**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-41-22

Иванов В.С.

**Руководитель:**

доцент Павлов Л.А.

Чебоксары 2025

**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 24](#_Toc431747933)

[5. Программная реализация СУ-трансляции 26](#_Toc431747934)

[Заключение 29](#_Toc431747935)

[Список использованной литературы 29](#_Toc431747936)

Задание к РГР

1. Проработать все теоретические вопросы, связанные с восходящей синтаксически управляемой трансляцией языка *MyScala*, 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 восходящей 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. Модуль = "Program" Идент ";" Блок "endProg".
2. Идент = Буква { Буква | Цифpа }.
3. Блок = { СписокОписаний } "begin" ПоследОператоров "end".
4. СписокОписаний = Описание { ";" Описание } ";".
5. Описание = Тип СписокИдент.
6. СписокИдент = Идент { "," Идент }.
7. Тип = ПростойТип | ТипМассив.
8. ПростойТип = Идент.
9. ТипМассив = ПростойТип "[" Целое "]".
10. ПоследОператоров = Оператор { ";" Оператор }.
11. Оператор = Присваивание | Цикл | Условие.
12. Присваивание = Переменная "=" Выражение.
13. Переменная = Идент | ИндексПеременная.
14. ИндексПеременная = Идент "[" Простое выражение "]"
15. Repeat = "repeat" ПоследОператоров "until" Выражение.
16. Выражение = ПростоеВыраж [ Отношение ПростоеВыраж ].
17. Отношение = "<" | "<=" | ">" | ">=" | "==" | "!=".
18. ПростоеВыраж = Терм { АддитОперация Терм }.
19. АддитОперация = "+" | "–" | "||".
20. Терм = Фактор { МультОперация Фактор }.
21. МультОперация = "\*" | "/" | "&&".
22. Фактор = Константа | Пеpеменная | "(" Выражение ")" | "not" Фактор.
23. Константа = ЧисловаяКонст.
24. ЧисловаяКонст = Целое [ "." Целое ] ["e" [ ("+" | "–" ) ] Целое ].
25. Целое = Цифpа { Цифpа }.
26. Цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

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:

*1. Module → "Program" ID ";" Block "endProg"*

*2. Block → DescriptionList "begin" ListStmt "end"*

*3. DescriptionList → Description ";" | DescriptionList Description ";"*

*4. Description → Type ListId*

*5. ListId → ID | ListId "," ID*

*6. Type → SimpleType | ArrayType*

*7. SimpleType → ID*

*8. ArrayType → SimpleType "[" Integer "]"*

*9. ListStmt → Stmt | ListStmt ";" Stmt*

*10. Stmt → Assignment | Repeat | Condition*

*11. Assignment → Variable "=" Expression*

*12. Repeat → "repeat" ListStmt "until" Expression*

*13. Variable → ID | IndexVariable*

*14. IndexVariable → ID "[" SimpleExpression "]"*

*15. Expression → SimpleExpression | SimpleExpression Relation SimpleExpression*

*16. Relation → "<" | "<=" | ">" | ">=" | "==" | "!="*

*17. SimpleExpression → Term | SimpleExpression AddOp Term*

*18. AddOp → "+" | "–" | "||"*

*19. Term → Factor | Term MulOp Factor*

*20. MulOp → "\*" | "/" | "&&"*

*21. Factor → Const | Variable | "(" Expression ")" | "not" Factor*

*22. Const → NumericConst*

*23. NumericConst → Integer | Integer "." Integer | Integer "e" Integer | Integer "e" "+" Integer | Integer "e" "–" Integer*

*24. Integer → Digit | Integer Digit*

*25. Digit → "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"*

Данная грамматика является *SLR*(1)-грамматикой и ориентирована на восходящий разбор.

# 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), String. 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.

***Выражения типов***.

В языке *MyScala* предусмотрен конструктор struct(t), где t – таблица символов с информацией о полях этого типа записи;

## 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),

*Type* – тип идентификатора,

*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 (idents), 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*:**

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

function newTemp(type, val=null) {

let size = types[type].size

alignAddress(size)

return idents[0].push({

lex: '$' + ++tmpVarCnt,

type: type,

addr: nextAddr,

val: val

}) - 1

}

Процедура *alignAddress(size)* выравнивает адрес по заданному размеру.

function alignAddress(size) {

while (nextAddr % size != 0) nextAddr++

nextAddr += size

}

Функция *merge*(*arr1, arr2*) возвращает объединенный список.

function merge(arr1, arr2) {  
 if (arr1 === null || arr2 === null) return arr1 || arr2 || null  
 return [...arr1, ...arr2]  
}

Процедура *backPatch*(*indices*, *i*) устанавливает *i* в качестве целевой метки в каждую команду из списка *indices*.

function backPatch(indices, i) {

if (!indices) return

indices.forEach(idx => instrMem[idx] = instrMem[idx].replace('?', i))

}

Процедура *gen*(*instr*) инкрементно добавляет новую команду к последовательности сформированных к данному моменту команд (инкрементирует *nextInstr*).

function gen(instr) {

const addr = nextInstr++ + ''

instrMem.push(addr.padStart(3, '0') + ': ' + instr)

}

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

typeError(1) – повторное объявление идентификатора  
typeError(2) – идентификатор не является именем типа  
typeError(3) – идентификатор не является именем переменной  
typeError(4) – несовместимость типов  
typeError(5) – переменная должна быть типа Boolean

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

Разработанное СУО представлено в табл. 1.

Таблица .

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

|  |  |  |
| --- | --- | --- |
| Продукция с действиями | Семантические действия | Символ действия |
| *S* → *LstDecl* **begin** *LstStmt* **end** | - |  |
| *LstDecl* → *Decl* | – |  |
| *LstDecl* → *LstDecl1* *Decl* | – |  |
| *Decl* → **id** {A1} *LstId* **;** {A2} | idx = idents[tblSt.top()].findIndex(el => el.lex == **id**) if (idx != -1) typeError(1) idx = idents[tblSt.top()].push({ lex: **id** }) - 1 attrSt.push(idx) | A1 |
| type\_idx = typeSt.pop() idx = attrSt.pop()  startAddr = null tbl\_idx = null  // если тип – структура if (types[type\_idx].hasOwnProperty(tbl))    // создаем копию таблицы структуры  idents[tblSt.top()][idx].tbl = tbl\_idx  // если сейчас не описание структуры, то задаем адреса if (tblSt.top() == 0)  alignAddress(types[type\_idx].size)  idents[tblSt.top()][idx].addr  = startAddr || nextAddr  idents[tbl\_idx][idx].type = type\_idx offset += types[type\_idx].size | A2 |
| *Decl* → **class id** {A3} **{** *LstDecl* **}** {A4} | type\_idx = types.findIndex(el => el.lex == **id**) *if* (type\_idx != -1) typeError(1)  tbl\_idx = idents.push([]) - 1 type\_idx = types.push({ lex: **id**, tbl: tbl\_idx }) - 1  offsetSt.push(offset) offset = 0 tblSt.push(tbl\_idx) typesSt.push(type\_idx) | А3 |
| type\_idx = typesSt.pop() types[type\_idx].size = offset offset = offsetSt.pop() tblSt.pop() | А4 |
| *LstId* → **,** **id** {A5} *LstId1* {A6} | idx = idents[tblSt.top()].findIndex(el => el.lex == **id**) if (idx != -1) typeError(1) idx = idents[tblSt.top()].push({ lex: **id** }) - 1 attrSt.push(idx) | A1 |
| type\_idx = typeSt.top() idx = attrSt.pop()  startAddr = null tbl\_idx = null  // если тип – структура if (types[type\_idx].hasOwnProperty(tbl))    // создаем копию таблицы структуры  idents[tblSt.top()][idx].tbl = tbl\_idx  // если сейчас не описание структуры, то задаем адреса if (tblSt.top() == 0)  alignAddress(types[type\_idx].size)  idents[tblSt.top()][idx].addr  = startAddr || nextAddr - idents[tblSt.top()][idx].size  idents[tbl\_idx][idx].type = type\_idx offset += types[type\_idx].size | A6 |
| *LstId* → **:** **id** {A7} | type\_idx = types.findIndex(el => el.lex === **id**) if (type\_idx == -1) typeError(2) attrSt.push(type\_idx) | A7 |
| *LstStmt* → *Stmt* | – |  |
| *LstStmt* → *LstStmt1* *M* *Stmt*{A8} | Stmt\_nextlist = nextlistSt.pop() LstStmt1\_nextlist = nextlistSt.pop() M\_instr = instrSt.pop()  backPatch(LstStmt1\_nextlist, M\_instr) nextlistSt.push(Stmt.nextlist) | A8 |
| *Stmt* → *Var* **=** {A9} *Expr* **;** {A10} | attrSt.push(**=**.posStart) | A9 |
| { Expr\_tbl, Expr\_idx } = attrSt.pop() assign\_pos = attrSt.pop() { Var\_tbl, Var\_idx } = attrSt.pop()  Var = idents[Var.tbl\_idx][Var.idx] Expr = idents[Expr\_tbl][Expr\_idx]  *if*(Var.type != Expr.type) typeError(4, assign\_pos) gen(Var.addr '=' Expr.addr) | A10 |
| *Stmt* → **if** *Expr* **{** *N* *LstCase* **}** {A11} | LstCase\_nextlist = attrSt.pop() N\_nextlist = attrSt.pop()  backPatch(N\_nextlist, nextinstr)  { Expr\_tbl, Expr\_idx } = attrSt.pop()  Expr = idents[Expr\_tbl][Expr\_idx]  gen('if ' + Expr.lex + ' goto ' + Q.shift())  let sec\_label = Q.shift()  if (sec\_label == 'undefined')  gen('else goto ' + sec\_label)  else  gen('else goto ' + (nextInstr + 1))  backPatch(LstCase\_nextlist, nextinstr) nextlistSt.push([nextinstr]) | A11 |
| *Var* → **id** {A12} | idx = idents[0].findIndex(el => el.lex == **id**.lex) if (idx == -1) typeError(3) attrSt.push({0, idx }) | A12 |
| *Var* → *ClassVar* **.** **id** {A13}  x.y.z  ^ | idx = idents[tblSt.top()].findIndex(el => el.lex == **id**.lex) if (idx == -1) typeError(3) attrSt.push({ tblSt.top(), idx }) tblSt.pop() | A13 |
| *ClasstVar* → **id** {A14}  x.y.z ^ | idx = idents[0].findIndex(el => el.lex == **id**.lex) if (idx == -1) typeError(3)  tblSt.push(idents[0][idx].tbl) | A14 |
| *ClassVar* → *ClassVar1* **. id** {A15}  x.y.z  ^ | idx = idents[tblSt.top()].findIndex(el => el.lex == **id**.lex) if (idx == -1) typeError(3)  tblSt.push(idents[tblSt.top()][idx].tbl) tblSt.pop() | A15 |
| *ThenElse* → **then** **:** *M* *LstStmt*{A16} | M\_instr = instrSt.pop() num = attrSt.pop()  Q.push({ num, M\_instr })  nextlistSt.push(merge([nextinstr], null))  gen('goto ?') | A16 |
| *ThenElse* → *ThenElse 1* **else** **:** *M* *LstStmt*{A17} | M\_instr = instrSt.pop() LstCase1\_nextlist = nextlistSt.pop()  Q.push({ -1, M\_instr })  nextlistSt.push(merge(  merge(LstCase1\_nextlist, [nextinstr]), null))  gen('goto ?') | A17 |
| *Expr → SmpExpr* | – |  |
| *Expr → SmpExpr1* **rel**{A19} *SmpExpr2*{A20} | attrSt.push(**rel**) | A19 |
| { SmpExpr2\_tbl, SmpExpr2\_idx } = attrSt.pop() rel = attrSt.pop() { SmpExpr1\_tbl, SmpExpr1\_idx } = attrSt.pop()  SmpExpr2 = idents[SmpExpr2\_tbl][SmpExpr2\_idx] SmpExpr1 = idents[SmpExpr1\_tbl][SmpExpr1\_idx]  *if* (SmpExpr1.type ≠ SmpExpr2.type)  typeError(4, rel.posStart)  Expr = newTemp(SmpExpr1.type) attrSt.push({tbl: 0, idx: Expr})  gen('if' SmpExpr1.addr rel.name SmpExpr2.addr 'goto' nextInstr+3) gen(Expr.addr '= false') gen('goto' nextInstr+2) gen(Expr.addr '= true') | A20 |
| *SmpExpr → Term* | – |  |
| *SmpExpr → SmpExpr1* **add**{A21} *Term* {A22} | attrSt.push(**add**) | A21 |
| { Term\_tbl, Term\_idx } = attrSt.pop() additOp = attrSt.pop() { SmpExpr1\_tbl, SmpExpr1\_idx } = attrSt.pop()  Term = idents[Term\_tbl][Term\_idx] SmpExpr1 = idents[SmpExpr1\_tbl][ SmpExpr1\_idx]  *if* (SmpExpr1.type != Term.type) typeError(4, additOp)  SmpExpr\_idx = newTemp(SmpExpr1.type) SmpExpr = idents[0][SmpExpr\_idx]  gen(SmpExpr.lex '=' SmpExpr1.lex additOp.name Term.lex)  attrSt.push({ tbl: 0, idx: SmpExpr\_idx }) | A22 |
| *Term* → *Factor* | – |  |
| *Term* → *Term1* **mul** {A23} *Factor*{A24} | attrSt.push(**mul**) | A23 |
| { Factor\_tbl, Factor\_idx } = attrSt.pop() multOp = attrSt.pop() { Term1\_tbl, Term1\_idx } = attrSt.pop()  Term1 = idents[Term1\_tbl][Term1\_idx] Factor = idents[Factor\_tbl][Factor\_idx]  *if* (Term1.type ≠ Factor.type) typeError(4, multOp)  Term\_idx = newTemp(Term1.type) Term = idents[0][Term\_idx] gen(Term.lex '=' Term1.lex multOp.name Factor.lex)  attrSt.push({ tbl: 0, idx: Term\_idx }) | A24 |
| *Factor* → *Const* {A25} | { type, val } = attrSt.pop() let idx = newTemp(type, val) attrSt.push(0, idx) gen(idents[idx].addr ':' Const.addr ' = true') | A25 |
| *Factor* → *Var* | - |  |
| *Factor* → **(** *Expr* **)** | – |  |
| *Factor* → **!** {A26} *Factor1*{A27} | attrSt.push(**!**) | A26 |
| Factor1\_idx = attrSt.pop() not = attrSt.pop()  let Factor1 =  idents[Factor1\_idx.tbl\_idx][Factor1\_idx.idx] if (Factor1.type != Types.BOOL) typeError(5, not)  Factor\_idx = newTemp(Factor1.type, !Factor1.value) Factor = idents[0][Factor\_idx] gen(Factor.lex ' = not ' Factor1.lex)  attrSt.push({ 0, Factor\_idx }) | A27 |
| *Const* → **num** {A28} | if (**num**.attr == 0) type = Types.STR if (**num**.attr == 1) type = Types.FLOAT if (**num**.attr == 2) type = Types.FLOAT attrSt.push(({type: type, val: **num**.val}) | A28 |
| *Const* → **str** {A29} | attrSt.push(({type: Types.STR, val: **str**.val}) | A29 |
| *Сonst* → **true** {A30} | attrSt.push(({type: Types.BOOL, val: true}) | A30 |
| *Сonst* → **false** {A31} | attrSt.push({type: Types.BOOL, val: false}) | A31 |
| *M* → ε{A32} | instrSt.push(nextinstr) | A32 |
| *N* → ε{A33} | nextlistSt.push([nextinstr]) gen('goto ?') Q = [] | A33 |

# 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*).

Получена следующая СУТ:

S = LstDecl "begin" LstStmt "end".

LstDecl = Decl.

LstDecl = LstDecl Decl.

Decl = "id" A1 LstId ";" {A2}.

Decl = "class" "id" A3 "{" LstDecl "}" {A4}.

LstId = "," "id" A5 LstId {A6}.

LstId = ":" "id" {A7}.

LstStmt = Stmt.

LstStmt = LstStmt M Stmt {A8}.

Stmt = Var "=" A9 Expr ";" {A10}.

Stmt = "if" "(" Expr ")" "{" N ThenElse "}" {A11}.

Var = "id" {A12}.

Var = StructVar "." "id" {A13}.

StructVar = "id" {A14}.

StructVar = StructVar "." "id" {A15}.

ThenElse = "then" ":" M LstStmt {A16}.

ThenElse = ThenElse "else" ":" M LstStmt {A17}.

Expr = SmpExpr.

Expr = SmpExpr "rel" A19 SmpExpr {A20}.

SmpExpr = Term.

SmpExpr = SmpExpr "add" A21 Term {A22}.

Term = Factor.

Term = Term "mul" A23 Factor {A24}.

Factor = Const {A25}.

Factor = Var.

Factor = "(" Expr ")".

Factor = "!" A26 Factor {A27}.

Const = "num" {A28}.

Const = "str" {A29}.

Const = "true" {A30}.

Const = "false" {A31}.

M = {A32}.

N = {A33}.

A1 = {A1}.

A3 = {A3}.

A5 = {A5}.

A9 = {A9}.

A19 = {A19}.

A21 = {A21}.

A23 = {A23}.

A26 = {A26}.!

*SLR*(1)-метод требует вычисления функции

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

для каждого нетерминала *X* грамматики. Эта функция определяет множество терминалов, которые могут следовать непосредственно за нетерминалом *X* в какой-либо сентенциальной форме, выводимой из начального нетерминала *S*. Формальный процесс вычисления функции *Follow* подробно изложен в [4] при рассмотрении *LL*(1)-грамматик.

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

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

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

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

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

Для вычисления множеств *Follow*, построения *SLR*(1)-автомата (табл. 3) и соответствующей *SLR*(1)-таблицы разбора (табл. 4) использован генератор таблиц разбора (приложение TblGenerator.exe).

Таблица 3

*SLR*(1)-автомат

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № сост | Конфигурация | Символ перехода (свертки) | Состояние- преемник | Свертка {действие} |
| 1 | S = • LstDecl "begin" LstStmt "end" | LstDecl | 2 |  |
|  | LstDecl = • Decl | Decl | 3 |  |
|  | LstDecl = • LstDecl Decl | LstDecl | 2 |  |
|  | Decl = • "id" A1 LstId ";" {A2} | id | 4 |  |
|  | Decl = • "class" "id" A3 "{" LstDecl "}" {A4} | class | 5 |  |
| 2 | S = LstDecl • "begin" LstStmt "end" | begin | 6 |  |
|  | LstDecl = LstDecl • Decl | Decl | 7 |  |
|  | Decl = • "id" A1 LstId ";" {A2} | id | 4 |  |
|  | Decl = • "class" "id" A3 "{" LstDecl "}" {A4} | class | 5 |  |
| 3 | LstDecl = Decl • | begin, id, class, } |  | R2 |
| 4 | Decl = "id" • A1 LstId ";" {A2} | A1 | 8 |  |
|  | A1 = {A1} • | ,, : |  | R34 {A1} |
| 5 | Decl = "class" • "id" A3 "{" LstDecl "}" {A4} | id | 9 |  |
| 6 | S = LstDecl "begin" • LstStmt "end" | LstStmt | 10 |  |
|  | LstStmt = • Stmt | Stmt | 11 |  |
|  | LstStmt = • LstStmt M Stmt {A8} | LstStmt | 10 |  |
|  | Stmt = • Var "=" A9 Expr ";" {A10} | Var | 12 |  |
|  | Stmt = • "if" "(" Expr ")" "{" N ThenElse "}" {A11} | if | 13 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 7 | LstDecl = LstDecl Decl • | begin, id, class, } |  | R3 |
| 8 | Decl = "id" A1 • LstId ";" {A2} | LstId | 16 |  |
|  | LstId = • "," "id" A5 LstId {A6} | , | 17 |  |
|  | LstId = • ":" "id" {A7} | : | 18 |  |
| 9 | Decl = "class" "id" • A3 "{" LstDecl "}" {A4} | A3 | 19 |  |
|  | A3 = {A3} • | { |  | R35 {A3} |
| 10 | S = LstDecl "begin" LstStmt • "end" | end | stop |  |
|  | LstStmt = LstStmt • M Stmt {A8} | M | 20 |  |
|  | M = {A32} • | id, if |  | R32 {A32} |
| 11 | LstStmt = Stmt • | end, id, }, if, else |  | R8 |
| 12 | Stmt = Var • "=" A9 Expr ";" {A10} | = | 21 |  |
| 13 | Stmt = "if" • "(" Expr ")" "{" N ThenElse "}" {A11} | ( | 22 |  |
| 14 | Var = "id" {A12} • | ;, =, ), rel, add, mul |  | R12 {A12} |
|  | StructVar = "id" {A14} • | . |  | R14 {A14} |
| 15 | Var = StructVar • "." "id" {A13} | . | 23 |  |
|  | StructVar = StructVar • "." "id" {A15} | . | 23 |  |
| 16 | Decl = "id" A1 LstId • ";" {A2} | ; | 24 |  |
| 17 | LstId = "," • "id" A5 LstId {A6} | id | 25 |  |
| 18 | LstId = ":" • "id" {A7} | id | 26 |  |
| 19 | Decl = "class" "id" A3 • "{" LstDecl "}" {A4} | { | 27 |  |
| 20 | LstStmt = LstStmt M • Stmt {A8} | Stmt | 28 |  |
|  | Stmt = • Var "=" A9 Expr ";" {A10} | Var | 12 |  |
|  | Stmt = • "if" "(" Expr ")" "{" N ThenElse "}" {A11} | if | 13 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 21 | Stmt = Var "=" • A9 Expr ";" {A10} | A9 | 29 |  |
|  | A9 = {A9} • | id, (, !, num, str, true, false |  | R37 {A9} |
| 22 | Stmt = "if" "(" • Expr ")" "{" N ThenElse "}" {A11} | Expr | 30 |  |
|  | Expr = • SmpExpr | SmpExpr | 31 |  |
|  | Expr = • SmpExpr "rel" A19 SmpExpr {A20} | SmpExpr | 31 |  |
|  | SmpExpr = • Term | Term | 32 |  |
|  | SmpExpr = • SmpExpr "add" A21 Term {A22} | SmpExpr | 31 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "mul" A23 Factor {A24} | Term | 32 |  |
|  | Factor = • Const {A25} | Const | 34 |  |
|  | Factor = • Var | Var | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
|  | Factor = • "!" A26 Factor {A27} | ! | 37 |  |
|  | Const = • "num" {A28} | num | 38 |  |
|  | Const = • "str" {A29} | str | 39 |  |
|  | Const = • "true" {A30} | true | 40 |  |
|  | Const = • "false" {A31} | false | 41 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 23 | Var = StructVar "." • "id" {A13} | id | 42 |  |
|  | StructVar = StructVar "." • "id" {A15} | id | 42 |  |
| 24 | Decl = "id" A1 LstId ";" {A2} • | begin, id, class, } |  | R4 {A2} |
| 25 | LstId = "," "id" • A5 LstId {A6} | A5 | 43 |  |
|  | A5 = {A5} • | ,, : |  | R36 {A5} |
| 26 | LstId = ":" "id" {A7} • | ; |  | R7 {A7} |
| 27 | Decl = "class" "id" A3 "{" • LstDecl "}" {A4} | LstDecl | 44 |  |
|  | LstDecl = • Decl | Decl | 3 |  |
|  | LstDecl = • LstDecl Decl | LstDecl | 44 |  |
|  | Decl = • "id" A1 LstId ";" {A2} | id | 4 |  |
|  | Decl = • "class" "id" A3 "{" LstDecl "}" {A4} | class | 5 |  |
| 28 | LstStmt = LstStmt M Stmt {A8} • | end, id, }, if, else |  | R9 {A8} |
| 29 | Stmt = Var "=" A9 • Expr ";" {A10} | Expr | 45 |  |
|  | Expr = • SmpExpr | SmpExpr | 31 |  |
|  | Expr = • SmpExpr "rel" A19 SmpExpr {A20} | SmpExpr | 31 |  |
|  | SmpExpr = • Term | Term | 32 |  |
|  | SmpExpr = • SmpExpr "add" A21 Term {A22} | SmpExpr | 31 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "mul" A23 Factor {A24} | Term | 32 |  |
|  | Factor = • Const {A25} | Const | 34 |  |
|  | Factor = • Var | Var | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
|  | Factor = • "!" A26 Factor {A27} | ! | 37 |  |
|  | Const = • "num" {A28} | num | 38 |  |
|  | Const = • "str" {A29} | str | 39 |  |
|  | Const = • "true" {A30} | true | 40 |  |
|  | Const = • "false" {A31} | false | 41 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 30 | Stmt = "if" "(" Expr • ")" "{" N ThenElse "}" {A11} | ) | 46 |  |
| 31 | Expr = SmpExpr • | ;, ) |  | R18 |
|  | Expr = SmpExpr • "rel" A19 SmpExpr {A20} | rel | 47 |  |
|  | SmpExpr = SmpExpr • "add" A21 Term {A22} | add | 48 |  |
| 32 | SmpExpr = Term • | ;, ), rel, add |  | R20 |
|  | Term = Term • "mul" A23 Factor {A24} | mul | 49 |  |
| 33 | Term = Factor • | ;, ), rel, add, mul |  | R22 |
| 34 | Factor = Const {A25} • | ;, ), rel, add, mul |  | R24 {A25} |
| 35 | Factor = Var • | ;, ), rel, add, mul |  | R25 |
| 36 | Factor = "(" • Expr ")" | Expr | 50 |  |
|  | Expr = • SmpExpr | SmpExpr | 31 |  |
|  | Expr = • SmpExpr "rel" A19 SmpExpr {A20} | SmpExpr | 31 |  |
|  | SmpExpr = • Term | Term | 32 |  |
|  | SmpExpr = • SmpExpr "add" A21 Term {A22} | SmpExpr | 31 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "mul" A23 Factor {A24} | Term | 32 |  |
|  | Factor = • Const {A25} | Const | 34 |  |
|  | Factor = • Var | Var | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
|  | Factor = • "!" A26 Factor {A27} | ! | 37 |  |
|  | Const = • "num" {A28} | num | 38 |  |
|  | Const = • "str" {A29} | str | 39 |  |
|  | Const = • "true" {A30} | true | 40 |  |
|  | Const = • "false" {A31} | false | 41 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 37 | Factor = "!" • A26 Factor {A27} | A26 | 51 |  |
|  | A26 = {A26} • | id, (, !, num, str, true, false |  | R41 {A26} |
| 38 | Const = "num" {A28} • | ;, ), rel, add, mul |  | R28 {A28} |
| 39 | Const = "str" {A29} • | ;, ), rel, add, mul |  | R29 {A29} |
| 40 | Const = "true" {A30} • | ;, ), rel, add, mul |  | R30 {A30} |
| 41 | Const = "false" {A31} • | ;, ), rel, add, mul |  | R31 {A31} |
| 42 | Var = StructVar "." "id" {A13} • | ;, =, ), rel, add, mul |  | R13 {A13} |
|  | StructVar = StructVar "." "id" {A15} • | . |  | R15 {A15} |
| 43 | LstId = "," "id" A5 • LstId {A6} | LstId | 52 |  |
|  | LstId = • "," "id" A5 LstId {A6} | , | 17 |  |
|  | LstId = • ":" "id" {A7} | : | 18 |  |
| 44 | Decl = "class" "id" A3 "{" LstDecl • "}" {A4} | } | 53 |  |
|  | LstDecl = LstDecl • Decl | Decl | 7 |  |
|  | Decl = • "id" A1 LstId ";" {A2} | id | 4 |  |
|  | Decl = • "class" "id" A3 "{" LstDecl "}" {A4} | class | 5 |  |
| 45 | Stmt = Var "=" A9 Expr • ";" {A10} | ; | 54 |  |
| 46 | Stmt = "if" "(" Expr ")" • "{" N ThenElse "}" {A11} | { | 55 |  |
| 47 | Expr = SmpExpr "rel" • A19 SmpExpr {A20} | A19 | 56 |  |
|  | A19 = {A19} • | id, (, !, num, str, true, false |  | R38 {A19} |
| 48 | SmpExpr = SmpExpr "add" • A21 Term {A22} | A21 | 57 |  |
|  | A21 = {A21} • | id, (, !, num, str, true, false |  | R39 {A21} |
| 49 | Term = Term "mul" • A23 Factor {A24} | A23 | 58 |  |
|  | A23 = {A23} • | id, (, !, num, str, true, false |  | R40 {A23} |
| 50 | Factor = "(" Expr • ")" | ) | 59 |  |
| 51 | Factor = "!" A26 • Factor {A27} | Factor | 60 |  |
|  | Factor = • Const {A25} | Const | 34 |  |
|  | Factor = • Var | Var | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
|  | Factor = • "!" A26 Factor {A27} | ! | 37 |  |
|  | Const = • "num" {A28} | num | 38 |  |
|  | Const = • "str" {A29} | str | 39 |  |
|  | Const = • "true" {A30} | true | 40 |  |
|  | Const = • "false" {A31} | false | 41 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 52 | LstId = "," "id" A5 LstId {A6} • | ; |  | R6 {A6} |
| 53 | Decl = "class" "id" A3 "{" LstDecl "}" {A4} • | begin, id, class, } |  | R5 {A4} |
| 54 | Stmt = Var "=" A9 Expr ";" {A10} • | end, id, }, if, else |  | R10 {A10} |
| 55 | Stmt = "if" "(" Expr ")" "{" • N ThenElse "}" {A11} | N | 61 |  |
|  | N = {A33} • | then |  | R33 {A33} |
| 56 | Expr = SmpExpr "rel" A19 • SmpExpr {A20} | SmpExpr | 62 |  |
|  | SmpExpr = • Term | Term | 32 |  |
|  | SmpExpr = • SmpExpr "add" A21 Term {A22} | SmpExpr | 62 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "mul" A23 Factor {A24} | Term | 32 |  |
|  | Factor = • Const {A25} | Const | 34 |  |
|  | Factor = • Var | Var | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
|  | Factor = • "!" A26 Factor {A27} | ! | 37 |  |
|  | Const = • "num" {A28} | num | 38 |  |
|  | Const = • "str" {A29} | str | 39 |  |
|  | Const = • "true" {A30} | true | 40 |  |
|  | Const = • "false" {A31} | false | 41 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 57 | SmpExpr = SmpExpr "add" A21 • Term {A22} | Term | 63 |  |
|  | Term = • Factor | Factor | 33 |  |
|  | Term = • Term "mul" A23 Factor {A24} | Term | 63 |  |
|  | Factor = • Const {A25} | Const | 34 |  |
|  | Factor = • Var | Var | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
|  | Factor = • "!" A26 Factor {A27} | ! | 37 |  |
|  | Const = • "num" {A28} | num | 38 |  |
|  | Const = • "str" {A29} | str | 39 |  |
|  | Const = • "true" {A30} | true | 40 |  |
|  | Const = • "false" {A31} | false | 41 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 58 | Term = Term "mul" A23 • Factor {A24} | Factor | 64 |  |
|  | Factor = • Const {A25} | Const | 34 |  |
|  | Factor = • Var | Var | 35 |  |
|  | Factor = • "(" Expr ")" | ( | 36 |  |
|  | Factor = • "!" A26 Factor {A27} | ! | 37 |  |
|  | Const = • "num" {A28} | num | 38 |  |
|  | Const = • "str" {A29} | str | 39 |  |
|  | Const = • "true" {A30} | true | 40 |  |
|  | Const = • "false" {A31} | false | 41 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 59 | Factor = "(" Expr ")" • | ;, ), rel, add, mul |  | R26 |
| 60 | Factor = "!" A26 Factor {A27} • | ;, ), rel, add, mul |  | R27 {A27} |
| 61 | Stmt = "if" "(" Expr ")" "{" N • ThenElse "}" {A11} | ThenElse | 65 |  |
|  | ThenElse = • "then" ":" M LstStmt {A16} | then | 66 |  |
|  | ThenElse = • ThenElse "else" ":" M LstStmt {A17} | ThenElse | 65 |  |
| 62 | Expr = SmpExpr "rel" A19 SmpExpr {A20} • | ;, ) |  | R19 {A20} |
|  | SmpExpr = SmpExpr • "add" A21 Term {A22} | add | 48 |  |
| 63 | SmpExpr = SmpExpr "add" A21 Term {A22} • | ;, ), rel, add |  | R21 {A22} |
|  | Term = Term • "mul" A23 Factor {A24} | mul | 49 |  |
| 64 | Term = Term "mul" A23 Factor {A24} • | ;, ), rel, add, mul |  | R23 {A24} |
| 65 | Stmt = "if" "(" Expr ")" "{" N ThenElse • "}" {A11} | } | 67 |  |
|  | ThenElse = ThenElse • "else" ":" M LstStmt {A17} | else | 68 |  |
| 66 | ThenElse = "then" • ":" M LstStmt {A16} | : | 69 |  |
| 67 | Stmt = "if" "(" Expr ")" "{" N ThenElse "}" {A11} • | end, id, }, if, else |  | R11 {A11} |
| 68 | ThenElse = ThenElse "else" • ":" M LstStmt {A17} | : | 70 |  |
| 69 | ThenElse = "then" ":" • M LstStmt {A16} | M | 71 |  |
|  | M = {A32} • | id, if |  | R32 {A32} |
| 70 | ThenElse = ThenElse "else" ":" • M LstStmt {A17} | M | 72 |  |
|  | M = {A32} • | id, if |  | R32 {A32} |
| 71 | ThenElse = "then" ":" M • LstStmt {A16} | LstStmt | 73 |  |
|  | LstStmt = • Stmt | Stmt | 11 |  |
|  | LstStmt = • LstStmt M Stmt {A8} | LstStmt | 73 |  |
|  | Stmt = • Var "=" A9 Expr ";" {A10} | Var | 12 |  |
|  | Stmt = • "if" "(" Expr ")" "{" N ThenElse "}" {A11} | if | 13 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 72 | ThenElse = ThenElse "else" ":" M • LstStmt {A17} | LstStmt | 74 |  |
|  | LstStmt = • Stmt | Stmt | 11 |  |
|  | LstStmt = • LstStmt M Stmt {A8} | LstStmt | 74 |  |
|  | Stmt = • Var "=" A9 Expr ";" {A10} | Var | 12 |  |
|  | Stmt = • "if" "(" Expr ")" "{" N ThenElse "}" {A11} | if | 13 |  |
|  | Var = • "id" {A12} | id | 14 |  |
|  | Var = • StructVar "." "id" {A13} | StructVar | 15 |  |
|  | StructVar = • "id" {A14} | id | 14 |  |
|  | StructVar = • StructVar "." "id" {A15} | StructVar | 15 |  |
| 73 | ThenElse = "then" ":" M LstStmt {A16} • | }, else |  | R16 {A16} |
|  | LstStmt = LstStmt • M Stmt {A8} | M | 20 |  |
|  | M = {A32} • | id, if |  | R32 {A32} |
| 74 | ThenElse = ThenElse "else" ":" M LstStmt {A17} • | }, else |  | R17 {A17} |
|  | LstStmt = LstStmt • M Stmt {A8} | M | 20 |  |
|  | M = {A32} • | id, if |  | R32 {A32} |

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

Элемент таблицы e*lemTR* представляется следующими полями:

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

Алгоритм СУ-трансляции по таблице разбора (*parserSt* – стек состояний анализатора, *NomStlb*(*Token*.*Code*) – функция перекодировки кода токена в соответствующий номер столбца таблицы разбора):

fillTermsNonTerms()

idents = [[ /\* Глобальные переменные \*/ ]]

types = [

{ lex: 'Int', size: 4 },

{ lex: 'Float', size: 8 },

{ lex: 'Boolean', size: 1 },

{ lex: 'String', size: 64 }

]

offset = 0

nextAddr = 0

nextInstr = 0

instrMem = []

tmpVarCnt = 0

Q = []

tblSt = new Stack(0)

attrSt = new Stack

instrSt = new Stack

typesSt = new Stack

offsetSt = new Stack

nextlistSt = new Stack

let tokenCnt = 0

let token = tokenList[tokenCnt]

let prevToken = {...token}

let parserState = 1 // текущее состояние стека парсера

let parserSt = new Stack(1) // начальное состояние в стек

const TypesElemTR = new Enum(

'ERROR', // элемент ошибки

'SHIFT', // элемент сдвига

'REDUCE', // элемент свертки

'BREAK' // элемент останова

)

let elemTR = { type: TypesElemTR.ERROR } // элемент ошибки

if (token.code > TypesElemTR.ERROR) { // нет лексической ошибки

let lex = token.name

// пока не элемент останова

while (elemTR.type != TypesElemTR.BREAK) {

parserState = parserSt.top() // текущее состояние анализатора

let xmlNode = getXmlNode(parserState, lex)

elemTR = {

type: parseInt(xmlNode.getAttribute('ElType')),

par: parseInt(xmlNode.getAttribute('ElPar')),

left: xmlNode.getAttribute('Left'),

act: xmlNode.getAttribute('Act')

}

switch (elemTR.type) {

case TypesElemTR.ERROR: // синтаксическая ошибка

syntaxError(parserState, token)

return false

case TypesElemTR.SHIFT: // элемент сдвига

parserSt.push(elemTR.par) // состояние в стек

if (terms.has(lex)) {

prevToken = {...token}

token = tokenList[++tokenCnt]

if (token.code === 0) return false // ЛЕКСИЧЕСКАЯ ОШИБКА

}

lex = token.name

break

case TypesElemTR.REDUCE: // элемент свертки

for (let i = 1; i <= elemTR.par; i++)

parserSt.pop() // удаление верхних элементов стека

lex = elemTR.left // нетерминал левой части как новый входной символ

if (elemTR.act != '') { // есть действие

if (!semanticAction(elemTR.act, prevToken)) // выполнить действия

return false

}

break

}

}

gen('stop')

}

Переменная *prevToken* служит для сохранения предшествующего токена перед получением от сканера очередного входного токена.

Процедура *semanticAction*(*elemTR*.*Act*, *prevToken*) по коду действия (параметр *elemTR*.*act*) выбирает соответствующую подпрограмму, реализующую действие; для некоторых действий необходим токен (параметр *prevToken*).

# 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*:**

Реализация стека с помощью класса:

class Stack {

constructor(initialItem = null) {

this.items = []

if (initialItem !== null) this.push(initialItem)

}

push(item) { this.items.push(item) }

pop() {

if (this.isEmpty()) return null

return this.items.pop()

}

top() {

if (this.isEmpty()) return null

return this.items[this.items.length - 1]

}

isEmpty() { return this.items.length === 0 }

size() { return this.items.length }

clear() { this.items = [] }

}

***Процедуры и функции*:**

СУ-трансляция реализована в виде функции

**function** SUT, возвращающей **true**, если СУ-трансляция выполнена успешно, и **false** в противном случае.

Процедуры формирования текстов об ошибках:

**function** typeError (code, token);

**function** syntaxError (state, token);

В первых двух процедурах входной параметр *code* являются кодом ошибки, входной параметр *state* процедуры *syntaxError* 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** semanticAction (actionCode, token);

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;

**function** alignAddress(size);

**function** merge(arr1, arr2)

**function** backPatch(indices, i);

**function** gen(Op, Arg1, Arg2, Res: integer);

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

Семантические действия *A*1, …, *A*18 реализованы функциями, возвращающими **true**, если действие выполнено успешно (ошибки не обнаружены), и **false** в противном случае.

Для отображения результатов трансляции и распределения памяти реализованы подпрограммы:

***Результаты тестирования и отладки****.*

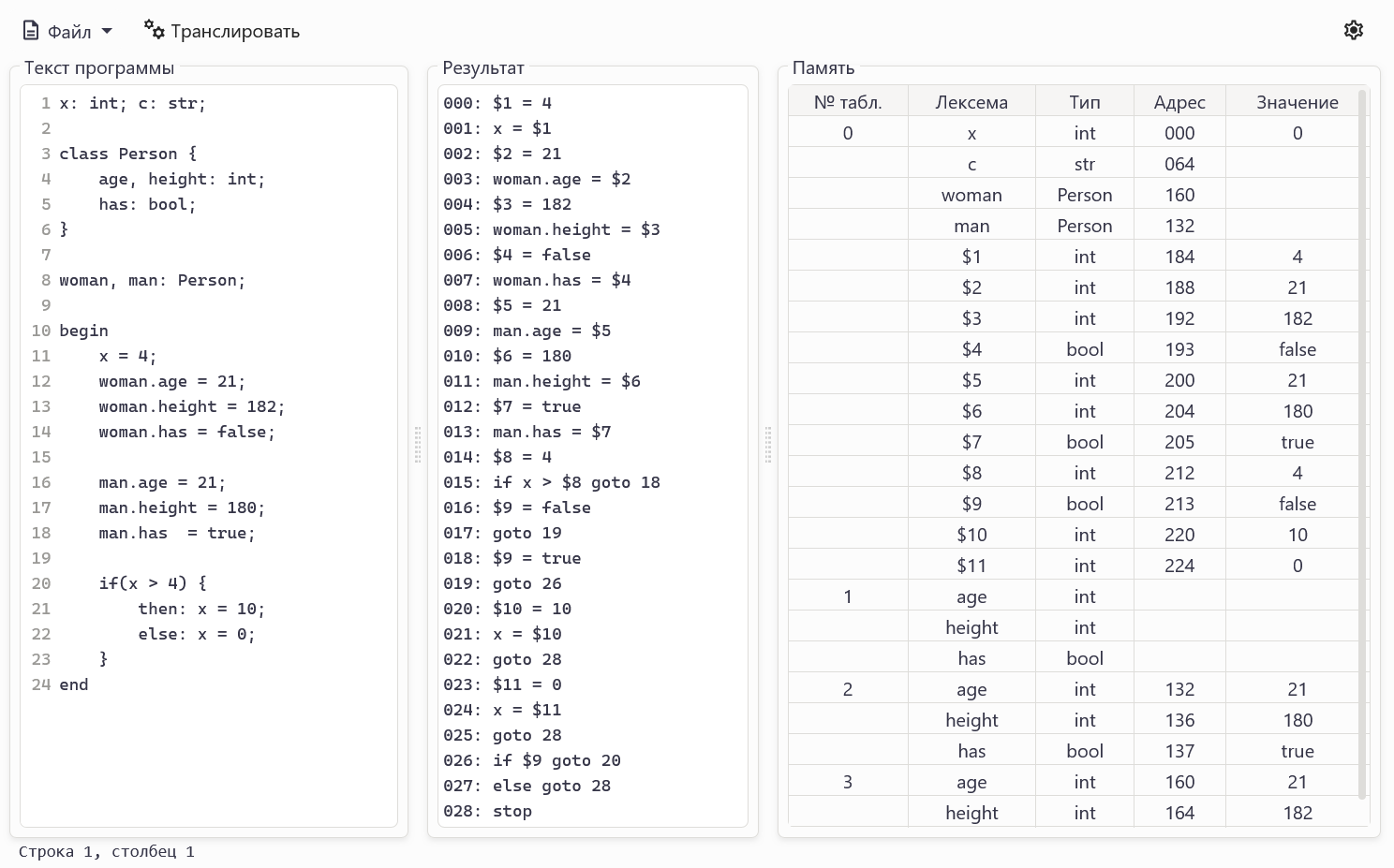


Рис. . Интерфейс приложения и пример успешной СУ-трансляции



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*. 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.