*) |
| *LstId* → **id** **,** *LstId*1 | *LstId*1.*cat* := *LstId*.*cat**LstId*1.*type* := *LstId*.*type**LstId*1.*size* := *LstId*.*size**AddType*(**id**.*ns*, *LstId*.*cat*, *LstId*.*type*, *LstId*.*size*) |
|
| *LstStmt* → *Stmt* |  |
| *LstStmt* →  *Stmt***;** *LstStmt*1 |  |
| *Stmt* → *StmtNoLbl* |  |
| *Stmt* → **lbl :***StmtNoLbl* | **if** *Lbls*[**lbl**.*ns*]*.Index* <> **null** **then** *Type\_Error*(3)*Lbls*[**lbl**.*ns*]*.Index* := *NextInstr**BackPatch*(*Lbls*[**lbl**.*ns*]*.Lst*, *NextInstr*) |
| *StmtNoLbl* → *AssStmt* |  |
| *StmtNoLbl* → *TestStmt* |  |
| *AssStmt* → **id** **ass** *Term* | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catVarName* **then** *Type\_Error*(4)**if** *Idents*[**id**.*ns*].*Type* ≠ *Term*.*type* **then** *Type\_Error*(5)*Gen*(–1, *Term*.*addr*, –1, **id**.*ns*) |
| *AssStmt* → **id** **ass** *Term***goto lbl** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catVarName* **then** *Type\_Error*(4)**if** *Idents*[**id**.*ns*].*Type* ≠ *Term*.*type* **then** *Type\_Error*(5)*Gen*(–1, *Term*.*addr*, –1, **id**.*ns*)**if** *Lbls*[**lbl**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl**.*ns*]*.Lst, NextInstr*) *Gen*(**goto**, –1, –1, –1)**end else** *Gen*(**goto**, –1, –1, *Lbls*[**lbl**.*ns*]*.Index*) |
| *Term* → **id** | **if** *Idents*[**id**.*ns*].*Cat* **not** **in** [*catConstName*, *catVarName*] **then** *Type\_Error*(6)*Term*.*addr* := **id**.*ns**Term*.*type* := *Idents*[**id**.*ns*].*Type* |
| *Term* → **id (***Term*1 **)** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catFuncName* **then** *Type\_Error*(7)*Term*.*addr* := *NewTemp*()*Term*.*type* := *Idents*[**id**.*ns*].*Type**AddType*(*Term*.*addr*, *catVarName*, *Term*.*type*, *Idents*[**id**.*ns*].*Size*)*Gen*(**id**.*ns*, *Term*1.*addr*, –1, *Term*.*addr*) |
| *Term* → **id (***Term*1 **,***Term*2 **)** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catFuncName* **then** *Type\_Error*(7)**if** *Term*1.*type* ≠ *Term*2.*type* **then** *Type\_Error*(5)*Term*.*addr* := *NewTemp*()*Term*.*type* := *Idents*[**id**.*ns*].*Type**AddType*(*Term*.*addr*, *catVarName*, *Term*.*type*, *Idents*[**id**.*ns*].*Size*)*Gen*(**id**.*ns*, *Term*1.*addr*, *Term*2.*addr*, *Term*.*addr*) |
| *TestStmt* → **if** *Term* **then** **lbl1** **else** **lbl2** | **if** *Term*.*type* ≠ *Boolean* **then** *Type\_Error*(8)**if** *Lbls*[**lbl1**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl1**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl1**.*ns*]*.Lst, NextInstr*) *Gen*(**if**, *Term*.*addr*, –1, –1)**end else** *Gen*(**if**, *Term*.*addr*, –1, *Lbls*[**lbl1**.*ns*]*.Index*)**if** *Lbls*[**lbl2**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl2**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl2**.*ns*]*.Lst, NextInstr*) *Gen*(**goto**, –1, –1, –1)**end else** *Gen*(**goto**, –1, –1, *Lbls*[**lbl2**.*ns*]*.Index*) |
| *StopStmt* → **lbl : stop** | **if** *Lbls*[**lbl**.*ns*]*.Index* <> **null** **then** *Type\_Error*(3)*Lbls*[**lbl**.*ns*]*.Index* := *NextInstr**BackPatch*(*Lbls*[**lbl**.*ns*]*.Lst*, *NextInstr*)*Gen*(**stop**, –1, –1, –1) |
| *StopStmt* → **stop** | *Gen*(**stop**, –1, –1, –1) |

 **Примечание для студента**. В приведенном СУО у нетерминалов *LstStmt* и *Stmt* нет атрибутов типа *nextlist* (указатель на список команд переходов), необходимых для метода обратных поправок, поскольку подобные списки формируются в таблице меток *Lbls*. Нет также атрибутов типа *truelist* и *falselist* для нетерминалов для логических выражений, поскольку в языке *MyLang* нет логических выражений. У вас они будут.

# Синтаксически управляемая схема трансляции

 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. 2.

Таблица 2

Схема трансляции (СУТ) для проверки типов и трансляции в трехадресный код языка *MyLang*

| Продукция с действиями | Семантические правила | Семантические действия | Символдействия |
| --- | --- | --- | --- |
| *S* → **scheme** **id** {*A*1} **;** *Block* **endsch** | *AddType* (**id***.ns, catProgName*, *void*, 0) | *AddType* (**id***.ns, catProgName*, *void*, 0) | *A*1 |
| *Block* → *LstDecl* **start***LstStmt***;** *StopStmt* {*A*2} | *TestLbls*() | *TestLbls*() | *A*2 |
| *LstDecl* → *Decl* |  |  |  |
| *LstDecl* →  *Decl* *LstDecl*1 |  |  |  |
| *Decl* → **id** {*A*3}*LstId* **;** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*LstId*.*cat* := *catVarName**LstId*.*type* := *Idents*[**id**.*ns*].*Type**LstId*.*size* := *Idents*[**id**.*ns*].*Size* | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*Push*(*catVarName*, *AttrSt*)*Push*(*Idents*[**id**.*ns*].*Type*, *AttrSt*)*Push*(*Idents*[**id**.*ns*].*Size*, *AttrSt*) | *A*3 |
| *Decl* → **const** **id** {*A*4}*LstId* **;** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*LstId*.*cat* := *catFuncName**LstId*.*type* := *Idents*[**id**.*ns*].*Type**LstId*.*size* := *Idents*[**id**.*ns*].*Size* | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*Push*(*catConstName*, *AttrSt*)*Push*(*Idents*[**id**.*ns*].*Type*, *AttrSt*)*Push*(*Idents*[**id**.*ns*].*Size*, *AttrSt*) | *A*4 |
| *Decl* → **functional** **id** {*A*5}*LstId* **;** | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*LstId*.*cat* := *catFuncName**LstId*.*type* := *Idents*[**id**.*ns*].*Type**LstId*.*size* := *Idents*[**id**.*ns*].*Size* | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catTypeName* **then** *Type\_Error*(2)*Push*(*catFuncName*, *AttrSt*)*Push*(*Idents*[**id**.*ns*].*Type*, *AttrSt*)*Push*(*Idents*[**id**.*ns*].*Size*, *AttrSt*) | *A*5 |
| *LstId* → **id** {*A*6} | *AddType*(**id**.*ns*, *LstId*.*cat*, *LstId*.*type*, *LstId*.*size*) | *t*1 := *Pop*(*AttrSt*) //атрибут *LstId*.*size**t*2 := *Pop*(*AttrSt*) //атрибут *LstId*.*type**t*3 := *StackTop*(*AttrSt*) //атрибут *LstId*.*cat**Push*(*t*2, *AttrSt*) //вернуть в стек атрибут *LstId*.*type**Push*(*t*1, *AttrSt*) //вернуть в стек атрибут *LstId*.*size**AddType*(**id**.*ns*, *t*3, *t*2, *t*1) | *A*6 |
| *LstId* → **id** {*A*6} **,**  *LstId*1 | *AddType*(**id**.*ns*, *LstId*.*cat*, *LstId*.*type*, *LstId*.*size*) | *t*1 := *Pop*(*AttrSt*) //атрибут *LstId*.*size**t*2 := *Pop*(*AttrSt*) //атрибут *LstId*.*type**t*3 := *StackTop*(*AttrSt*) //атрибут *LstId*.*cat**Push*(*t*2, *AttrSt*) //вернуть в стек атрибут *LstId*.*type**Push*(*t*1, *AttrSt*) //вернуть в стек атрибут *LstId*.*size**AddType*(**id**.*ns*, *t*3, *t*2, *t*1) | *A6* |
| *LstId*1.*cat* := *LstId*.*cat**LstId*1.*type* := *LstId*.*type**LstId*1.*size* := *LstId*.*size* | правила копирования, в верхней части стека уже находятся наследуемые атрибуты *LstId*, поэтому действия со стеком атрибутов не нужны |  |
| *LstStmt* → *Stmt* |  |  |  |
| *LstStmt* →  *Stmt***;** *LstStmt*1 |  |  |  |
| *Stmt* → *StmtNoLbl* |  |  |  |
| *Stmt* → **lbl** {*A*7}**:***StmtNoLbl* | **if** *Lbls*[**lbl**.*ns*]*.Index* <> **null** **then** *Type\_Error*(3)*Lbls*[**lbl**.*ns*]*.Index* := *NextInstr**BackPatch*(*Lbls*[**lbl**.*ns*]*.Lst*, *NextInstr*) | **if** *Lbls*[**lbl**.*ns*]*.Index* <> **null** **then** *Type\_Error*(3)*Lbls*[**lbl**.*ns*]*.Index* := *NextInstr**BackPatch*(*Lbls*[**lbl**.*ns*]*.Lst*, *NextInstr*) | *A*7 |
| *StmtNoLbl* → *AssStmt* |  |  |  |
| *StmtNoLbl* → *TestStmt* |  |  |  |
| *AssStmt* → **id** {*A*8} **ass** *Term* {*A*9} | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catVarName* **then** *Type\_Error*(4) | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catVarName* **then** *Type\_Error*(4)*Push*(**id**.*ns*, *AttrSt*) //в стек **id**.*ns* | *A*8 |
| **if** *Idents*[**id**.*ns*].*Type* ≠ *Term*.*type* **then** *Type\_Error*(5)*Gen*(–1, *Term*.*addr*, –1, **id**.*ns*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Term*.*type**t*2 := *Pop*(*AttrSt*) //атрибут *Term*.*addr**t*3 := *Pop*(*AttrSt*) //атрибута **id**.*ns***if** *Idents*[*t*3].*Type* ≠ *t*1 **then** *Type\_Error*(5)*Gen*(–1, *t*2, –1, *t*3) | *A*9 |
| *AssStmt* → **id** {*A*8} **ass** *Term***goto lbl** {*A*10} | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catVarName* **then** *Type\_Error*(4) | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catVarName* **then** *Type\_Error*(4)*Push*(**id**.*ns*, *AttrSt*) //в стек **id**.*ns* | *A*8 |
| **if** *Idents*[**id**.*ns*].*Type* ≠ *Term*.*type* **then** *Type\_Error*(5)*Gen*(–1, *Term*.*addr*, –1, **id**.*ns*)**if** *Lbls*[**lbl**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl**.*ns*]*.Lst, NextInstr*) *Gen*(**goto**, –1, –1, –1)**end else** *Gen*(**goto**, –1, –1, *Lbls*[**lbl**.*ns*]*.Index*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Term*.*type**t*2 := *Pop*(*AttrSt*) //атрибут *Term*.*addr**t*3 := *Pop*(*AttrSt*) //атрибут **id**.*ns***if** *Idents*[*t*3].*Type* ≠ *t*1 **then** *Type\_Error*(5)*Gen*(–1, *t*2, –1, *t*3)**if** *Lbls*[**lbl**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl**.*ns*]*.Lst, NextInstr*) *Gen*(**goto**, –1, –1, –1)**end else** *Gen*(**goto**, –1, –1, *Lbls*[**lbl**.*ns*]*.Index*) | *A*10 |
| *Term* → **id** {*A*11} | **if** *Idents*[**id**.*ns*].*Cat* **not** **in** [*catConstName*, *catVarName*]**then** *Type\_Error*(6)*Term*.*addr* := **id**.*ns**Term*.*type* := *Idents*[**id**.*ns*].*Type* | **if** *Idents*[**id**.*ns*].*Cat* **not** **in** [*catConstName*, *catVarName*]**then** *Type\_Error*(6)*Push*(**id**.*ns*, *AttrSt*) //в стек *Term*.*addr**Push*(*Idents*[**id**.*ns*].*Type*, *AttrSt*) //в стек *Term*.*type* | *A*11 |
| *Term* → **id** {*A*12}**(***Term*1 **)** {*A*13} | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catFuncName* **then** *Type\_Error*(7) | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catFuncName* **then** *Type\_Error*(7)*Push*(**id**.*ns*, *AttrSt*) //в стек **id**.*ns* | *A*12 |
| *Term*.*addr* := *NewTemp*()*Term*.*type* := *Idents*[**id**.*ns*].*Type**AddType*(*Term*.*addr*, *catVarName*, *Term*.*type*, *Idents*[**id**.*ns*].*Size*)*Gen*(**id**.*ns*, *Term*1.*addr*, –1, *Term*.*addr*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Term*1.*type**t*2 := *Pop*(*AttrSt*) //атрибут *Term*1.*addr**t*3 := *Pop*(*AttrSt*) //атрибут **id**.*ns**t*1 := *NewTemp*() //атрибут *Term*.*addr**AddType*(*t*1, *catVarName*, *Idents*[*t*3].*Type*, *Idents*[*t*3].*Size*)*Push*(*t*1, *AttrSt*) //в стек *Term*.*addr**Push*(*Idents*[*t*3].*Type*, *AttrSt*) //в стек *Term*.*type**Gen*(*t*3, *t*2, –1, *t*1) | *A*13 |
| *Term* → **id** {*A*12}**(***Term*1 **,***Term*2 **)** {*A*14} | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catFuncName* **then** *Type\_Error*(7) | **if** *Idents*[**id**.*ns*].*Cat* ≠ *catFuncName* **then** *Type\_Error*(7)*Push*(**id**.*ns*, *AttrSt*) //в стек **id**.*ns* | *A*12 |
| **if** *Term*1.*type* ≠ *Term*2.*type* **then** *Type\_Error*(5)*Term*.*addr* := *NewTemp*()*Term*.*type* := *Idents*[**id**.*ns*].*Type**AddType*(*Term*.*addr*, *catVarName*, *Term*.*type*, *Idents*[**id**.*ns*].*Size*)*Gen*(**id**.*ns*, *Term*1.*addr*, *Term*2.*addr*, *Term*.*addr*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Term*2.*type**t*2 := *Pop*(*AttrSt*) //атрибут *Term*2.*addr**t*3 := *Pop*(*AttrSt*) //атрибут *Term*1.*type**t*4 := *Pop*(*AttrSt*) //атрибут *Term*1.*addr**t*5 := *Pop*(*AttrSt*) //атрибут **id**.*ns***if** *t*3 ≠ *t*1**then** *Type\_Error*(5)*t*1 := *NewTemp*() //атрибут *Term*.*addr**AddType*(*t*1, *catVarName*, *Idents*[*t*5].*Type*, *Idents*[*t*5].*Size*)*Push*(*t*1, *AttrSt*) //в стек *Term*.*addr**Push*(*Idents*[*t*5].*Type*, *AttrSt*) //в стек *Term*.*type**Gen*(*t*5, *t*4, *t*2, *t*1) | *A*14 |
| *TestStmt* → **if** *Term* {*A*15} **then** **lbl1** {*A*16} **else** **lbl2** {*A*17} | **if** *Term*.*type* ≠ *Boolean* **then** *Type\_Error*(8) | *t*1 := *Pop*(*AttrSt*) //атрибут *Term*.*type***if** *t*1 ≠ *Boolean* **then** *Type\_Error*(8) | *A*15 |
| **if** *Lbls*[**lbl1**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl1**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl1**.*ns*]*.Lst, NextInstr*) *Gen*(**if**, *Term*.*addr*, –1, –1)**end else** *Gen*(**if**, *Term*.*addr*, –1, *Lbls*[**lbl1**.*ns*]*.Index*) | *t*1 := *Pop*(*AttrSt*) //атрибут *Term*.*addr***if** *Lbls*[**lbl1**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl1**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl1**.*ns*]*.Lst, NextInstr*) *Gen*(**if**, *t*1, –1, –1)**end else** *Gen*(**if**, *t*1, –1, *Lbls*[**lbl1**.*ns*]*.Index*) | *A*16 |
| **if** *Lbls*[**lbl2**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl2**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl2**.*ns*]*.Lst, NextInstr*) *Gen*(**goto**, –1, –1, –1)**end else** *Gen*(**goto**, –1, –1, *Lbls*[**lbl2**.*ns*]*.Index*) | **if** *Lbls*[**lbl2**.*ns*]*.Index* = **null** **then****begin** *Lbls*[**lbl2**.*ns*]*.Lst* := *Merge*(*Lbls*[**lbl2**.*ns*]*.Lst, NextInstr*) *Gen*(**goto**, –1, –1, –1)**end else** *Gen*(**goto**, –1, –1, *Lbls*[**lbl2**.*ns*]*.Index*) | *A*17 |
| *StopStmt* → **lbl** {*A*7}**: stop** {*A*18} | **if** *Lbls*[**lbl**.*ns*]*.Index* <> **null** **then** *Type\_Error*(3)*Lbls*[**lbl**.*ns*]*.Index* := *NextInstr**BackPatch*(*Lbls*[**lbl**.*ns*]*.Lst*, *NextInstr*) | **if** *Lbls*[**lbl**.*ns*]*.Index* <> **null** **then** *Type\_Error*(3)*Lbls*[**lbl**.*ns*]*.Index* := *NextInstr**BackPatch*(*Lbls*[**lbl**.*ns*]*.Lst*, *NextInstr*) | *A*7 |
| *Gen*(**stop**, –1, –1, –1) | *Gen*(**stop**, –1, –1, –1) | *A*18 |
| *StopStmt* → **stop** {*A*18} | *Gen*(**stop**, –1, –1, –1) | *Gen*(**stop**, –1, –1, –1) | *A*18 |

 **Комментарии к табл. 2.**

 Действия *A*3, *A*4 и *A*5 размещены перед нетерминалом *LstId*, поскольку в них вычисляются наследуемые атрибуты *LstId*. В этих действиях также проверяется, является ли **id** именем типа.

 Для продукции *LstId* → **id** {*A*6} **,** *LstId*1семантические правила разбиты на 2 группы, поскольку первая группа (действие *A*6) сразу после получения от сканера токена **id** должна занести данные об **id** в таблицу идентификаторов, а вторая группа – установить наследуемые атрибуты *LstId*1, в действии, непосредственно предшествующем этому нетерминалу. Поскольку во второй группе применяются правила копирования, а в верхней части стека уже находятся наследуемые атрибуты *LstId*, действия со стеком атрибутов не нужны. Поэтому в действии *A*6 после исключения из стека атрибутов нетерминала *LstId* и их использования они снова должны быть возвращены в стек, поскольку эти атрибуты являются наследуемыми и могут понадобиться для последующего разбора (время их жизни еще не завершилось).

 В продукции *Stmt* → **lbl** {*A*7}**:***StmtNoLbl* действие *A*7 размещено после **lbl**, чтобы в случае обнаружения ошибки прекратить последующий разбор нетерминала *StmtNoLbl*), т. е. действие *A*7 выполнить сразу после получения **lbl** от сканера. Аналогично для продукции *AssStmt* → **id** **ass** *Term* правила разбиты на группы *A*8 и *A*9, чтобы в случае, если **id** не является переменной (семантическая ошибка), прекратить последующий разбор терма (нетерминал *Term*). По этим же причинам выполнено разбиение правил для продукции *AssStmt* → **id** **ass** *Term* **goto lbl**.

 Для продукций *Term* → **id (***Term*1**)** и *Term* → **id (***Term*1**,** *Term*2**)** также нет смысла продолжать разбор в случае, если **id** не является функциональным идентификатором.

 В продукциях с нетерминалом *Decl* в левой части для экономии памяти в конце правых частей продукций можно было бы выполнить действия по удалению из стека трех наследуемых атрибутов нетерминала *LstId*, поскольку время их жизни закончилось. Но поскольку они не влияют на последующий разбор, очистка стека атрибутов будет выполнена после завершения трансляции (честно говоря, не хотелось перенумеровывать действия, возможно позже добавлю действия).

 **Примечание для студента**. Приведенные выше комментарии к табл. 2 можете не включать в отчет по РГР. Главное, чтобы Вы могли обосновать размещение символов действий при защите РГР.

 Получена следующая СУТ:

 *S* → **scheme** **id** {*A*1} **;** *Block* **endsch**

 *Block* → *LstDecl* **start***LstStmt***;** *StopStmt*{*A*2}

 *LstDecl* → *Decl* **|** *Decl* *LstDecl*

 *Decl* → **id**{*A*3}*LstId* **;** **|** **const** **id**{*A*4}*LstId* **;** **|** **functional** **id**{*A*5}*LstId* **;**

 *LstId* → **id** {*A*6} **| id** {*A*6} **,** *LstId*

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

 *Stmt* → *StmtNoLbl* **| lbl** {*A*7} **:***StmtNoLbl*

 *StmtNoLbl* → *AssStmt* **|** *TestStmt*

 *AssStmt* → **id** {*A*8} **ass** *Term* {*A*9} **| id** {*A*8} **ass** *Term***goto lbl** {*A*10}

 *Term* → **id** {*A*11} **| id** {*A*12} **(***Term***)** {*A*13} **| id** {*A*12} **(***Term***,***Term***)** {*A*14}

 *TestStmt* → **if** *Term* {*A*15} **then** **lbl** {*A*16} **else** **lbl** {*A*17}

 *StopStmt* → **lbl** {*A*7} **: stop** {*A*18} **| stop** {*A*18}

 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**

 **Примечание для студента**. В данной грамматике нет левой рекурсии. У Вас обязательно будут леворекурсивные продукции (по крайней мере для арифметических выражений), поэтому надо начать преобразования с устранения левой рекурсии, а только затем применять при необходимости левую факторизацию.

 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 *LstDecl* → *Decl* **|** *Decl* *LstDecl*

 *LstDecl* → *Decl* **〈**ε **|** *LstDecl***〉**

получим

 *LstDecl* → *Decl* *A*

 *A* → ε **|** *LstDecl*

Факторизация *LstId* → **id** {*A*6} **| id** {*A*6} **,** *LstId*

 *LstId* → **id** {*A*6} **〈**ε **|** **,** *LstId***〉**

получим

 *LstId* → **id** {*A*6} *B*

 *B* → ε **|** **,** *LstId*

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

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

получим

 *LstStmt* → *Stmt* *C*

 *C* → ε **|** **;** *LstStmt*

Факторизация *AssStmt* → **id** {*A*8} **ass** *Term* {*A*9} **| id** {*A*8} **ass** *Term***goto lbl** {*A*10}

 *AssStmt* → **id** {*A*8} **ass** *Term* **〈**{*A*9} **|****goto lbl** {*A*10}**〉**

получим

 *AssStmt* → **id** {*A*8} **ass** *Term* *D*

 *D* → {*A*9} **|****goto lbl** {*A*10}

Факторизация *Term* → **id** {*A*11} **|** **id** {*A*12} **(** *Term***)** {*A*13} **|** **id** {*A*12} **(***Term***,***Term***)** {*A*14}

 *Term* → **id** **〈**{*A*11} **|** {*A*12} **(** *Term***)** {*A*13} **|** {*A*12} **(***Term***,***Term***)** {*A*14}]**〉**

получим

 *Term* → **id** *E*

 *E* → {*A*11} **|** {*A*12} **(***Term***)** {*A*13} **|** {*A*12} **(** *Term***,***Term***)** {*A*14}

Факторизация *E* → {*A*12} **(***Term***)** {*A*13} **|** {*A*12} **(** *Term***,***Term***)** {*A*14}

 *E* → {*A*11} **|** {*A*12} **(***Term* **〈****)** {*A*13} **|** **,** *Term***)** {*A*14}**〉**

получим

 *E* → {*A*11} **|** {*A*12} **(***Term* *F*

 *F* → **)** {*A*13} **|** **,***Term***)** {*A*14}

Получили грамматику

 *S* → **scheme** **id** {*A*1} **;** *Block* **endsch**

 *Block* → *LstDecl* **start***LstStmt***;** *StopStmt*{*A*2}

 *LstDecl* → *Decl* *A*

 *A* → ε **|** *LstDecl*

 *Decl* → **id**{*A*3}*LstId* **;** **|** **const** **id**{*A*4}*LstId* **;** **|** **functional** **id**{*A*5}*LstId* **;**

 *LstId* → **id** {*A*6} *B*

 *B* → ε **|** **,** *LstId*

 *LstStmt* → *Stmt* *C*

 *C* → ε **|** **;** *LstStmt*

 *Stmt* → *StmtNoLbl* **|** **lbl** {*A*7} **:** *StmtNoLbl*

 *StmtNoLbl* → *AssStmt* **|** *TestStmt*

 *AssStmt* → **id** {*A*8} **ass** *Term* *D*

 *D* → {*A*9} **|****goto** **lbl** {*A*10}

 *Term* → **id** *E*

 *E* → {*A*11} **|** {*A*12} **(** *Term* *F*

 *F* → **)** {*A*13} **|** **,** *Term***)** {*A*14}

 *TestStmt* → **if** *Term* {*A*15} **then** **lbl** {*A*16} **else** **lbl** {*A*17}

 *StopStmt* → **lbl** {*A*7} **:** **stop** {*A*18} **|** **stop** {*A*18}

 Еще не *LL*(1)-грамматика, поскольку *DS*(*C* → ε) = {;} и *DS*(*C* → **;** *LstStmt*) = {;}. Продолжим преобразование. Включим терминал **;** в конструкцию *LstStmt*. Получим

 *Block* → *LstDecl* **start***LstStmt StopStmt*{*A*2}

 *LstStmt* → *Stmt* **;** *C*

 *C* → ε **|** *LstStmt*

Получили грамматику

 *S* → **scheme** **id** {*A*1} **;** *Block* **endsch**

 *Block* → *LstDecl* **start***LstStmt StopStmt*{*A*2}

 *LstDecl* → *Decl* *A*

 *A* → ε **|** *LstDecl*

 *Decl* → **id**{*A*3}*LstId* **;** **|** **const** **id**{*A*4}*LstId* **;** **|** **functional** **id**{*A*5}*LstId* **;**

 *LstId* → **id** {*A*6} *B*

 *B* → ε **|** **,** *LstId*

 *LstStmt* → *Stmt* **;** *C*

 *C* → ε **|** *LstStmt*

 *Stmt* → *StmtNoLbl* **|** **lbl** {*A*7} **:** *StmtNoLbl*

 *StmtNoLbl* → *AssStmt* **|** *TestStmt*

 *AssStmt* → **id** {*A*8} **ass** *Term* *D*

 *D* → {*A*9} **|****goto** **lbl** {*A*10}

 *Term* → **id** *E*

 *E* → {*A*11} **|** {*A*12} **(** *Term* *F*

 *F* → **)** {*A*13} **|** **,** *Term***)** {*A*14}

 *TestStmt* → **if** *Term* {*A*15} **then** **lbl** {*A*16} **else** **lbl** {*A*17}

 *StopStmt* → **lbl** {*A*7} **: stop** {*A*18} **|** **stop** {*A*18}

 Все еще не *LL*(1)-грамматика, поскольку есть общий терминал **lbl** в множествах *DS*(*C* → ε) = {**lbl**, **stop**} и *DS*(*C* → *LstStmt*) = {**id**, **lbl**, **if**}. Продолжим преобразование.

 В *DS*(*C* → ε) терминал **lbl** ∈ *Follow*(*С*) = *Follow*(*LstStmt*), согласно продукции *Block* → *LstDecl* **start***LstStmt StopStmt*{*A*2} терминал **lbl** ∈ *First*(*StopStmt*), т. е. является первым символом некоторых строк, выводимых из *StopStmt*. В *DS*(*C* → *LstStmt*) терминал **lbl** ∈ *First*(*LstStmt*) = *First*(*Stmt*). Поэтому этот направляющий символ (терминал **lbl**) следует последовательностью замен вхождений «продвинуть» в продукцию *Block* → *LstDecl* **start***LstStmt StopStmt*{*A*2}.

 Для локализации преобразований последовательность нетерминалов *LstStmt StopStmt* обозначим новым нетерминалом *G* и добавим продукцию *G* → *LstStmt StopStmt*. В результате получим

 *Block* → *LstDecl* **start***G*{*A*2}

 *G* → *LstStmt StopStmt*

Определим подмножество необходимых для преобразований продукций:

 *G* → *LstStmt StopStmt*

 *LstStmt* → *Stmt* **;** *C*

 *C* → ε **|** *LstStmt*

 *Stmt* → *StmtNoLbl* **|** **lbl** {*A*7} **:***StmtNoLbl*

 *StopStmt* → **lbl** {*A*7} **:** **stop** {*A*18} **|** **stop** {*A*18}

 Здесь нет *StmtNoLbl*-продукций, поскольку в множества *DS* этих продукций не входит терминал **lbl**, т, е. они не влияют на необходимые преобразования.

 Для удобства преобразований запишем эти продукции в сокращенном виде, используя следующие обозначения:

 *L* – *LstStmt*

 *S* – *StopStmt*

 *O* – *Stmt*

 *n* – *StmtNoLbl*

 *l* – **lbl** {*A*7} **:**

 *s* – **stop** {*A*18}

Продукции в сокращенном виде:

 *G* → *LS*

 *L* → *O***;***C*

 *C* → ε **|** *L*

 *O* → *n* **|** *ln*

 *S* → *s* **|** *ls*

Замена вхождений *C* в *L* → *O***;***C*

 *L* → *O***;** **|** *O***;***L*

Замена вхождений *O* в полученных *L*-продукциях

 *L* → *n***;** **|** *n***;***L* **|** *ln***;** **|** *ln***;***L*

Замена вхождений *L* в *G* → *LS*

 *G* → *n***;***S***|** *n***;***LS* **|** *ln***;***S* **|** *ln***;***LS*

Замена вхождений *S* в полученных *G*-продукциях

 *G* → *n***;***s***|** *n***;***Ls* **|** *ln***;***s* **|** *ln***;***Ls* **|** *n***;***ls***|** *n***;***Lls* **|** *ln***;***ls* **|** *ln***;***Lls*

Факторизация *G* → *n***;***s***|** *n***;***Ls* **|** *ln***;***s* **|** *ln***;***Ls* **|** *n***;***ls***|** *n***;***Lls* **|** *ln***;***ls* **|** *ln***;***Lls*

 *G* → *n***;〈***s***|** *Ls* **|** *ls***|** *Lls***〉** **|** *ln***;〈***s* **|** *Ls* **|** *ls* **|** *Lls***〉**

получим

 *G* → *n***;***H* **|** *ln***;***H*

 *H* → *s***|** *Ls* **|** *ls***|** *Lls*

Замена вхождений *L* в *H* → *Ls* **|** *Lls*

 *H* → *s* **|** *ls***|** *n***;***s***|** *n***;***ls***|** *n***;***Ls***|** *n***;***Lls***|** *ln***;***s***|** *ln***;***ls***|** *ln***;***Ls***|** *ln***;***Lls*

Факторизация *H* → *n***;***s***|** *n***;***ls***|** *n***;***Ls***|** *n***;***Lls***|** *ln***;***s***|** *ln***;***ls***|** *ln***;***Ls***|** *ln***;***Lls*

 *H* → *s* **|** *ls***|** *n***;〈***s***|** *ls***|** *Ls***|** *Lls***〉****|** *ln***;〈***s***|** *ls***|** *Ls***|** *Lls***〉**

в скобках **〈***s***|** *ls***|** *Ls***|** *Lls***〉** не что иное, как *H*, получаем

 *H* → *s* **|** *ls***|** *n***;***H***|** *ln***;***H*

Факторизация *H* → *ls***|** *ln***;***H*

 *H* → *s* **|** *n***;***H* **|** *l***〈***s***|** *n***;***H***〉**

получаем

 *H* → *s* **|** *n***;***H* **|** *lK*

 *K* → *s***|** *n***;***H*

тогда с учетом *K* → *s***|** *n***;***H* получим

 *H* → *K* **|** *lK*

Результат преобразований:

 *G* → *n***;***H* **|** *ln***;***H*

 *H* → *K* **|** *lK*

 *K* → *s***|** *n***;***H*

Вернемся к исходным обозначениям символов грамматики:

 *G* → *StmtNoLbl* **;** *H* **|** **lbl** {*A*7} **:** *StmtNoLbl* **;** *H*

 *H* → *K* **|** **lbl** {*A*7} **:** *K*

 *K* → **stop** {*A*18}**|** *StmtNoLbl* **;** *H*

 **Примечание для студента**. 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 получим *LL*(1)-грамматику, ориентированную на нисходящую синтаксически управляемую трансляцию:

 *S* → **scheme** **id** {*A*1} **;** *Block* **endsch**

 *Block* → *LstDecl* **start***G*{*A*2}

 *LstDecl* → *Decl* *A*

 *A* → ε **|** *LstDecl*

 *Decl* → **id**{*A*3}*LstId* **;** **|** **const** **id**{*A*4}*LstId* **;** **|** **functional** **id**{*A*5}*LstId* **;**

 *LstId* → **id** {*A*6} *B*

 *B* → ε **|** **,** *LstId*

 *G* → *StmtNoLbl* **;** *H* **|** **lbl** {*A*7} **:** *StmtNoLbl* **;** *H*

 *H* → *K* **|** **lbl** {*A*7} **:** *K*

 *K* → **stop** {*A*18}**|** *StmtNoLbl* **;** *H*

 *StmtNoLbl* → *AssStmt* **|** *TestStmt*

 *AssStmt* → **id** {*A*8} **ass** *Term* *D*

 *D* → {*A*9} **|****goto** **lbl** {*A*10}

 *Term* → **id** *E*

 *E* → {*A*11} **|** {*A*12} **(** *Term* *F*

 *F* → **)** {*A*13} **|** **,** *Term***)** {*A*14}

 *TestStmt* → **if** *Term* {*A*15} **then** **lbl** {*A*16} **else** **lbl** {*A*17}

 Направляющие символы и схема грамматики приведены в табл. 3, а *LL*(1)-таблица разбора – в табл. 4. Для построения этих таблиц использован генератор таблиц разбора (приложение TblGenerator.exe).

Таблица 3

Направляющие символы и схема грамматики

| №п/п | Продукция | Направляющиесимволы (DS) | Схемаграмматики |
| --- | --- | --- | --- |
| 1 | S = "scheme" "id" {A1} ";" Block "endsch" | scheme | 1 = 2 3 4 5 6 7 |
| 2 | Block = LstDecl "start" G {A2} | id, const, functional | 8 = 9 10 11 12 |
| 3 | LstDecl = Decl A | id, const, functional | 13 = 14 15 |
| 4 | A = | start | 16 = 18 |
| 5 | A = LstDecl | id, const, functional | 17 = 19 |
| 6 | Decl = "id" {A3} LstId ";" | id | 20 = 23 24 25 26 |
| 7 | Decl = "const" "id" {A4} LstId ";" | const | 21 = 27 28 29 30 31 |
| 8 | Decl = "functional" "id" {A5} LstId ";" | functional | 22 = 32 33 34 35 36 |
| 9 | LstId = "id" {A6} B | id | 37 = 38 39 40 |
| 10 | B = | ; | 41 = 43 |
| 11 | B = "," LstId | , | 42 = 44 45 |
| 12 | G = StmtNoLbl ";" H | id, if | 46 = 48 49 50 |
| 13 | G = "lbl" {A7} ":" StmtNoLbl ";" H | lbl | 47 = 51 52 53 54 55 56 |
| 14 | H = K | id, stop, if | 57 = 59 |
| 15 | H = "lbl" {A7} ":" K | lbl | 58 = 60 61 62 63 |
| 16 | K = "stop" {A18} | stop | 64 = 66 67 |
| 17 | K = StmtNoLbl ";" H | id, if | 65 = 68 69 70 |
| 18 | StmtNoLbl = AssStmt | id | 71 = 73 |
| 19 | StmtNoLbl = TestStmt | if | 72 = 74 |
| 20 | AssStmt = "id" {A8} "ass " Term D | id | 75 = 76 77 78 79 80 |
| 21 | D = {A9} | ; | 81 = 83 |
| 22 | D = "goto" "lbl" {A10} | goto | 82 = 84 85 86 |
| 23 | Term = "id" E | id | 87 = 88 89 |
| 24 | E = {A11} | ;, ,, goto, ), then | 90 = 92 |
| 25 | E = {A12} "(" Term F | ( | 91 = 93 94 95 96 |
| 26 | F = ")" {A13} | ) | 97 = 99 100 |
| 27 | F = "," Term ")" {A14} | , | 98 = 101 102 103 104 |
| 28 | TestStmt = "if" Term {A15} "then" "lbl" {A16} "else" "lbl" {A17} | if | 105 = 106 107 108 109 110 111 112 113 114 |
| LL(1)-грамматика |

Таблица 4

*LL*(1)-таблица разбора

| № | Terminals | Jump | Accept | Stack | Action | Error |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | scheme | 2 | false | false |  | true |
| 2 | scheme | 3 | true | false |  | true |
| 3 | id | 4 | true | false |  | true |
| 4 |  | 5 | false | false | A1 | true |
| 5 | ; | 6 | true | false |  | true |
| 6 | id, const, functional | 8 | false | true |  | true |
| 7 | endsch | 0 | true | false |  | true |
| 8 | id, const, functional | 9 | false | false |  | true |
| 9 | id, const, functional | 13 | false | true |  | true |
| 10 | start | 11 | true | false |  | true |
| 11 | id, lbl, if | 46 | false | true |  | true |
| 12 |  | 0 | false | false | A2 | true |
| 13 | id, const, functional | 14 | false | false |  | true |
| 14 | id, const, functional | 20 | false | true |  | true |
| 15 | id, start, const, functional | 16 | false | false |  | true |
| 16 | start | 18 | false | false |  | false |
| 17 | id, const, functional | 19 | false | false |  | true |
| 18 | start | 0 | false | false |  | true |
| 19 | id, const, functional | 13 | false | false |  | true |
| 20 | id | 23 | false | false |  | false |
| 21 | const | 27 | false | false |  | false |
| 22 | functional | 32 | false | false |  | true |
| 23 | id | 24 | true | false |  | true |
| 24 |  | 25 | false | false | A3 | true |
| 25 | id | 37 | false | true |  | true |
| 26 | ; | 0 | true | false |  | true |
| 27 | const | 28 | true | false |  | true |
| 28 | id | 29 | true | false |  | true |
| 29 |  | 30 | false | false | A4 | true |
| 30 | id | 37 | false | true |  | true |
| 31 | ; | 0 | true | false |  | true |
| 32 | functional | 33 | true | false |  | true |
| 33 | id | 34 | true | false |  | true |
| 34 |  | 35 | false | false | A5 | true |
| 35 | id | 37 | false | true |  | true |
| 36 | ; | 0 | true | false |  | true |
| 37 | id | 38 | false | false |  | true |
| 38 | id | 39 | true | false |  | true |
| 39 |  | 40 | false | false | A6 | true |
| 40 | ;, , | 41 | false | false |  | true |
| 41 | ; | 43 | false | false |  | false |
| 42 | , | 44 | false | false |  | true |
| 43 | ; | 0 | false | false |  | true |
| 44 | , | 45 | true | false |  | true |
| 45 | id | 37 | false | false |  | true |
| 46 | id, if | 48 | false | false |  | false |
| 47 | lbl | 51 | false | false |  | true |
| 48 | id, if | 71 | false | true |  | true |
| 49 | ; | 50 | true | false |  | true |
| 50 | id, lbl, stop, if | 57 | false | false |  | true |
| 51 | lbl | 52 | true | false |  | true |
| 52 |  | 53 | false | false | A7 | true |
| 53 | : | 54 | true | false |  | true |
| 54 | id, if | 71 | false | true |  | true |
| 55 | ; | 56 | true | false |  | true |
| 56 | id, lbl, stop, if | 57 | false | false |  | true |
| 57 | id, stop, if | 59 | false | false |  | false |
| 58 | lbl | 60 | false | false |  | true |
| 59 | id, stop, if | 64 | false | false |  | true |
| 60 | lbl | 61 | true | false |  | true |
| 61 |  | 62 | false | false | A7 | true |
| 62 | : | 63 | true | false |  | true |
| 63 | id, stop, if | 64 | false | false |  | true |
| 64 | stop | 66 | false | false |  | false |
| 65 | id, if | 68 | false | false |  | true |
| 66 | stop | 67 | true | false |  | true |
| 67 |  | 0 | false | false | A18 | true |
| 68 | id, if | 71 | false | true |  | true |
| 69 | ; | 70 | true | false |  | true |
| 70 | id, lbl, stop, if | 57 | false | false |  | true |
| 71 | id | 73 | false | false |  | false |
| 72 | if | 74 | false | false |  | true |
| 73 | id | 75 | false | false |  | true |
| 74 | if | 105 | false | false |  | true |
| 75 | id | 76 | false | false |  | true |
| 76 | id | 77 | true | false |  | true |
| 77 |  | 78 | false | false | A8 | true |
| 78 | ass | 79 | true | false |  | true |
| 79 | id | 87 | false | true |  | true |
| 80 | ;, goto | 81 | false | false |  | true |
| 81 | ; | 83 | false | false |  | false |
| 82 | goto | 84 | false | false |  | true |
| 83 |  | 0 | false | false | A9 | true |
| 84 | goto | 85 | true | false |  | true |
| 85 | lbl | 86 | true | false |  | true |
| 86 |  | 0 | false | false | A10 | true |
| 87 | id | 88 | false | false |  | true |
| 88 | id | 89 | true | false |  | true |
| 89 | ;, ,, goto, (, ), then | 90 | false | false |  | true |
| 90 | ;, ,, goto, ), then | 92 | false | false |  | false |
| 91 | ( | 93 | false | false |  | true |
| 92 |  | 0 | false | false | A11 | true |
| 93 |  | 94 | false | false | A12 | true |
| 94 | ( | 95 | true | false |  | true |
| 95 | id | 87 | false | true |  | true |
| 96 | ,, ) | 97 | false | false |  | true |
| 97 | ) | 99 | false | false |  | false |
| 98 | , | 101 | false | false |  | true |
| 99 | ) | 100 | true | false |  | true |
| 100 |  | 0 | false | false | A13 | true |
| 101 | , | 102 | true | false |  | true |
| 102 | id | 87 | false | true |  | true |
| 103 | ) | 104 | true | false |  | true |
| 104 |  | 0 | false | false | A14 | true |
| 105 | if | 106 | false | false |  | true |
| 106 | if | 107 | true | false |  | true |
| 107 | id | 87 | false | true |  | true |
| 108 |  | 109 | false | false | A15 | true |
| 109 | then | 110 | true | false |  | true |
| 110 | lbl | 111 | true | false |  | true |
| 111 |  | 112 | false | false | A16 | true |
| 112 | else | 113 | true | false |  | true |
| 113 | lbl | 114 | true | false |  | true |
| 114 |  | 0 | false | false | A17 | 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], 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 [4].

 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 *TR* 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:

 **procedure** *SUT*

 **begin**

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

 *S* ⇐ 0 //начальное состояние стека

 *la* := **true**

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

 *Sym* := *Token*.*Code* //*Sym* – очередной входной символ

 **while** (*Sym* ≠ *EndTrm*) **and** (*i* ≠ 0) **do**

 **begin**

 **if** (*TR*[*i*].*Action* ≠ 0) **or** (*Sym* ∈ *TR*[*i*].*Terminals*) **then**

 **begin**

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

 *Semantic\_Action*(*TR*[*i*].*Action*) //выполнить действия

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

 **if** *TR*[*i*].*Jump* ≠ 0 **then**

 **begin**

 **if** *TR*[*i*].*Stack* **then** *S* ⇐ *i* + 1

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

 **end else** *i* ⇐ *S*

 **end else**

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

 **begin**

 *i* := *i* + 1

 *la* := **false**

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

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

 **begin**

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

 *Sym* := *Token*.*Code* //*Sym* – очередной входной символ

 **end**

 **end** //while

 **end** //procedure *SUT*

 Данный алгоритм является модификацией алгоритма из [4]. 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.

 В переменной *EndTrm* хранится терминал, выполняющий функции маркера конца ввода.

 Процедура *Syntax\_Error*(*i*) генерирует сообщение о синтаксической ошибке по номер *i* строки, в которой была зафиксирована ошибка. Процедура *Semantic\_Action*(*TR*[*i*].*Action*) по коду действия (параметр *TR*[*i*].*Action*) выбирает соответствующую подпрограмму, реализующую действие.

 **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. Для синтаксических ошибок предусмотрено единственное сообщение «SynErr: Недопустимый токен», поскольку предусмотрено визуальное выделение ошибочной лексемы, а также указание номера строки и позиции в строке этой лексемы.

 **Примечание для студента**. 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.

 Проиллюстрируем это на примере полученной таблицы разбора. Пусть ошибка возникла в состоянии (строке) 46. Это означает, что в данном состоянии в качестве входного токена может быть только **id** или **if** (соответствующие терминалы в поле *Terminals*). Заметим, что точно эти же входные токены допустимы и в состояниях 48, 54, 65 и 68. Таким образом, разные состояния анализатора могут иметь одну и ту же семантику ошибок. Это следует учесть при определении перечня формулировок сообщений о синтаксических ошибках.

Таблица 5

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

|  |  |
| --- | --- |
| Код ошибки | Название ошибки |
|  | ***Ошибки переполнения*** |
| BndErr\_1 | Переполнение таблицы идентификаторов (не более 255) |
| BndErr\_2 | Переполнение таблицы меток (не более 255) |
| BndErr\_3 | Переполнение памяти данных (не более 255 байт) |
| BndErr\_4 | Переполнение памяти команд (не более 255 команд) |
|  | ***Семантические ошибки*** |
| TypeErr\_1 | Повторное объявление идентификатора |
| TypeErr\_2 | Идентификатор не является именем типа |
| TypeErr\_3 | Не может быть двух операторов с одинаковыми метками |
| TypeErr\_4 | Идентификатор не является именем переменной |
| TypeErr\_5 | Несовместимость типов |
| TypeErr\_6 | Идентификатор не является именем переменной или именем константы |
| TypeErr\_7 | Должен быть функциональный идентификатор |
| TypeErr\_8 | Терм должен быть типа Boolean |
| TypeErr\_9 | Нет оператора, помеченного данной меткой |
|  | ***Синтаксические ошибки*** |
| SynErr | Недопустимый токен |

# 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.

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

 связный список адресов команд для tElemLbl

tPnt = ^tElSp;

tElSp = **record** Info: integer; Next: tPnt; **end**;

 таблица меток

tElemLbl = **record** Lex: string; Index: integer; Lst: tPnt; **end**;

Lbls: **array** [1..MaxLbl] **of** tElemLbl;

 таблица идентификаторов

tElemIde = **record** Lex: string; Cat, Tip, Size: Byte; Addr: integer; **end**;

Idents: **array** [1..MaxId] **of** tElemIde;

 структура токена

tToken = **record** Code, Attr: Byte; Name: string; LexBeg, LexEnd: integer; **end**;

 память трехадресных команд (четверки)

tInst = **record** Op, Arg1, Arg2, Res: integer; **end**;

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

 память данных

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

 таблица разбора

tElemTR = **record** ElType, ElPar, Left, Act: Byte; **end**;

TR: **array** [1..MaxSostTR, 1..MaxSymTR] **of** tElemTR;

TR: **array** [1..39] **of** tElemLL, где элемент таблицы имеет следующую структуру:

tElemLL = **record**

 Terminals:tPointSet; //множество

 Jump:Word;

 Accept,Stack,Error:Boolean;

 Action:Byte;

 **end**;

Назначение полей и перечень их возможных значений подробно рассмотрены в разделе 4.

 По сравнению с РГР №1 структура токена tToken модифицирована, а именно: добавлены поля *Name* (обозначение токена для сопоставления с номером столбца в таблице разбора), *LexBeg* (начальная позиция лексемы в тексте) и *LexEnd* (конечная позиция лексемы в тексте). *LexBeg* и *LexEnd* добавлены для визуального выделения ошибочной лексемы в исходном тексте транслируемой программы. Выполнена модификация сканера в части установки значений этих полей (в соответствующих конечных состояниях автомата сканера).

 ***Процедуры и функции*:**

 СУ-трансляция реализована в виде функции

 **function** SUT: Boolean,

возвращающей **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.

 Выполнение семантических действий реализовано функцией

 **function** Semantic\_Action (ActionCode: Byte; Token: tToken): Boolean;

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** TestLbls: Boolean;

 **function** NewTemp: integer;

 **function** AddType(ns: integer; Cat, Tip, Size:Byte; Token: tToken): Boolean;

 **function** Merge(p: tPnt; i: integer): tPnt;

 **procedure** BackPatch(var p: tPnt; i: integer);

 **procedure** Gen(Op, Arg1, Arg2, Res: integer);

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

 Семантические действия *A*1, …, *A*18 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** A16 (Token: tToken);

 в) действия, в которых нет обнаружения ошибок и нет входных параметров, реализованы процедурами, например, **procedure** A18;

 Для отображения результатов трансляции и распределения памяти реализованы подпрограммы:

 **function** OutInstr(Ind: integer): string; //отображение трехадресной команды 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 «LL-таблица» предназначена для импорта *LL*(1)-таблицы разбора из xml-файла, сформированного генератором таблиц разбора.

 Для запуска СУ-трансляции служит кнопка «СУТ». Результаты трансляции отражаются в элементе «Результаты тестирования», 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 нисходящий 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*. Нисходящий синтаксический анализ: конспект лекций / Л.А. Павлов.– Чебоксары: Изд-во Чуваш. ун-та, 2003. – 48 с.

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

 6. *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.

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

 8. 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.