# Задача 1 « Построение конечного автомата»

## Теоретическая часть:

**Определение:** **Регуля́рные выраже́ния** (англ. *regular expressions*, сокр. **RegExp**, **RegEx**, жарг. ***регэ́кспы*** или ***ре́гексы***) — система синтаксического разбора текстовых фрагментов по формализованному шаблону, основанная на системе записи образцов для поиска.

**Определение:** **Регуля́рным мно́жеством** (или, **регуля́рным языком**) называется формальный язык, который удовлетворяет приведённым ниже свойствам.

|  |  |  |
| --- | --- | --- |
| **№** | **Свойство** | **Описание** |
| **1** | øЄR(Σ) | Пустое множество является регулярным множеством в алфавите Σ |
| **2** | {ε}ЄR(Σ) | Множество, состоящее из одной лишь пустой строки является регулярным множеством в алфавите Σ |
| **3** | Для любого aЄΣ:{a}ЄR(Σ) | Множество, состоящее из одного любого символа алфавита Σ является регулярным множеством в алфавите Σ |
| **4** | PЄR(Σ)˄Q ЄR(Σ)→(PUQ) ЄR(Σ) | Если два какие-либо множества являются регулярными в алфавите Σ, то и их объединение тоже является регулярным множеством в алфавите Σ |
| **5** | PЄR(Σ)˄Q ЄR(Σ)→PQ ЄR(Σ) | Если два какие-либо множества являются регулярными в алфавите Σ, то и множество, составленное из всевозможных сцеплений пар их элементов тоже является регулярным множеством в алфавите Σ |
| **6** | PЄR(Σ)→P\* ЄR(Σ) | Если какое-либо множество является регулярным в алфавите Σ, то множество всевозможных сцеплений его элементов тоже является регулярным множеством в алфавите Σ |

**Определение:** **Диаграмма переходов** – стилизованная блок-схема, изображает действия, выполняемые лексическим анализатором при вызове его синтаксическим анализатором для получения очередного токена.

Позиции в диаграмме переходов изображаются кружками и называются **состояниями**. Состояния соединены стрелками, называемыми **дугами**.

Одно из состояний имеет метку start – это **начальное состояние диаграммы** переходов в момент начала распознавания токена.

**Определение:** **Конечный автомат** — в теории алгоритмов математическая абстракция, позволяющая описывать пути изменения состояния объекта в зависимости от его текущего состояния и входных данных, при условии, что общее возможное количество состояний конечно. Конечный автомат является частным случаем абстрактного автомата.

**Определение:** **Детерминированным конечным автоматом** (ДКА) называется такой автомат, в котором для каждой последовательности входных символов существует лишь одно состояние, в которое автомат может перейти из текущего.

**Определение:** **Распознавателем** языка называется программа, которая получает на вход строку ***х*** и отвечает «да», если ***х*** – предложение языка, или в противном случае «нет».

**Определение:** **Недетерминированный автомат** может иметь более одного перехода из состояния при одном и том же входном символе.

Правила, которые **определяют регулярные выражения над алфавитом** ***E***.

1. ***Е*** представляет собой регулярное выражение, обозначающее ***{e}***, т е множество, содержащее пустую строку.
2. Если а является символом из ***Е***, то ***а*** – регулярное выражение, обозначающее ***{a}***, т е множество, содержащее строку ***а***. Хотя мы используем одну и ту же запись, технически регулярное выражение ***а*** отличается от строки ***а*** и символа ***а***. Говорим ли мы о регулярном выражении, строке или символе – становится понятно из контекста.
3. Предположим, что ***r*** и ***s*** – регулярные выражения, обозначающие языки ***L( r)*** и ***L(s)***.

Тогда

* ***( r)|(s)*** представляет собой регулярное выражение, обозначающее ***L( r)*** и ***L(s)***.
* ***( r)(s)*** – регулярное выражение , обозначающее ***L( r)L(s)***.
* ***( r)\**** - регулярное выражение, обозначающее ***(L( r))\****.
* ***( r)*** – регулярное выражение, обозначающее ***L( r)***.

Язык, задаваемый регулярным выражением, называется **регулярным множеством**.

Пусть ***Е={a,b}****.*

1. Регулярное выражение ***a|b*** обозначает множество ***{a,b}***.
2. Регулярное выражение ***(a|b)(a|b)*** – ***{aa, ab, ba,bb}***, множество всех строк из ***а*** и ***b*** длины 2. Другое регулярное выражение для того же множества – ***аа | ab | ba | bb***.
3. Регулярное выражение ***а\**** - множество всех строк из нуля или более а, т е ***{e, a, aa,aaa, …}***.
4. Регулярное выражение ***(a|b)\**** обозначает множество всех строк, содержащих нуль или несколько экземпляров ***а*** и ***b***, т е множество всех строк из ***а*** и ***b***. Другое регулярное выражение для этого множества – ***(а\*b\*)\****.
5. Регулярное выражение ***а| a\*b*** – множество содержащее строку ***а*** и все ее строки, состоящие из нуля или нескольких ***а***, за которыми следует ***b***.

**Алгебраические свойства регулярных выражений**

|  |  |
| --- | --- |
| **Аксиома** | **Описание** |
| r|s=s|r | Оператор | коммутативен |
| r|(s|t)=(r|s)|t | Оператор | ассоциативен |
| (rs)t=r(st) | Конкатенация ассоциативна |
| r(s|t)=rs|rt  (s|t)r=sr|tr | Конкатенация дистрибутивна над | |
| er=r  re=r | Е является единичным элементом по отношению к конкатенации |
| r\*=(r|e)\* | Связь между ***\**** и ***е*** |
| r\*\*=r\* | Оператор \* идемпотентен |

Недетерминированный конечный автомат (nondeterministic finite automaton, NFA) представляет собой математическую модель, состоящую из

* Множества состояний ***s***;
* Множества входных символов ***Е*** (символов входного алфавита);
* Функции переходов ***move***, которая отображает пары символ-состояние на множество состояний;
* Состояния ***s0***, известное как стартовое (начальное);
* Множества состояний ***F***, известных как допускающие (конечные);

Детерминированный конечный автомат (deterministic finite automaton, DFA) является специальным случаем недетерминированного конечного автомата, в котором

* Отсутствуют состояния, имеющие ***е***-переходы;
* Для каждого состояния ***s*** и входного символа ***а*** существует не более одной дуги, исходящей из ***s*** и помеченной как ***а***.

**Детерминированный конечный автомат** имеет для любого входного символа не более одного перехода из каждого состояния. Если для представления функции переходов ДКА используется таблица, то каждая запись в ней представляет собой единственное состояние. Следующий алгоритм имитирует поведение ДКА при обработке входной строки.

**Алгоритм «Моделирование ДКА»**

**Вход.** Входная строка ***х***, завершаемая символом конца файла ***eof***, и ДКА ***D*** со стартовым состоянием ***s0*** и множеством заключительных состояний ***F***.

**Выход.** «Да», если D допускает х, и «нет» в противном случае.

**Метод.** Ко входной строке х применяется алгоритм. Функция ***move(s,c)*** дает состояние, в которое происходит переход из состояния ***s*** при входном символе ***с***. Функция ***nextchar*** возвращает очередной символ строки ***х***.

*S:=s0;*

*C:= nextchar;*

*While c≠ eof do begin*

*S:= move(s,c);*

*C:= nextchar;*

*End;*

*If s € F then*

*Return “yes”*

*Else return “no”*

**Алгоритм. «Построение ДКА из НКА (построение подмножества)**

**Вход.** НКА ***N***.

**Выход.** ДКА ***D***, допускающий тот же язык.

**Метод.** Данный алгоритм строит таблицу переходов ***Dtran*** так, чтобы **D** «параллельно» все возможные перемещения ***N*** по данной входной строке.

Операции над состояниями НКА

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| e-closure(s)  (e-closure(s)) | Множество состояний НКА, достижимых из состояния ***s*** по е-переходам |
| e-closure(T)  (e-closure(T)) | Множество состояний НКА, достижимых из какого-либо состояния ***s*** из ***T*** только по ***е***-переходам |
| Move(T,a) | Множество состояний НКА, в которые имеется переход из какого-либо состояния ***s*** из ***T*** по входному символу ***а*** |

**Построение подмножества**

Изначально ***e-closure(s0)*** является единственным состоянием в ***Dstates*** и непомечено;

***While*** *в* ***Dstates*** *имеется непомеченное состояние* ***Т******do begin***

*Пометить* ***Т****;*

***For*** *каждый входной символ* ***а do begin***

***U:=e-closure(move(T,a)****);*

***If U not € Dstates then***

*Добавить* ***U*** *как непомеченное состояние в* ***Dstates****;*

***Dtran[T,a]:=U***

***End***

***End***

Множество состояний ***Dstates*** автомата ***D*** и таблицу его переходов ***Dtran*** можно создать следующим образом. Каждое состояние ***D*** соответствует множеству состояний НКА, в которых может находиться ***N*** после чтения некоторой последовательности входных символов, включая все возможные е-переходы до и после считанных символов. Стартовое состояние ***D*** – ***e-closure(s0)***. Состояния и переходы добавляются в ***D*** согласно алгоритму. Состояние ***D*** является допускающим, если оно представляет собой множество состояний НКА, содержащих как минимум одно допускающее состояние ***N***.

**Вычисление е-замыкания**

*Внести все состояния множества* ***Т*** *в стек* ***stack****;*

*Инициализировать* ***e-closure(T)*** *множеством* ***Т****;*

***While stack*** *не пуст* ***do begin***

*Снять со стека верхний элемент* ***t***

***For*** *каждое состояние* ***u*** *c дугой*

*От* ***t*** *к* ***u****, помеченной* ***е do***

***If u not € e-closure(T) then begin***

*Добавить* ***u*** *к* ***e-closure(T)****;*

*Поместить* ***u*** *в* ***stack***

***End***

***End***

**Алгоритм. «Минимизация количества состояний ДКА»**

**Вход.** ДКА ***М*** с множеством состояний ***S***: множеством входных символов ***Е***: переходами, определенными для всех состояний и входных символов: стартовым состоянием ***s0*** и множеством заключительных состояний ***F***.

**Выход.** ДКА ***M’***, допускающий тот же язык, что и ***М***, и имеющий наименьшее возможное количество состояний.

**Метод.**

1. Построить начальное разбиение ***П*** множества состояний с двумя группами; заключительные состояния ***F*** и незаключительные состояния ***S-F***.
2. Применить процедуру к разбиению ***П*** для построения нового разбиения ***Пн***.
3. Если ***Пnew=П***, то ***Пfinal=П***, и перейти к шагу ***(4)***. В противном случае повторить шаг ***(2)*** с ***П:=Пnew***
4. Выбрать одно состояние в каждой группе разбиения ***Пfinal*** в качестве представителя этой группы. Представители будут состояниями ДКА ***М’***. пусть s является представителем. Предположим, что для входного символа ***а*** в ***М*** существует переход из ***s*** в ***t***. Пусть ***r*** - представитель группы, в которой находится ***t*** (***r*** может являться ***t***). Тогда ***М’*** имеет переход из ***s*** в ***r*** по ***а***. Стартовым состоянием ***M’*** сделать представителя группы, содержащей стартовое состояние ***s0*** автомата ***М***, а заключительными состояниями ***М’*** – представителей в ***F***.
5. Если ***М’*** имеет мертвое состояние, т е состояние ***d***, которое не является заключительным и имеет переходы в себя для всех входных символов, удалим его из ***M’***. Удалим также все состояния не достижимые из стартового.

**Построение Пnew**

***For*** *каждая группа* ***G € П do begin***

*Разделить* ***G*** *на подгруппы, такие, что два состояния* ***s*** *и* ***t*** *из* ***G*** *находятся в одной и той же подгруппе тогда и только тогда, когда для всех входных символов а состояния* ***s*** *и* ***t*** *переходы по* ***а*** *в состояния из одной и той же группы* ***П***

*Заменить* ***G*** *в* ***Пnew*** *множеством всех созданных групп*

***End***

## Условие:

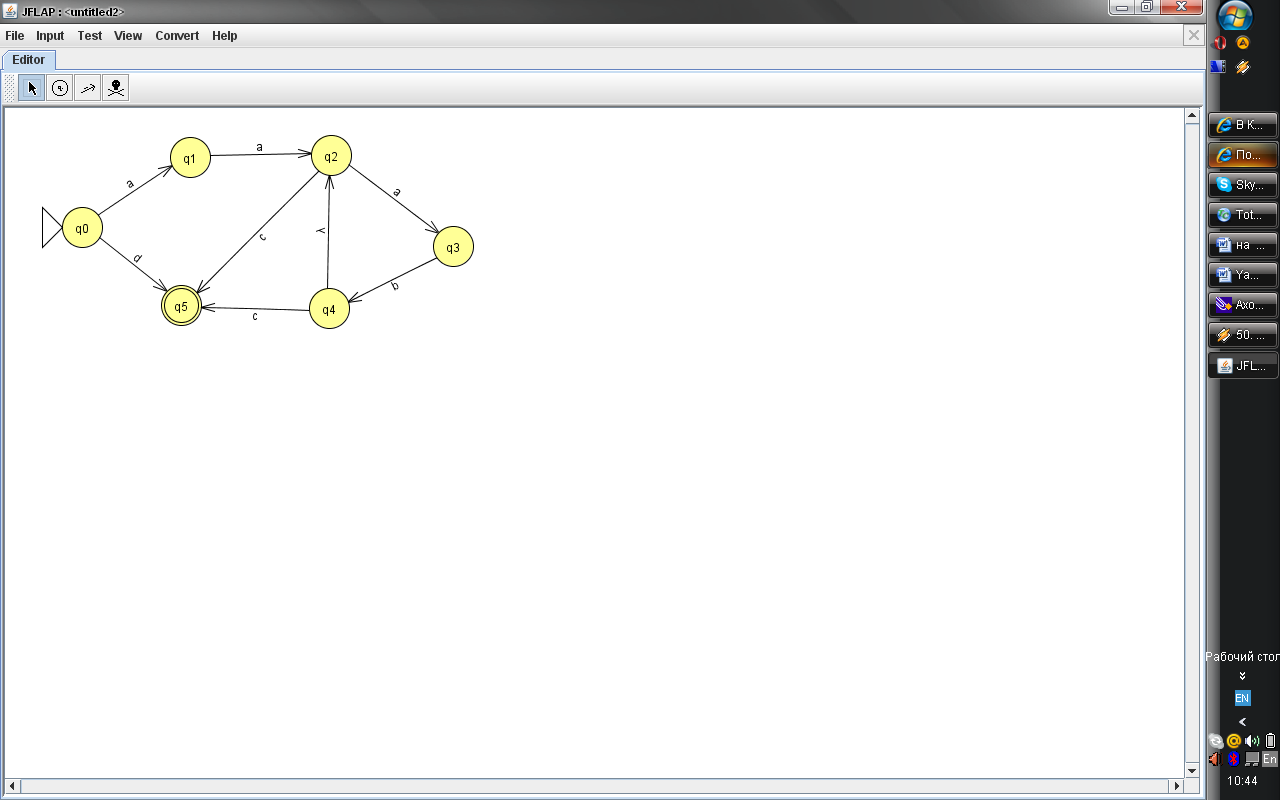
1. По регулярному выражению построить диаграмму переходов конечного автомата.
2. По построенной диаграмме построить таблицу состояний.
3. Проверить является ли построенный конечный автомат недетерминированным, записать объяснение.
4. Если конечный автомат является недетерминированным, то преобразовать его в детерминированный.
5. Проверить является ли построенный конечный автомат минимальным, записать объяснение.
6. Если конечный автомат не является минимальным, то минимизировать его.
7. Если производились преобразования построенного конечного автомата, то построить соответствующее ему регулярное выражение

## Варианты:

|  |  |
| --- | --- |
| **№** | **Формулировка варианта задания** |
|  | a2(ab)\*c|d |
|  | d|a2(ab)\*c |
|  | (d|a2)(ab)\*c |
|  | (d|a)2(ab)\*c |
|  | ((d|a)2ab)\*c |
|  | с+((d|a)2ab)\* |
|  | (с+)+((d|a)2ab)\* |
|  | a2(ab)\*(c|d)+ |
|  | (a|b|c)+abc(a|b|c)\* |
|  | (a|b|c)+abc4(a|b|c)\* |
|  | (d|a)\*(ab)\*c\* |
|  | (d|a)\*|(ab)\*c\* |
|  | (d|a)\*|(ab)\*|c\* |
|  | (d|a)\*(ab)\*|c\* |
|  | (d|a+)\*(ab)\*|c\* |
|  | (d|a\*)\*(ab)\*|c\* |
|  | a\*b\*c\*|a+b+c+ |
|  | a\*b\*(c\*|c+)b+a+ |
|  | a\*b\*(c\*|c+)\*b+a+ |
|  | a\*b\*(c\*|c+)cb+a+ |
|  | abc|cbaa+(bc|cb)a\* |
|  | ab(c\*|c)baa+(bc|cb)a\* |
|  | (abc|cb)a\*a+(bc|cb)a\* |
|  | a(bc|cba)a+(bc|cb)a\* |
|  | abc|cbaa+(bc|cb)a\* |
|  | (abc|cba)\*a+(bc|cb)a\* |
|  | abc|cbaa+(bc|cb)a\* |
|  | abc|cba(a+(bc|cb)a)\* |
|  | (abc|cbaa)+(bc|cb)a\* |
|  | (abc|cbaa+(bc|cb)a)\* |

## Пример построения конечного автомата:

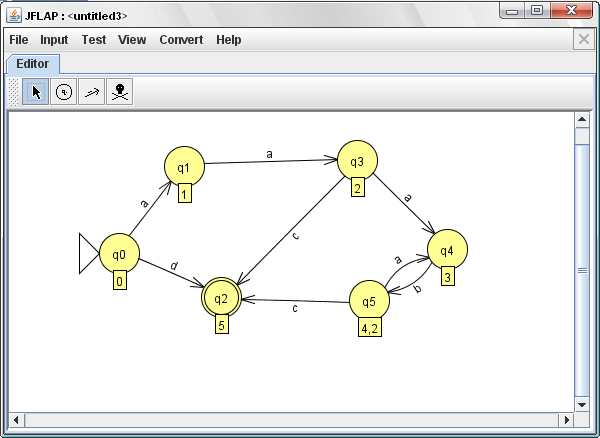
1. Формулировка задания: **d|a2(ab)\*c**
2. Диаграмма переходов конечного автомата:



Примечание:

|  |  |
| --- | --- |
|  | – обозначение начального состояния |
|  | – обозначение конечного состояния |
|  | *–* обозначение перехода изсостояния***q1***вcсостояние***q2***илипо***a***илипо***b*** |

1. Построенный автомат является недетерминированным, потому что есть ***е-переходы***. **Детерминируем** конечный автомат.

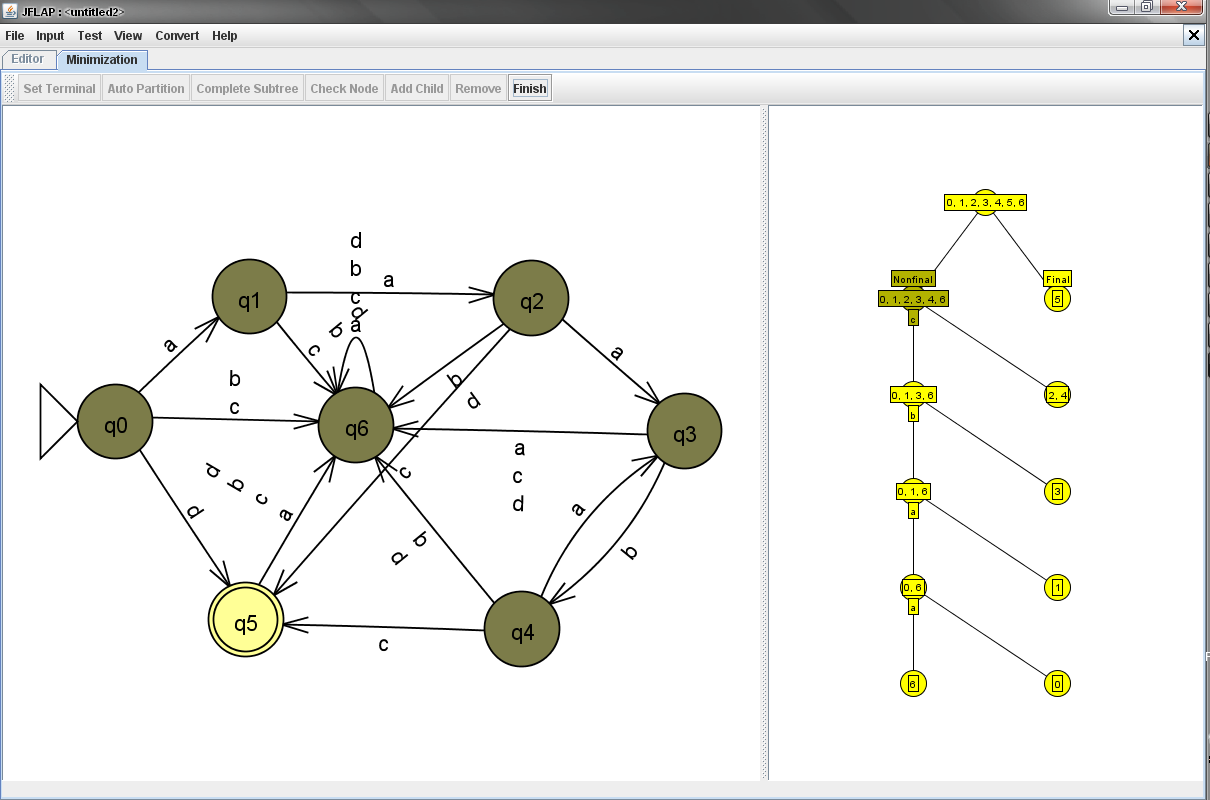


1. Для полученного конечного автомата построим **таблицу состояний**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** |
| **q0** | q1 |  |  | q2 |
| **q1** | q3 |  |  |  |
| **q2** |  |  |  |  |
| **q3** | q4 |  | q2 |  |
| **q4** |  | q5 |  |  |
| **q5** | q4 |  | q2 |  |

1. Конечный автомат является неминимизированным. **Минимизируем** .

Вводим дополнительное состояние***q6***

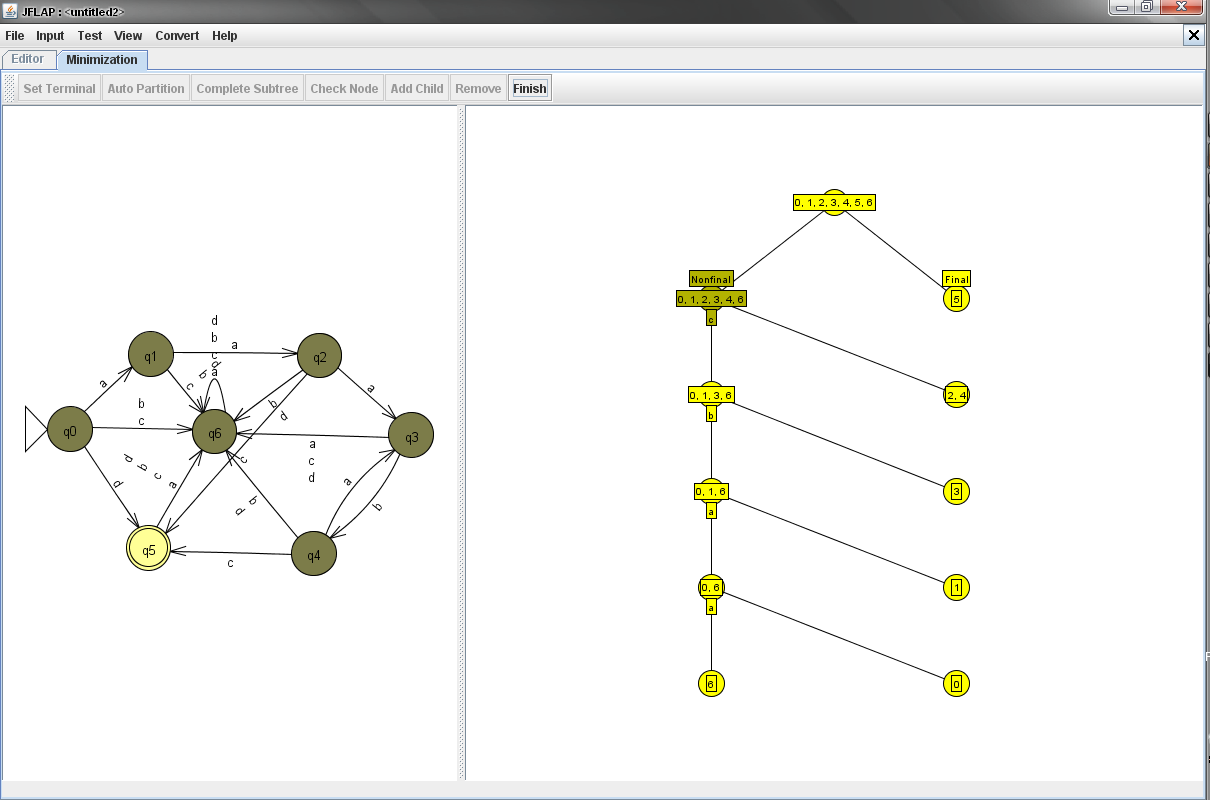


Строим дерево разбора*.*

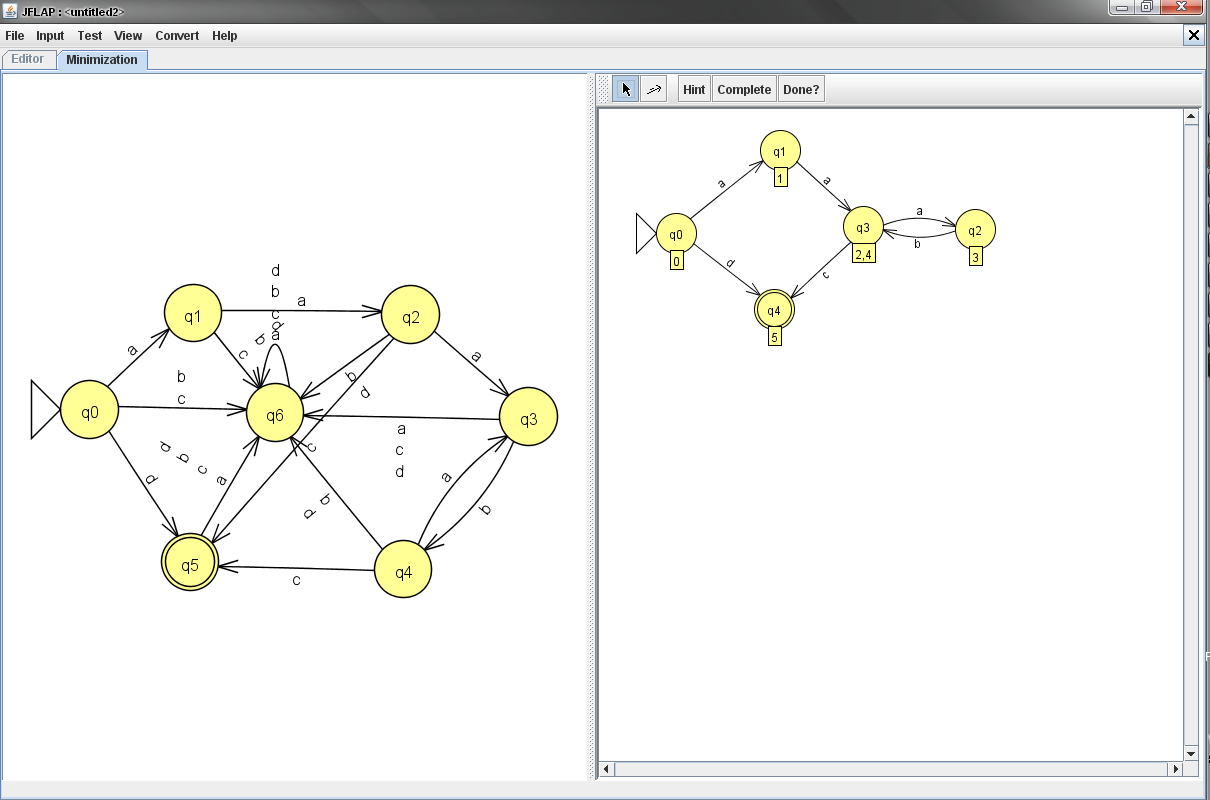
Алгоритм минимизации.

1. Пусть множество П(0,1,2,3,4,5,6) – множество всех состояний. Разобьем его на два подмножества согласно условию с состояниями (0,1,2,3,4,6) и (5).
2. Первое подмножество разбиваем еще на 2 с состояниями (0,1,3,6) и (2,4), потому что для всех входных символов переход по входному символу в состояние из одной и той же группы.
3. Первое подмножество делим на 2 с состояниями (0,1,6) и (3).
4. Перовое подмножество делим на 2 с состояниями (0,6) и (1).
5. Первое подмножество после предыдущего деления делим на 2 с состояниями (0) и (6).

В результате получили, что из состояний 2 и 4 можно оставить только одно, например 2. Количество состояний конечного автомата уменьшилось на одно.



Из полученного дерева видим, что состояния ***q2*** и***q4***можно объединить в одно состояние ***q3****.*



**Получили детерминированный и минимизированный конечный автомат.**

# Задача 2 «Построение КС-грамматики»

## Теоретическая часть

**Определение:** **Контекстно-свободная грамматика** состоит из 4 компонентов - терминалов, нетерминалов, стартового символа и продукций.

**Определение:** **Терминал (терминальный символ)** — объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов ASCII — латинские буквы, цифры и спец. символы.

**Определение:** **Нетерминал (нетерминальный символ)** — объект, обозначающий какую-либо *сущность* языка (например: формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

**Определение:** Один из нетерминалров грамматики считается **стартовымсимволом**, и множество строк, которые он обозначает, является языком, определяемым грамматикой.

**Определение:** **Продукции** грамматики определяют способ, которым терминалы и нетерминалы могут объединяться для создания строк. Каждая продукция состоит из нетерминала, за которым следует стрелка, и строка нетерминалов и терминалов.

**Определение*:*** Продукция называется **продукцией нетерминала**, если он записан в левой части.

**Определение*:*** **порождающая грамматика*****G*** *– это четверка* ***(VT, VN, P, S)***, где

***VT*** – алфавит терминальных символов (терминалов),

***VN*** – алфавит нетерминальных символов (нетермитналов), не пересекающийся с ***VT***,

***P*** – конечное подмножество множества ***(VTﮟVN)+ \*(VT∩VN)\****; элемент ***(α, β)*** множества ***Р*** называется правилом вывода и записывается в виде ***α → β***,

***S*** – начальный символ (цель) грамматики, ***S € VN***.

**Определение*:*** дерево называется **деревом вывода** (или **деревом разбора**) в КС-грамматике ***G = (VT, VN, P, S),***  если выполнены следующие условия:

* 1. каждая вершина помечена символом из множества ***(VTﮟVNﮟε),*** при этом корень дерева помечен символом ***S***; листья – символами из ***(VTﮟε);***
  2. если вершина дерева помечена символом ***А € VN***, а ее непосредственные потомки – символами ***а1***, ***а2***, ..., ***а***n, где каждое ***ai*** ***€ (VTﮟVN)***, то ***А→а1а2...аn*** – правило вывода в этой грамматике;
  3. если вершина дерева помечена символом ***А € VN***, а ее единственный непосредственный потомок помечен символом ***ε*** , то А ***→ ε*** – правило вывода в этой грамматике.

**Определение*:*** **КС-грамматика** называется **неоднозначной**, если существует хотя бы одна цепочка ***α € L(G)***, для которой может быть построено два или более различных деревьев вывода.

**Определение:** **Левая факторизация** представляет собой преобразование грамматики в пригодную для предикативного анализа.

**Определение:** **Грамматика** является **леворекурсивной**, если в ней имеется нетерминал ***А***, такой, что существует порождение ***А =>Aα*** для некоторой строки ***α***.

**Соглашения по обозначениям**

1. Эти символы являются терминалами:

**А)** строчные буквы из начала алфавита, такие как a,b,c;

**Б)** символы операторов, такие как +, -, и т.п.;

**В)** символы пунктуации, такие как запятые, скобки и т.п.;

**Г)** цифры 0, 1, 2, …, 9;

**Д)** строки, выделенные полужирным шрифтом, такие как **id** или **if**.

1. Эти символы являются нетерминалами:

**А)** прописные буквы из начала алфавита, такие как A, B, C;

**Б)** буква S, которая обычно обозначает стартовый символ;

**В)** имена из строчных букв, выделенные курсивом, такие как *stmt* или *expr*.

1. Прописные буквы из конца алфавита, такие как X, Y, Z, представляют грамматические символы, т е либо терминалы, либо нетерминалы.
2. Строчные буквы из конца алфавита, такие как u, v, …, z, обозначают строки терминалов.
3. Строчные греческие буквы, такие как α, β, µ, представляют строки грамматических символов. Таким образом, в общем виде продукция может быть записана как А → α, в которой одиночный нетерминал А располагается справа от стрелки, а строка грамматических символов α – справа от стрелки.
4. Если А → α1, A → α2, …, A → αk представляют собой продукции с А в левой части, то можем записать А → α1| α2| …| αk. Мы называем α1, α2, …, αk альтернативами А.
5. Если иное не указано явно, левая часть первой продукции является стартовым символом.
6. Продукция - →.

**Классификация грамматик и я зыков по Хомскому**

**(грамматики классифицируются по виду их правил вывода)**

**Тип 0:**

Грамматика ***G = (VT, VN, P, S)***, называется **грамматикой типа 0**, если на правила вывода не накладывается никаких ограничений (кроме тех, которые указаны в определении грамматики).

**Тип 1:**

Грамматика ***G = (VT, VN, P, S)***, называется **неукорачивающей** грамматикой, если каждое правило Р имеет вид ***α → β***, где ***α € (VTﮟVN)+***, ***β €*** ***(VTﮟVN)+ и | α| <= | β|***.

Грамматика ***G = (VT, VN, P, S)***, называется **контекстно-зависимой *(КЗ)***, если каждое правило из ***Р*** имеет вид ***α → β***, где ***α = ξ1А ξ2***; ***β = ξ1γ ξ2***; ***А*** € ***VN***; ***γ € (VTﮟVN)+***; ***ξ1, ξ2 € (VTﮟVN)\****.

**Тип 2:**

Грамматика ***G = (VT, VN, P, S)***, называется **контекстно-свободной *(КС)***, если каждое правило из Р имеет вид ***А → β***, где ***А € VN, β*** ***€ (VTﮟVN)+***.

Грамматика ***G = (VT, VN, P, S)***, называется **укорачивающей** **контекстно-свободной (УКС)**, если каждое правило из ***Р*** имеет вид ***А → β***, где ***А € VN, β € (VTﮟVN)\****.

**Тип 3:**

Грамматика ***G = (VT, VN, P, S)***, называется **праволинейной**, если каждое правило из ***Р*** имеет вид ***А → tB*** либо ***А →t***, где ***А € VN***, ***В € VN***, ***t €*** ***VN***.

Грамматика ***G = (VT, VN, P, S)***, называется **леволинейной**, если каждое правило из ***Р*** имеет вид ***А→ Bt*** либо ***А →t***, где ***А € VN, В € VN, t € VN***.

**Алгоритм «Устранение левой рекурсии»**

**Вход.** Грамматика ***G*** без циклов и ***ε***-продукций.

**Выход.** Эквивалентная грамматика без левой рекурсии.

**Метод.** Применить алгоритм устранения левой рекурсии. Результирующая грамматика без левой рекурсии может иметь ***ε***-продукции.

1. *Расположить нетерминалы в некотором порядке* ***А1, А2, ..., Аn***
2. ***for i:=1 to n do begin***

***for j:=1 to i-1 do begin***

*Заменить каждую продукцию вида* ***Ai → Ajγ***

*Продукциями* ***А →δ1γ| δ2γ| …| δkγ***

*Где* ***Аj А →δ1| δ2| …| δk*** *– все текущие* ***Аj*** *– продукции*

***End***

*Устранить непосредственную левую рекурсию*

*Среди* ***Аi*** *– продукций*

***End***

**Алгоритм «Левая факторизация грамматики»**

**Вход.** Грамматика ***G***.

**Выход.** Эквивалентная левофакторизованная грамматика

**Метод.** Для каждого нетерминала ***А*** находим наидлиннейший префикс ***α***, общий для двух или большего числа альтернатив. Если ***α≠ε***, т е имеется нетривиальный общий префикс, заменим все продукции ***А→αβ1| αβ2| …| αβn| γ***, где ***γ*** представляет все альтернативы, не начинающиеся с ***α***, продукциями

***А →αА’| γ***

***A’→ β1| β2| …| βn***

Здесь ***А’*** – новый нетерминал. Выполняем это преобразование до тех пор, пока никакие две альтернативы не будут иметь общий префикс.

## Условие:

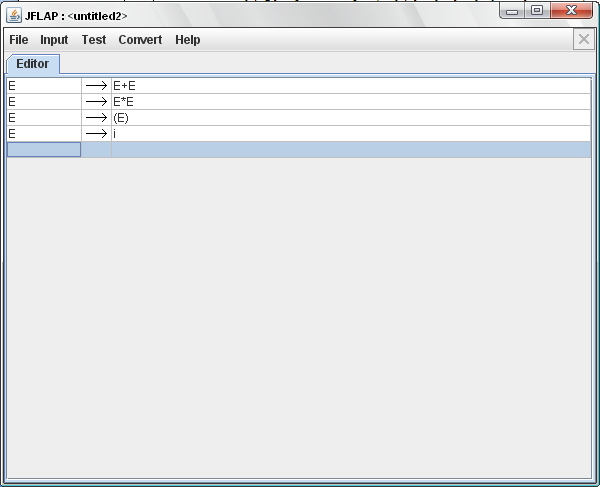
1. По описанию языка построить порождающую грамматику.
2. Определить тип построенной грамматики и свойства.
3. Построить для трех примеров деревья разбора.
4. Если грамматика является леворекурсивной, то устранить её.
5. Если грамматика является не левофакторизованной, то левофакторизовать.
6. Для модифицированной грамматики построить деревья разбора по тем же примерам что и в п.3

## Варианты:

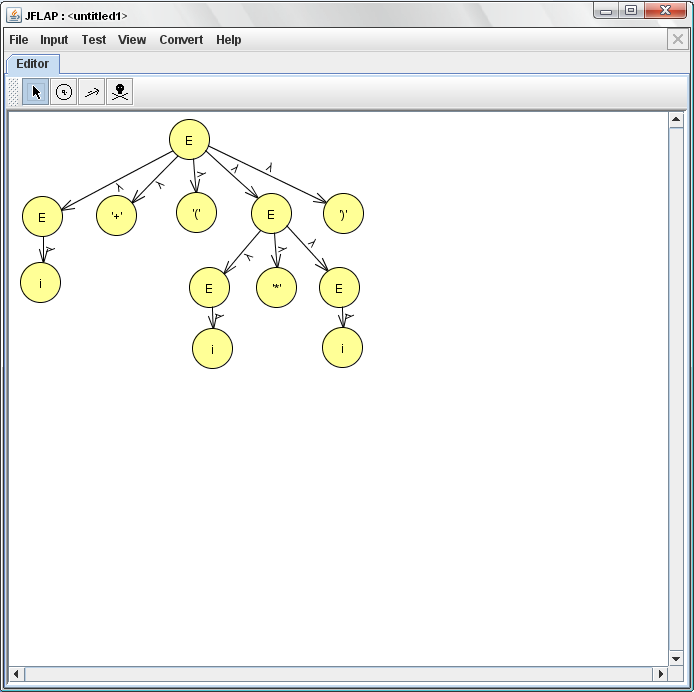
|  |  |
| --- | --- |
| № | Формулировка варианта задания |
|  | <E> ::= <E> ‘+’ <E> | <E> ‘\*’ <E> | ‘(‘ <E> ‘)’ | ‘i’ |
|  | <S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]  <E> ::= ‘i’ | ‘i’ ‘==’ <E>  <O> ::= ‘o’ … |
|  | 1  type:  2  formalParameter:  3  block:  4 |
|  | 1 |
|  | <P> ::= [ <H> ] <B>  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= <H> [ <B> ]  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= [ <H> ] [ <B> ]  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= <H> (<B>)…  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= [ <H> ] <B>  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… [‘;’] |
|  | <P> ::= [ <H> ] <B>  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… [‘;’] |
|  | <P> ::= <H> [ <B> ]  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= ‘b’ ( [ ‘,’ ] ‘b’)… ‘;’ |
|  | <P> ::= <H>… [ <B> ]  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= ‘b’ ( [ ‘,’ ] ‘b’)… [ ‘;’ ] |
|  | <P> ::= <H> [ <B>… ]  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= [‘b’] ( [‘,’] ‘b’)… ‘;’ |
|  | <P> ::= ([ <H> ] [ <B> ])…  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= (<H> | <B>)…  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= <H> [ <B> ]  <H> ::= ‘h’ (‘t’ ‘i’)… | ε  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <E> ::= <E> ‘-’ <E> | <E> ‘+’ <E> | ‘(‘ <E> ‘)’ | ‘i’ |
|  | <S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | 1 |
|  | h:1  2  b:  3 |
|  | 1  h:  2  b:  3 |
|  | 11  h:  2  b:  33 |
|  | 11  h:  22  b:  333 |
|  | <S> ::= ‘if’ [ <E> ] ( [ ‘i’ ‘:’ ] ‘then’ <O> )…  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ‘then’ <O> | <O>  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ] | <O>  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ]  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <E> ::= <E> ‘\*’ <E> | <E> ‘=’ <E> | ‘(‘ <E> ‘)’ | ‘i’ |

## Пример построения КС-грамматики:

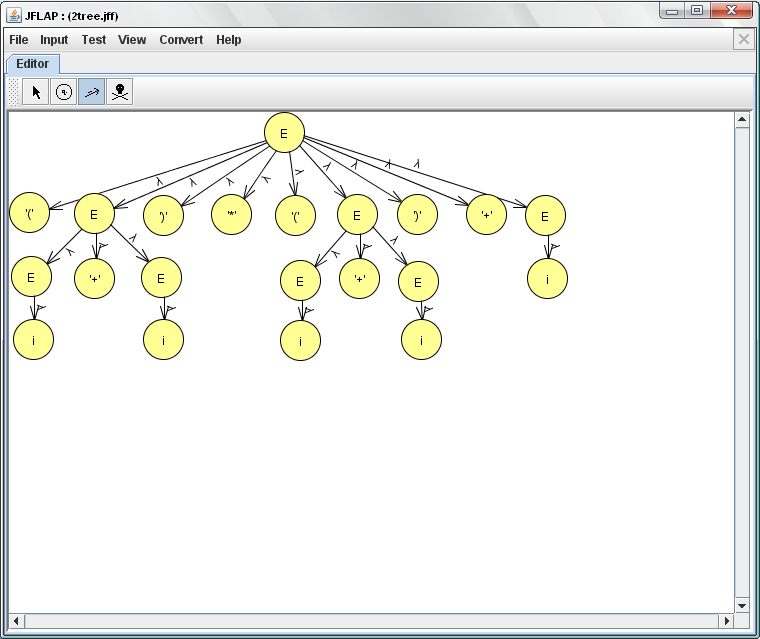
1. Дано описание языка: ***<E> ::= <E> ‘+’ <E> | <E> ‘\*’ <E> | ‘(‘ <E> ‘)’ | ‘i’***
2. Построим грамматику о описанию языка:



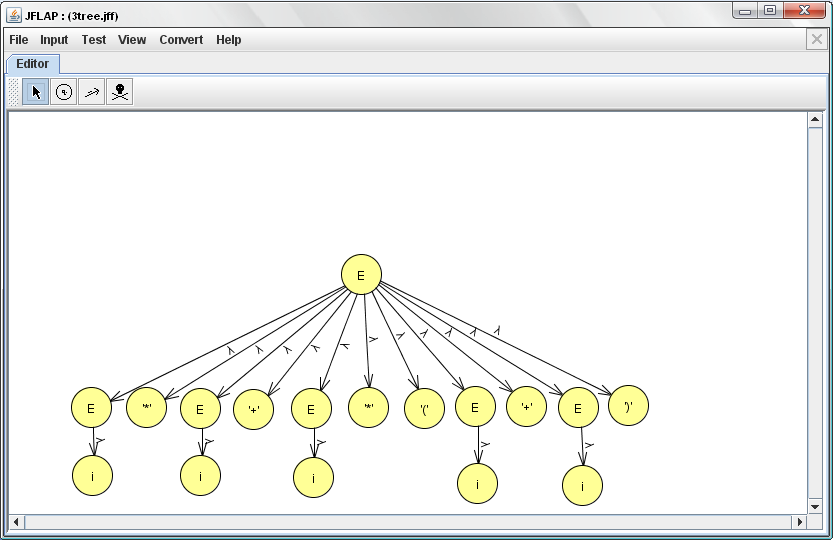
1. Грамматика является *контекстно-свободной (КС)*, потому что символы левой части - нетерминалы, а символы правой части – терминалы или нетерминалы.
2. Построим для трех примеров деревья разбора
3. ***i+ (i+i)***



1. ***(i+i)\*(i+i)+i***



1. ***i\*i+i\*(i+i)***



1. Грамматика является леворекурсивной. Преобразуем грамматику.

***α1=+Е β1=(Е)***

***α2=\*Е β2=i***

***Е = β1E’/ β2E’***

***E’= α1E’/ α2E’/ε***

***E→ (E)E’***

***E → iE’***

***E’ → +EE’***

***E’ → \*EE’***

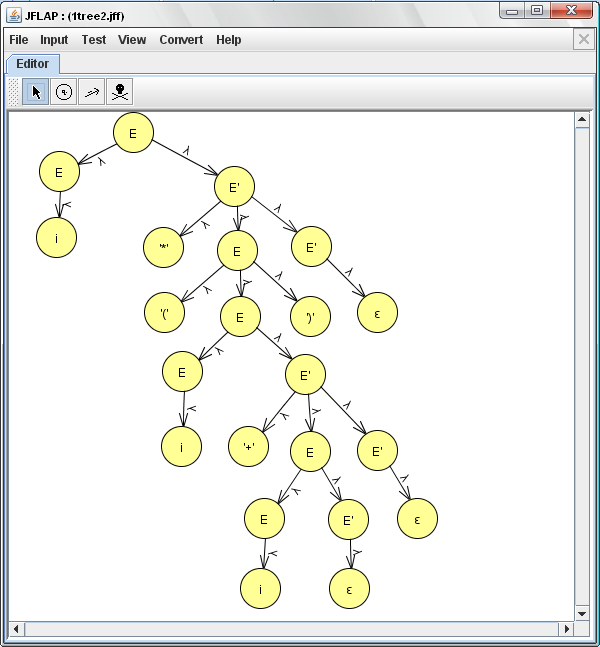
***E’ → ε***

1. Грамматика является левофакторизованной, потому что не встречается правил вида:

***А →аС***

***А → аD***

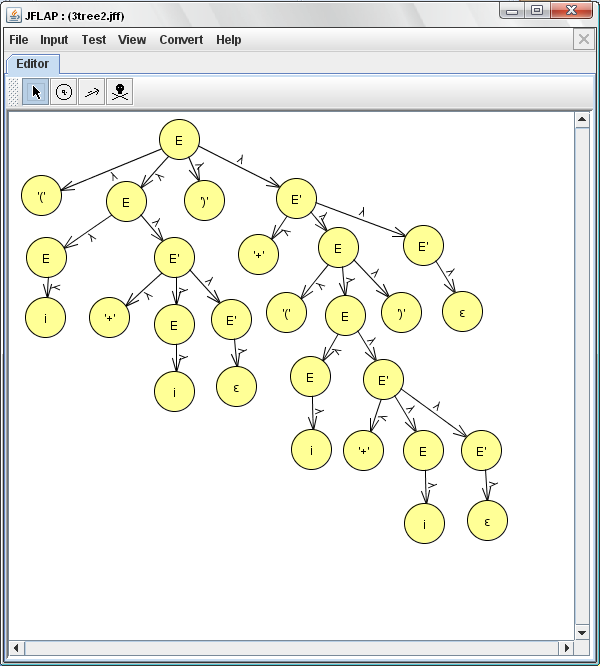
1. По преобразованной грамматике построим деревья разбора
2. ***i\*(i+i)***



1. ***i\*i+i***



1. ***(i+i)\*(i+i)***



# Задача 3 «Построение таблицы предиктивного анализатора»

## Теоретическая часть

**Определение*:*** **Предиктивный анализатор**представляет собой программу, содержащую процедуры для каждого нетерминального символа.

**Определение*:*** **Анализ методом рекурсивного спуска** (recursive-descent parsing) представляет собой способ нисходящего синтаксического анализа, при котором выполняется рад рекурсивных процедур для обработки входного потока (процедура связана с каждым нетерминальным символом грамматики).

**Определение*:*** **Предиктивный анализ** – анализ, при котором сканируемый символ однозначно определяет процедуру, выбранную для каждого нетерминала.

При построении предиктивного синтаксического анализатора можно создать его план в виде ***диаграммы переходов***. Для построения диаграммы переходов предиктивного синтаксического анализатора на основе грамматики вначале следует устранить из нее левые рекурсии, а затем провести левую факторизацию. После этого для каждого нетерминала ***А*** выполняется следующее.

1. Создаем начальное и заключительное состояния.
2. Для каждой продукции ***А →Х1Х2…Хn*** создаем путь от начального к заключительному состоянию с дугами, помеченными как ***Х1, Х2, …, Хn***.

**FIRST и FOLLOW**

Если ***α*** – произвольная строка символов грамматики, то определим ***FIRST(α)*** как множество терминалов, с которых начинаются строки, выводимые из ***α***. Если ***α → ε***, то ***ε € FIRST(α)***.

Определим ***FOLLOW(A)*** для нетерминала ***А*** как множество терминалов, которые могут располагаться непосредственно справа от ***А*** в некоторой сентенциальной форме, т е множество терминалов ***а***, таких, что существует порождения вида ***S=>αAaβ*** для некоторых ***α*** и ***β***.

Чтобы вычислить ***FIRST(X)*** для всех символов грамматики ***Х***, будем применять следующее правило до тех пор, пока ни к одному из множеств ***FIRST*** не смогут быть добавлены ни терминалы, ни ***ε***.

1. Если ***Х*** – терминал, то ***FIRST(X) = {X}***.
2. Если имеется продукция ***Х → ε***, добавим ***ε*** к ***FIRST(X)***.
3. Если ***Х*** – нетерминал и имеется продукция ***Х → Y1Y2…Yk***, то поместим ***а*** в ***FIRST(X)***, если для некоторого ***i a € FIRST(Yi)*** и ***ε*** входит во все множества ***FIRST(Y1), …, FIRST(Yi-1)***, т е ***Y1 … Yi-1 =>ε***. Если ε имеется во всех ***FIRST(Yi), i=1..k***, то добавляем ***ε*** к ***FIRST(X)***.

Теперь можно вычислить ***FIRST*** для любой строки ***Х1Х2… Хn*** следующим образом. Добавим к ***FIRST(Х1Х2… Хn)*** все ***не-ε*** символы из ***FIRST(Х1)***. Добавим также все ***не-ε*** символы из ***FIRST(Х2)***, если ***ε €*** ***FIRST(Х1)***, все ***не-ε*** символы из ***FIRST(Х3)***, если ε имеется как в ***FIRST(Х1)***, так и в ***FIRST(Х2)*** и т д. Добавим ***FIRST(Х1Х2… Хn)***, если для всех ***i FIRST(Хi)*** содержит ***ε***.

Чтобы **вычислить FOLLOW(A)** для всех нетерминалов ***А***, будем применять следующие правила до тех пор, пока ни к одному множеству ***FOLLOW*** нельзя будет добавить ни одного символа.

1. Поместим ***$*** в ***FOLLOW(S)***, где ***S*** – стартовый символ, а ***$*** - правый ограничитель входного потока.
2. Если имеется продукция ***А*** ***→ αВβ***, то все элементы множества ***FIRST(β)***, кроме ε помещаются в множество ***FOLLOW(В)***.
3. Если имеется продукция ***А → αВ*** или ***А →αВβ,*** где ***FIRST(β)*** содержит ***ε***, то все элементы из множества ***FOLLOW(А)*** помещаются в множество ***FOLLOW(В)***.

Для **построения таблицы предиктивного анализа** данной грамматики ***G*** может использоваться приведенный далее алгоритм. Идея алгоритма в следующем. Предположим, что ***А → α*** представляет собой продукцию, у которой ***а € FIRST(α)***. Тогда синтаксический анализатор заменит ***А*** строкой ***α*** при текущем входном символе ***а***. Единственная сложность возникает при ***α=ε*** или ***α=>ε***. В этом случае мы снова должны заменить ***А*** на ***а***, если текущий входной символ имеется в ***FOLLOW(А)*** или из входного потока получен ***$***, который входит в ***FOLLOW(А)***.

**Алгоритм «Построение таблицы предиктивного анализатора»**

**Вход.** Грамматика ***G***.

**Выход.** Таблица анализа ***М***.

**Метод.**

1. Для каждой продукции грамматики ***А → α*** выполняем шаги ***2*** и ***3***.
2. Для каждого терминала ***а*** из ***FIRST(α)*** добавляем ***А → α*** в ячейку ***М[A,a]***.
3. Если в ***FIRST(α)*** входит ***ε***, для каждого терминала ***b*** из ***FOLLOW(А)*** добавим ***А → α*** в ячейку ***М[A,b]***. Если ***ε*** входит в ***FIRST(α)***, а ***$*** - в ***FOLLOW(А)***, добавим ***А → α*** в ячейку ***М[A,$]***.
4. Сделаем каждую неопределенную ячейку таблицы ***М*** указывающей на ошибку.

## Восстановление после ошибок

**Правила восстановления после ошибок в «режиме паники»:**

1. В качестве первого приближения можем поместить в синхронизирующее множество для нетерминала ***А*** все символы из множества ***FOLLOW(A)***. Если пропустим все токены до элемента из ***FOLLOW(A)*** и снимем ***А*** со стека, вероятно, мы сможем продолжить синтаксический анализ.
2. В качестве синхронизирующего множества для ***А*** недостаточно использовать ***FOLLOW(A)***. К синхронизирующему множеству конструкций нижнего уровня можно добавить символы, с которых начинаются конструкции более высокого уровня.
3. Если мы добавим символы из ***FIRST(A)*** в синхронизирующее множество для нетерминала ***А***, станет возможным продолжение анализа в соответствии с ***А***, когда во входном потоке появится символ из множества ***FIRST(A)***.
4. Если нетерминал может порождать пустую строку, то в качестве продукции по умолчанию в качестве продукции может использоваться, порождающая ***ε***. Это может отсрочить обнаружение ошибки, зато не может вызвать ее пропуск, и сокращает число нетерминалов, которые должны быть рассмотрены в процессе восстановления после ошибки.
5. Если терминал на вершине стека не может быть сопоставлен с входным символом, то простейший метод состоит в снятии терминала со стека, генерации сообщения о вставке терминала в программу и продолжении синтаксического анализа.

## Условие:

1. По описанию языка построить ***КС***-грамматику.
2. Определить свойства полученной грамматики.
3. Если грамматика не обладает свойствами, требуемыми для построения таблицы предиктивного анализатора, то преобразовать грамматику к требуемой форме.
4. Определить значение функций ***FIRST*** и ***FOLLOW*** для разработанной грамматики.
5. Построить таблицу предиктивного анализатора.
6. Проверить правильность построения на трех примерах (один правильный, два неправильных).

## Варианты:

|  |  |
| --- | --- |
| № | Формулировка варианта задания |
|  | <E> ::= <E> ‘+’ <E> | <E> ‘\*’ <E> | ‘(‘ <E> ‘)’ | ‘i’ |
|  | <S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]  <E> ::= ‘i’ | ‘i’ ‘==’ <E>  <O> ::= ‘o’ … |
|  | 1  type:  2  formalParameter:  3  block:  4 |
|  | 1 |
|  | <P> ::= [ <H> ] <B>  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= <H> [ <B> ]  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= [ <H> ] [ <B> ]  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= <H> (<B>)…  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= [ <H> ] <B>  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… [‘;’] |
|  | <P> ::= [ <H> ] <B>  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… [‘;’] |
|  | <P> ::= <H> [ <B> ]  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= ‘b’ ( [ ‘,’ ] ‘b’)… ‘;’ |
|  | <P> ::= <H>… [ <B> ]  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= ‘b’ ( [ ‘,’ ] ‘b’)… [ ‘;’ ] |
|  | <P> ::= <H> [ <B>… ]  <H> ::= [‘h’] (‘t’ ‘i’)…  <B> ::= [‘b’] ( [‘,’] ‘b’)… ‘;’ |
|  | <P> ::= ([ <H> ] [ <B> ])…  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= (<H> | <B>)…  <H> ::= ‘h’ (‘t’ ‘i’)…  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <P> ::= <H> [ <B> ]  <H> ::= ‘h’ (‘t’ ‘i’)… | ε  <B> ::= ‘b’ (‘,’ ‘b’)… ‘;’ |
|  | <E> ::= <E> ‘-’ <E> | <E> ‘+’ <E> | ‘(‘ <E> ‘)’ | ‘i’ |
|  | <S> ::= ‘if’ <E> ‘then’ <O> [ ‘else’ <O> ]  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | 1 |
|  | h:1  2  b:  3 |
|  | 1  h:  2  b:  3 |
|  | 11  h:  2  b:  33 |
|  | 11  h:  22  b:  333 |
|  | <S> ::= ‘if’ [ <E> ] ( [ ‘i’ ‘:’ ] ‘then’ <O> )…  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ( ‘i’ ‘:’ ‘then’ <O> )…  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ‘then’ <O> | <O>  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ] | <O>  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <S> ::= ‘if’ [ <E> ] ‘then’ <O> [ ‘else’ <O> ]  <E> ::= ‘i’ | ‘i’ ‘<>’ <E>  <O> ::= ‘o’ <O> | <S> | ‘o’ |
|  | <E> ::= <E> ‘\*’ <E> | <E> ‘=’ <E> | ‘(‘ <E> ‘)’ | ‘i’ |

## Пример построения таблицы предиктивного анализатора:

1. Дано описание языка: ***<E> ::= <E> ‘+’ <E> | <E> ‘\*’ <E> | ‘(‘ <E> ‘)’ | ‘i’***
2. По описанию языка построили КС-грамматику:
3. Полученная грамматика является леворекурсивной и левофакторизованной. Преобразуем грамматику к требуемой форме.

***α1=+Е β1=(Е)***

***α2=\*Е β2=i***

***Е = β1E’/ β2E’***

***E’= α1E’/ α2E’/ε***

***E→ (E)E’***

***E → iE’***

***E’ → +EE’***

***E’ → \*EE’***

***E’ → ε***

1. ***E→TE’***
2. ***E’ → +TE’***
3. ***E’ → ε***
4. ***T →FT’***
5. ***T’ → \*FT’***
6. ***T’ → ε***
7. ***F → (E)***
8. ***F → d***
9. Определим значение функций ***FIRST*** и ***FOLLOW*** для разработанной грамматики.

**FIRSTFOLLOW**

***E={id, ( } E={$, ) }***

***E’={+, ε } E’ = {$, ) }***

***T = {id, ( } T = {+, $, ) }***

***T’ = {\*, ε } T’ = {+, $, ) }***

***F = {id, ( } F = {\*, $, ), + }***

1. Построим таблицу предиктивного анализатора. ***“Synch”*** указывает синхронизирующие токены, полученные из множества ***FOLLOW*** соответствующего терминала.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **id** | **+** | **\*** | **(** | **)** | **$** |
| **E** | E → TE’ |  |  | E →TE’ | Synch | Synch |
| **E’** |  | E’ → +TE’ |  |  | E’ → ε | E’ → ε |
| **T** | T →FT’ | Synch |  | T → FT’ | Synch | Synch |
| **T’** |  | T’ → ε | T’ →FT’ |  | T’ → ε | T’ → ε |
| **F** | F → id | Synch | Synch | F → (E) | Synch | Synch |

1. Проверим правильность построения на трех примерах.

**Правильный*: id\*(id+id)$***

|  |  |
| --- | --- |
| $E  $E’T  $E’T’F  $E’T’id  $E’T’  $E’T’F\*  $E’T’F  $E’T’)E(  $E’T’)E  $E’T’)E’T  $E’T’)E’T’F  $E’T’)E’T’id  $E’T’)E’T’  $E’T’)E’  $E’T’)E’T+  $E’T’)E’T  $E’T’)E’T’F  $E’T’)E’T’id  $E’T’)  $E’T’  $E’  $ | id\*(id+id)$  id\*(id+id)$  id\*(id+id)$  id\*(id+id)$  \*(id+id)$  \*(id+id)$  (id+id)$  (id+id)$  id+id)$  id+id)$  id+id)$  id+id)$  +id)$  +id)$  id)$  id)$  id)$  )$  )$  $  $  $ |

Строка разложена

**Неправильный: *)id\*+id$***

|  |  |  |  |
| --- | --- | --- | --- |
| $E  $E  $E’T  $E’T’F  $E’T’id  $E’T’  $E’T’F\*  $E’T’F  $E’T’  $E’  $E’T+  $E’T  $E’T’F  $E’T’id  $E’T’  $E’  $ | )id\*+id$  Id\*+id$  Id\*+id$  Id\*+id$  Id\*+id$  \*+id$  \*+id$  +id$  +id$  +id$  +id$  id$  id$  id$  $  $  $ | Ошибка, пропускаем «)»  Id € FIRST(E)  Ошибка, М[F,+] = Synch  F снимаем со стека |  |

Разложили строку с обработкой ошибок

**Неправильный: *id+(\*id)$***

|  |  |  |  |
| --- | --- | --- | --- |
| $E  $E’T  $E’T’F  $E’T’id  $E’T’  $E’  $E’T+  $E’T  $E’T’F  $E’T’)E(  $E’T’)E  $E’T’)E  $E’T’)E’T  $E’T’)E’T’F  $E’T’)E’T’  $E’T’)  $E’T’  $E’  $ | id+(\*id)$  id+(\*id)$  id+(\*id)$  id+(\*id)$  +(\*id)$  +(\*id)$  +(\*id)$  (\*id)$  (\*id)$  (\*id)$  \*id)$  Id)$  Id)$  Id)$  )$  )$  $  $  $ | Ошибка, пропускаем «\*»  Id € FIRST(E) |  |

Разложили строку с обработкой ошибки

# Задача «Построение КЗ-грамматики»

## Теоретическая часть

**Контекстно-зависимые (КЗ) грамматики и неукорачивающие грамматики**.

**Определение: Контекстно-зависимая грамматика** (**КЗ-грамматика**, **контекстная грамматика**) — частный случай формальной грамматики (тип 1 по иерархии Хомского), у которой левые и правые части всех продукций могут быть окружены терминальными и нетерминальными символами. Для грамматики ***G(VT,VN,P,S), V=VT∪VN*** все правила имеют вид:

* ***αAβ→αγβ***, где ***α***, ***β∈V\****, ***γ∈V+***, ***A∈VN***. Такие грамматики относят к контекстно-зависимым.
* ***α→β***, где ***α***, ***β∈V+***, ***|α|≤|β|***. Такие грамматики относят к неукорачивающим.

Эти классы грамматик эквивалентны. Могут использоваться при анализе текстов на естественных языках, однако при построении компиляторов практически не используются в силу своей сложности. Для контекстно-зависимых грамматик доказано утверждение: по некоторому алгоритму за конечное число шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет.

**Определение: НС-грамматикой** (или **грамматикой непосредственно-составляющих**) называется грамматика, правила которой имеют вид *aAb*® *agb*, где *A* О*N*, *a*, *b* О *V\**, *g* О *V*+. Цепочки *a* и *b*  задают **контекст**, в котором *A* можно заменять на *g*.

**Т е о р е м а.** Контекстно-зависимые языки рекурсивны.

**Доказательство**. Пусть задана неукорачивающая грамматика *G* и произвольная цепочка *a* О*T*+. Чтобы проверить, принадлежит ли *a* языку *L*(*G*), достаточно построить множество всех цепочек из *V*+, не превосходящих по длине *a*, и построить из этих цепочек последовательности, в которых цепочки не повторяются, упорядочены по длине (каждая следующая не короче предыдущей) и первая цепочка всегда *S*, а последняя - *a.* Таких последовательностей конечное множество. Остаётся проверить каждую из них, не является ли она выводом в *G*. Такая проверка завершается за конечное число шагов. Если вывод найден, то *a* О*L*(*G*), в противном случае - нет.

Отсюда следует, что любой рекурсивно перечислимый, но нерекурсивный язык (например, множество программ самоприменимых машин Тьюринга) является языком типа 0, но не языком типа 1.

Контекстно-зависимые грамматики содержат важный подкласс НС-грамматик.

Ниже перечислены некоторые **свойства контекстно-зависимых грамматик**:

* Все цепочки, последовательно выводимые из начального символа, имеют длину не меньшую, чем предыдущая, поскольку каждое правило должно оставлять длину цепочки неизменной или увеличивать ее1.
* Контекстно-зависимые грамматики генерируют цепочки, для хранения которых требуется фиксированный объем памяти. Например, такие грамматики способны распознавать цепочки вида anbncn, что не может сделать контекстно-свободная грамматика.
* Контекстно-зависимые грамматики обычно слишком сложны, чтобы быть практически полезными для моделирования языков программирования.

Контекстно-зависимые грамматики позволяют определять языковые структуры, не охваченные контекстно-независимыми грамматиками, но их практическое применение при создании анализаторов сопряжено с некоторыми сложностями следующего характера.

* При использовании контекстно-зависимых грамматик резко возрастает количество правил и нетерминальных символов. Представьте себе сложность контекстно-зависимой грамматики, необходимой для описания форм числа (единственного и множественного) и лица (первого, второго и третьего), а также всех остальных форм соглашений, принятых в английском языке.
* В контекстно-зависимых грамматиках размывается структура фраз языка, столь ясно представимая с помощью контекстно-независимых правил.
* При попытке описать более сложные соглашения и обеспечить семантическую со­гласованность самой грамматики теряются многие преимущества разделения син­таксического и семантического компонентов языка.
* Контекстно-зависимые грамматики не решают проблемы построения семантиче­ского представления значения текста. Анализатор, который просто принимает или отвергает предложение, никому не нужен. Он должен возвращать эффективное представление семантического значения предложений.

**Пример КЗ-грамматики:**

**G4 = ({B, C, S}, {a, b, c}, P, S)** где P:

**1. S → aSBC;**

**2. S → abc;**

**3. CB → BC;**

**4. bB → bb;**

**5. bC → bc;**

**6. cC → сc,**

порождает язык **{ a n b n c n }, n ≥ 1**.

По определению КЗ-грамматика не допускает правил: А **→ ε,** где **ε** - пустая цепочка. Т.е. КС-грамматика с пустыми цепочками в правой части правил не является контекстно-зависимой. Наличие пустых цепочек ведет к грамматике без ограничений.

## Условие:

1. По описанию языка построить порождающую грамматику.
2. Построить для трех примеров цепочки порождения.

## Варианты:

|  |  |
| --- | --- |
| **№** | **Формулировка задания** |
| **1** | {akbkck|k≥0} |
| **2** | {aibjcidj|i,j≥1} |
| **3** | {aibjck|1≤i≤j≤k} |
| **4** | {a2kbk|k≥0} |
| **5** | {a2k|k≥0} |
| **6** | {ak+1bk|k≥1} |
| **7** | {akbk+1ck|k≥1} |
| **8** | {akbkc2k|k≥0} |
| **9** | {ak+1bkck-1|k≥1} |
| **10** | {ak+1bick-1|k≥1, i≥2k} |
| **11** | {ak+ibk|k≥1, i≥0} |
| **12** | {aibj+1ck+2|1≤i≤j≤k} |
| **13** | {aib2jc3k|1≤i≤j≤k} |
| **14** | {aibjckdi|1≤i≤j≤k} |
| **15** | {aibjckdj|1≤i≤j≤k} |
| **16** | {aibjckdk|1≤i≤j≤k} |
| **17** | {aibj+1ckdi-1|1≤i≤j≤k} |
| **18** | {ai+1bjcidj|i,j≥1} |
| **19** | {aibj+1cidj|i,j≥1} |
| **20** | {aibjci+1dj|i,j≥1} |
| **21** | {aibjci+1dj+1|i,j≥1} |
| **22** | {ai+1bj-1cidj|i,j≥1} |
| **23** | {aibjcidj|i≥0,j≥1} |
| **24** | {aibjci|i≥0,j≥1} |
| **25** | {abjci|i≥0,j≥1} |
| **26** | {akbk+1ci|k≥1, i≥2} |
| **27** | {akbi+1ci|k≥1, i≥2} |
| **28** | {akbk+1cidk|k≥1, i≥2} |
| **29** | {a2kb2k+1cid2k|k≥1, i≥2} |
| **30** | {akbk+1c3idk|k≥1, i≥2} |

## Пример построения порождающей грамматики:

**Задан язык:{ anbncn | n≥1 }**

**Грамматика:**

1. S → aSBC
2. S → aBC
3. CB → HB
4. HB → HC
5. HC → BC
6. aB → ab
7. bB → bb
8. bC → bc
9. cC →cc

**Цепочка порождения aaa bbb ccc:**

→1 aSBC

→1 a**aSBC**BC

→2 aaaB**CB**CBC

→3 aaaB**HB**CBC

→4 aaaB**HC**CBC

→5 aaaB**BC**CBC

→3 aaaBBC**HB**C

→4 aaaBBC**HC**C

→5 aaaBBC**BC**C

→3 aaaBB**HB**CC

→4 aaaBB**HC**CC

→5 aaaBB**BC**CC

→6 aa**ab**BBCCC

→7 aaa**bb**BCCC

→7 aaab**bb**CCC

→8 aaabb**bc**CC

→9 aaabbb**cc**C

→9 aaabbbc**cc**

**Цепочка порождения aa bb cc:**

→1 aSBC

→2 a**aBC**BC

→3 aaB**HB**C

→4 aaB**HC**C

→5 aaB**BC**C

→6 aa**bB**CC

→7 aa**bb**CC

→8 aab**bc**C

→9 aabb**cc**

**Цепочка порождения a b c:**

→2 aBC

→6 **ab**C

→8 a**bc**

# Задача 4 «Построение таблицы SLR-анализатора»

## Теоретическая часть:

**Определение: *Восходящий синтаксический анализ*** – такой вид синтаксического анализа, который пытается строить дерево разбора для входной строки, начиная с листьев (снизу) и работая по направлению к корню дерева (вверх).

В данной задаче будет рассмотрен метод восходящего синтаксического анализа, известный как синтаксический анализ типа «перенос/свертка» (ПС-анализ).

**Определение: *LR-грамматика*** – такая грамматика, для которой возможно построить таблицу синтаксического анализа. Интуитивно, для того, чтобы грамматика была LR-грамматикой, достаточно, чтобы ПС-анализатор, читающий входной поток слева направо, был способен распознавать основы только при их появлении в стеке.

**Определение:** Рассмотрим наиболее простой метод LR-анализа, который называется ***SLR-анализом*** (Simple LR, простой LR). Недостатком этого метода является достаточно небольшое число грамматик, с которыми он работает, однако этот метод наиболее прост в реализации. Таблицу, построенную таким методом, будем называть SLR-таблицей, а синтаксический анализатор, работающий с SLR-таблицей, SLR-анализатором (а соответствующую грамматику – SLR-грамматикой).

**Определение: *LR(0)-пункт***, или ***элемент*** грамматики G – продукция G с точкой в некоторой позиции правой части. Следовательно, продукция, A -> XYZ дает четыре пункта:

1. A -> ∙XYZ
2. A -> X∙YZ
3. A -> XY∙Z
4. A -> XYZ

Продукция A -> ε генерирует только один пункт, A -> ∙ . Интуитивно, пункт указывает, какую часть продукции мы уже увидели в данной точке в процессе синтаксического анализа. Например, первый пункт, приведенный выше, определяет, что во входном потоке мы ожидаем встретить строку, порождаемую XYZ. Второй пункт указывает, что у нас уже есть строка, порожденная X, и мы ожидаем получить из входного потока строку, порождаемую YZ.

Система LR(0)-пунктов, которую назовем *канонической,* обеспечивает основу для построения SLR-анализаторов. Для построения канонической LR(0)-системы грамматики мы определим расширенную грамматику и две функции – ***closure*** и ***goto***.

Если G – грамматика со стартовым символом S, то G’, ***расширенная грамматика (augmented grammar)*** грамматика G, представляет собой G с новым стартовым символом S’ и продукцией S’ -> S. Назначение этой новой стартовой продукции – указать синтаксическому анализатору, когда он должен прекратить разбор и объявить о допущении входной строки. Таким образом, допуск строки происходит тогда и только тогда, когда синтаксический анализатор выполняет свертку, соответствующую продукции S’ -> S.

**Операция замыкания (closure).**

Если I – множество пунктов грамматики G, то closure(I) – множество пунктов, построенное из I по следующим правилам.

1. Изначально в closure(I) входят все пункты из I.
2. Если A -> α∙Bβ входит в closure(I) и B -> γ представляет собой продукцию, то добавляем в closure(I) пункт B -> ∙γ (если его там еще нет). Мы применяем это правило до тех пор, пока не внесем все возможные пункты в closure(I).

Наличие A -> α∙Bβ в closure(I) указывает, что в некоторый момент в процессе синтаксического анализа мы полагаем, что можем встретить во входном потоке подстроку, выводимую из Bβ. Но если имеется продукция B -> γ, то, естественно, мы также можем встретить в этот момент строку, выводимую из γ, поэтому включаем B -> ∙γ в closure(I).

**Операция goto.**

Второй полезной функцией является goto(I, X), где I является множеством пунктов, а X – символом грамматики. goto(I, X) определяется как замыкание (closure) множества всех пунктов [A -> α∙Bβ] ϵ I. Интуитивно, если I является множеством пунктов, допустимых для некоторого активного префикса γ, то goto(I, X) есть множество пунктов, допустимых для активного префикса γX.

**Алгоритм построения C системы множества пунктов для расширенной грамматики G’**

Приведем алгоритм, который позволяет построить каноническую систему LR(0)-пунктов для некоторой расширенной грамматики. Данный алгоритм использует описанные выше операции closure и goto.

**procedure** items(G’);

**begin**

C := {closure( { [S’ -> ∙S] } ) };

**repeat**

**for** каждое множество пунктов I в С и каждый символ грамматики X, такой, что goto(I, X) не является пустым и не принадлежит C **do**

добавить goto(I, X) к С

**until** больше нет множеств, которые можно добавить к C

**end**

**Алгоритм 4.1.**

**Алгоритм построения таблицы SLR-анализатора.**

Данный алгоритм строит функции ***action*** и ***goto*** SLR-анализа (эти функции будут описаны ниже при описании алгоритма LR-анализа (алгоритм 4.3.)). Он **не дает** однозначно определенные таблицы действий для всех грамматик, но он успешно работает для многих грамматик языков программирования. Данную грамматику G необходимо расширить до грамматики G’, и на основе G’ строим C – каноническую систему множества пунктов для G’ (алгоритм 4.1.). По системе C строим ***action***, функцию действий синтаксического анализа, и ***goto***, функцию переходов, в соответствии с приведенным далее алгоритмом. Он требует знания FOLLOW(A) для каждого нетерминала A грамматики.

***Вход.*** Расширенная грамматика G’

***Выход.*** Функции ***action*** и ***goto*** таблицы SLR-анализа для грамматики G’.

***Алгоритм.***

1. Построим C = {I0, I1, …, In} - систему множества LR(0)-пунктов для грамматики G’.
2. Состояние i строится на основе Ii. Действия синтаксического анализа для состояния i определяются следующим образом:
   1. Если [A -> α∙aβ] ϵ Ii и goto(Ii, a) = Ij, то определить action[i, a] как “перенос j”; здесь a должно быть нетерминалом;
   2. Если [A -> α∙] ϵ Ii, то определить action[i, a] как “свертка A -> α” для всех a из FOLLOW(A); здесь A не должно быть S’;
   3. Если [S’ -> S∙] ϵ Ii, то определить action[i, $] как “допуск”.

**Если по этим правилам генерируются конфликтующие**

**действия, мы говорим, что грамматика не является SLR(1).**

**Алгоритм не в состоянии построить синтаксический анализатор для нее. В этом случае необходимо преобразовать грамматику (возможно изменение языка) для SLR(1)-анализа.**

1. Переходы ***goto*** для состояния i и всех нетерминалов A строятся по правилу: если goto(Ii, A) = Ij, то goto[i,A] = j.
2. Все записи, не определенные по правилам (2) и (3), указываются как “ошибка”.
3. Начальное состояние синтаксического анализатора представляет собой состояние, построенное из множества пунктов, содержащего [S’ -> S].

**Алгоритм 4.2.**

Таблица синтаксического анализа, состоящая из функций ***action*** и ***goto***, определяемых алгоритмом 4.2., называется *SLR(1)-таблицей грамматики G*. LR-анализатор, использующий SLR(1)-таблицу для грамматики G, называется SLR(1)-анализатором для G, а соответствующая грамматика - SLR(1)-грамматикой. Обычно при указании SLR(1) часть “(1)” опускается, поскольку мы не работаем более чем с одним символом предпросмотра.

**Алгоритм LR-анализа.**

LR-анализатор состоит из входного потока, выхода, стека, управляющей программы и таблицы синтаксического анализа, состоящей из двух частей (*действие (action)* и *переход (goto)*). Управляющая программа для всех LR-анализаторов одна и та же; изменяются только таблицы синтаксического анализа. Программа синтаксического анализа считывает символы из входного буфера по одному и использует стек для хранения строк вида s0X1s1X2s2…Xmsm (sm находится на вершине стека). Каждый символ Xi является символом грамматики, а каждый si – символом, именуемым *состоянием*. Каждый символ состояния обобщает информацию, содержащуюся в стеке ниже его. Комбинация символа и состояния на вершине стека и текущего входного символа используется в качестве индекса таблицы синтаксического анализа и определяет дальнейшее действие – перенос или свертку. При реализации грамматические символы не обязаны появляться в стеке.

Таблица синтаксического анализа состоит из двух частей – функций действий синтаксического анализа ***action*** и функции переходов ***goto.***  Управляющая программа LR-анализатора функционирует следующим образом. Она определяет sm­, текущее состояние на вершине стека, и ai, текущий входной символ. Затем программа обращается к action[sm, ai], ячейке таблицы действий синтаксического анализа, определяемой состоянием sm и символами ai, которая может иметь одно из четырех значений:

1. перенос s, где s – состояние;
2. свертка в соответствии с продукцией A -> β;
3. допуск (accept);
4. ошибка (error);

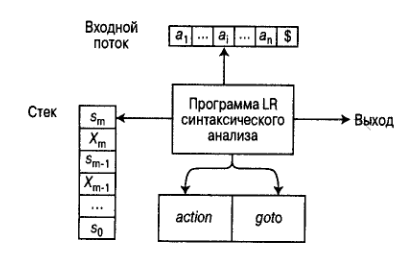


Рис. 4.1. Модель LR-анализатора

Функция goto получает в качестве аргументов состояние и символ грамматики и возвращает новое состояние.

*Конфигурация* LR-анализатора представляет собой пару, первый компонент которой – содержимое стека, а второй – непросмотренная часть входного потока: (s0X1s1X2s2…Xmsm, aiai+1…an$).

Следующий шаг синтаксического анализатора определяется текущим входным символом ai и состоянием на вершине стека sm в соответствии со значением ячейки таблицы action[sm, ai]. Конфигурации, получаемые после каждого из четырех типов действий следующие:

1. Если action[sm, ai] = “перенос s”, синтаксический анализатор выполняет перенос, переходя в конфигурацию (s0X1s1X2s2…Xmsmais, ai+1…an$). Синтаксический анализатор переносит в стек текущий входной символ ai и очередное состояние s, определяемое значением action[sm, ai]; текущим входным символом становится ai+1.
2. Если action[sm, ai] = “свертка A -> β”, то синтаксический анализатор выполняет свертку в конфигурацию (s0X1s1X2s2…Xm-rsm-rAs, aiai+1…an$), где s = goto[sm-r, ai], a r – длина β, правой части продукции. Здесь синтаксический анализатор вначале снимает со стека 2r символов (r символов состояний и r символов грамматики), выводя на вершину стека состояние sm-r. Затем вносит в стек A (левую часть продукции) и s, запись из ячейки goto[sm-r, A]. Текущий входной символ при этом не изменяется. Последовательность снимаемых со стека символов грамматики Xm-r+1…Xm всегда соответствуют β, правой части продукции свертки.
3. Если action[sm, ai] = “допуск”, синтаксический анализ завершается.
4. Если action[sm, ai] = “ошибка”, синтаксический анализатор обнаружил ошибку и вызывает подпрограмму восстановления после нее.

Полностью алгоритм LR-анализа приведен ниже. Все LR-анализаторы ведут себя одинаково; единственная разница между ними заключается в таблицах action и goto.

***Вход.*** Входная строка w и таблица LR-анализа с функциями action и goto для грамматики G.

***Выход.*** Если w ϵ L(G), выдается восходящий разбор для w; в противном случае выводится сообщение об ошибке.

***Алгоритм.*** Изначально синтаксический анализатор содержит в стеке начальное состояние s0, а во входном буфере – w$. Затем анализатор выполняет приведенный ниже алгоритм до тех пор, пока не будет достигнуто успешное завершение анализа или не обнаружена ошибка.

*Установить указатель ip на первый символ w$;*

**repeat forever begin**

*Пусть s – состояние на вершине стека, а a – символ, на который указывает ip*

**if** action[s,a] = “перенос s” **then begin**

Поместить в стек a, затем s’; переместить ip к следующему входному символу

**end**

**else if** action[s,a] = “свертка A -> β” **then begin**

Снять со стека 2\*|β| символов. Пусть s’ – текущее состояние на вершине стека. Поместить в стек A, затем goto[s’,A]; вывести продукцию A -> β.

**end**

**else if** action[s,a] = “допуск” **then**

**return**

**else** error()

**end**

***Пример части таблицы LR-анализа.***

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Состояние | action | | | | | | goto | | |
| id | + | \* | ( | ) | $ | E | T | F |
| 0 | s5 |  |  | s4 |  |  | 1 | 2 | 3 |
| 1 |  | s6 |  |  |  | accept |  |  |  |
| 2 |  | r2 | s7 |  | r2 | r2 |  |  |  |

* Поясним обозначения в данной таблице.  
  si означает перенос и i-е состояние на вершине стека,
* rj означает свертку в соответствии с продукцией номер j,
* accept означает допуск входной строки,
* пустая ячейка означает ошибку,
* В части ***action*** находятся терминальные символы грамматики, в части ***goto***, соответственно, нетерминальные.

**Восстановление после ошибок при LR-анализе.**

LR-анализатор обнаруживает ошибку при обращении к таблице ***action*** синтаксического анализа при нахождении там записи об ошибке (при обращении к таблице ***goto*** ошибки не выявляются).

Восстановление на уровне фразы реализуется путем проверки каждой ошибочной записи в таблице LR-анализа и принятия решения (на основе знания особенностей языка) о том, какая наиболее вероятная ошибка программиста могла привести к данной ситуации. После этого можно построить подходящую процедуру восстановления после ошибки; возможно, при этом придется изменить вершину стека и/или первые символы входного потока способом, соответствующим данной записи ошибки.

Таким образом, мы можем заполнить каждую пустую ячейку в таблице *action* указателем на программу обработки ошибок, которая будет выполнять некоторые действия, определенные разработчиком компилятора. Эти действия могут включать вставку символов в стек или входной поток и удаление их оттуда или изменение и перестановку входных символов, как и в случае синтаксического анализатора приоритета операторов. Как и в упомянутом случае, мы не должны допустить возможности зацикливания LR-анализатора. Стратегия, гарантирующая отсутствие зацикливания, требует либо удаления (переноса) из входного потока по меньшей мере одного символа, либо уменьшения стека или достижения конца входного потока. Снятие со стека состояния над нетерминалом следует избегать, поскольку такое изменение удаляет из стека успешно разобранную конструкцию.

## Условие:

1. Определить значение функций FIRST и FOLLOW для разработанной грамматики.
2. Построить множество пунктов.
3. Построить диаграмму переходов.
4. Построить таблицу SLR анализатора c учетом вариантов восстановления после ошибок.
5. Проверить правильность построения на трех примерах (один правильный, два неправильных).

## Варианты:

|  |  |
| --- | --- |
| № | Формулировка варианта задания |
|  | S -> Aa  A -> Ab | Bb  B -> cB | ε |
|  | S -> ABC  A -> BC | a  B -> A | b  C -> c | ε |
|  | S -> AB  A -> aAb | Ab | ε  B -> bB | b |
|  | S -> aAb  A -> BC | bc  B -> Aa  C -> c | ε |
|  | S -> AS | c  A -> Aa | Ca  C -> cA | c |
|  | S -> BS | c  A -> cB | c  B -> Ba | Aa |
|  | S -> ABc  A -> aAb | Ab | ε  B -> bB | bA |
|  | S -> Aa  A -> Ac | Bb | ε  B -> cB | ε |
|  | S -> aBS | c  A -> cB | c  B -> Ba | Aa | ε |
|  | S -> AC | c  A -> Aa | Ca  C -> cA | c |
|  | S -> SA | c  A -> Aa | Ca  C -> cA | c |
|  | S -> aAb  A -> BS | bc  B -> AaC  C -> c | ε |
|  | S -> aB | c  A -> cB | c  B -> Ba | Aa | ε |
|  | S -> ABC  A -> BC | a  B -> A | b  C -> c | ε |
|  | S -> AB  A -> aAb | Ab | ε  B -> bB | b |
|  | S -> aAb  A -> BC | bc  B -> Aa  C -> c | ε |
|  | S -> AS | c  A -> Aa | Ca  C -> cA | c |
|  | S -> BS | c  A -> cB | c  B -> Ba | Aa |
|  | S -> ABc  A -> aAb | Ab | ε  B -> bB | bA |
|  | S -> Aa  A -> Ac | Bb | ε  B -> cB | ε |
|  | S -> aBS | c  A -> cB | c  B -> Ba | Aa | ε |
|  | S -> AC | c  A -> Aa | Ca  C -> cA | c |
|  | S -> AC | c  A -> Aa | Ca  C -> cA | c |
|  | S -> SA | c  A -> Aa | Ca  C -> cA | c |
|  | S -> aAb  A -> BS | bc  B -> AaC  C -> c | ε |
|  | S -> aB | c  A -> cB | c  B -> Ba | Aa | ε |
|  | S -> ABC  A -> BC | a  B -> A | b  C -> c | ε |
|  | S -> AB  A -> aAb | Ab | ε  B -> bB | b |
|  | S -> aAb  A -> BC | bc  B -> Aa  C -> c | ε |
|  | S -> AS | c  A -> Aa | Ca  C -> cA | c |

## Пример построения таблицы SLR-анализатора:

1. Исходная грамматика для декларации функции:

* D -> i(P);
* P -> iF
* P -> ε
* F -> ,iF
* F -> ε

1. Построим функции FIRST и FOLLOW для грамматики, заданной в условии (подробнее построение этих функций описано в задаче 3).

FIRST(D) = {i} FOLLOW(D) = {$}

FIRST(P) = {i, ε} FOLLOW(P) = {)}

FIRST(F) ={‘,’, ε} FOLLOW(F) = {)}

1. Построим множество пунктов. Для этого пронумеруем правила и добавим фиктивное нулевое правило. После этого грамматика примет вид:
2. D’ -> D
3. D -> i(P);
4. P -> iF
5. P -> ε
6. F -> ,iF
7. F -> ε

А) Нулевой пункт I0 будет состоять из результата функции closure( { [D’ -> ∙D] } )

Б) Подсчитываем результат функции closure({[D’ -> ∙D]})

i) I0={ D’ -> ∙D}; Поскольку пункт стоит перед нетерминалом D, то пытаемся добавить в I0 пункт D -> ∙i(P); Такого пункта в I0 нет, поэтому добавляем его.

ii) I0 = { D’ -> ∙D, D -> ∙i(P);}. Поскольку пункт стоит только перед терминалами, а остальные нетерминалы уже проверялись, то множество пунктов I0 сформировано.

В) Подсчитываем I1 = goto(I0, D) = closure({[D’ -> D∙]})

i) I1 = {D’ -> D∙}. Поскольку пункт стоит только перед символами ε, то множество пунктов I1 сформировано.

Г) Подсчитываем I2 = goto(I0, i) = closure({[D -> i∙(P);]})

i) I2 = {D -> i∙(P);}. Поскольку пункт стоит только перед терминалами, то множество пунктов I2 сформировано.

Д) Подсчитываем I3 = goto(I2, ‘(‘) = closure ({[D -> i(∙P);]}).

i) I3 = {D -> i(∙P);}. Поскольку пункт стоит перед нетерминалом P, то пытаемся добавить в I3 пункты P -> ∙iF и P -> ∙ . Таких пунктов в I3 нет, поэтому добавляем их.

ii) I3 = {D -> i(∙P); , P -> ∙iF, P -> ∙}. Поскольку пункт стоит

только перед терминалами, а остальные нетерминалы уже

проверялись, то множество пунктов I3 сформировано.

Е) Подсчитываем I4 = goto(I3,P) = closure ({[D -> i(P∙);]}).

i) I4 = {D -> i(P∙);}. Поскольку пункт стоит только перед терминалами, то множество пунктов I4 сформировано.

Ж) Подсчитываем I5 = goto(I3, i) = closure({[P -> i∙F]}).

i) I5 = {P -> i∙F}. Поскольку пункт стоит перед нетерминалом F, то пытаемся добавить в I6 пункты F -> ∙,iF и F -> ∙ . Таких пунктов в I5 нет, поэтому добавляем их.

ii) I5 = {P -> i∙F, F -> ∙,iF, F -> ∙}. Поскольку пункт стоит

только перед терминалами, а остальные нетерминалы уже

проверялись, то множество пунктов I5 сформировано.

З) Подсчитываем I6 = goto(I4, ‘)’) = closure({[D -> i(P)∙;]}).

i) I6 = {D -> i(P)∙;}. Поскольку пункт стоит только перед

терминалами, то множество пунктов I6 сформировано.

И) Подсчитываем I­7 = goto(I5, F) = closure({[P -> iF∙]}).

i) I7 = {P -> iF∙}. Поскольку пункт стоит только перед

терминалами, то множество пунктов I7 сформировано.

K) Подсчитываем I8 = goto(I5, ‘,’) = closure({[F ->,∙iF]}).

i) I8 = {F ->,∙iF}. Поскольку пункт стоит только перед

терминалами, то множество пунктов I8 сформировано.

И) Подсчитываем I9 = goto(I6, ;) = closure({[D -> i(P);∙]}).

i) I9 = {D -> i(P);∙}. Поскольку пункт стоит только перед

терминалами, то множество пунктов I9 сформировано.

Л) Подсчитываем I10 = goto(I­8, i) = closure({[F ->,i∙F]}).

i) I10 = {F ->,i∙F}. Поскольку пункт стоит перед нетерминалом F, то пытаемся добавить в I10 пункты F -> ∙,iF и F -> ∙ . Таких пунктов в I10 нет, поэтому добавляем их.

ii) I10 = {F ->,i∙F, F -> ∙,iF, F -> ∙}. Поскольку пункт стоит

только перед терминалами, а остальные нетерминалы уже

проверялись, то множество пунктов I10 сформировано.

М) Подсчитываем I11 = goto(I10, F) = closure({[F ->,iF∙]}).

i) I11 = {F ->,iF∙}. Поскольку пункт стоит только перед

терминалами, то множество пунктов I11 сформировано.

Л) Подсчитываем I12 = goto(I10, ‘,’) = closure({[F ->,∙iF]}).

i) I12 = {F ->,∙iF}. Поскольку пункт стоит только перед

терминалами, то множество пунктов I12 сформировано. Замечаем, что I12­ = I8, поэтому не формируем нового множества пунктов. Таким образом, результат этого шага I8 = goto(I10, ‘,’) = {F ->,∙iF}.

Н) Больше множеств пунктов построить нельзя, поскольку либо все пункты расположены либо после символов ε, либо не порождают новых состояний.

Приведем подпрограммы для обработки ошибок.

Множество пунктов для данной грамматики имеет вид:

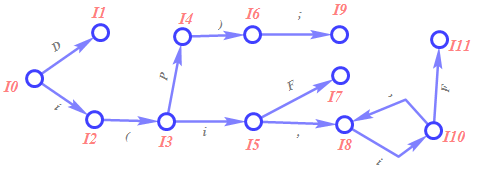
* I0 = {D’ -> ∙D, D -> ∙i(P);}
* I1 = {D’ -> D∙}
* I2 = {D -> i∙(P);}
* I3 = {D -> i(∙P); , P -> ∙iF, P -> ∙}
* I4 = {D -> i(P∙);}
* I5 = {P -> i∙F, F -> ∙,iF, F -> ∙}
* I6 = {D -> i(P)∙;}
* I7 = {P -> iF∙}
* I8 = {F ->,∙iF}
* I9 = {D -> i(P);∙}
* I10 = {F ->,i∙F, F -> ∙,iF, F -> ∙}
* I11 = {F ->,iF∙}

1. Диаграмма переходов представляет собой ориентированный граф, вершинами которого являются состояния, а дугами – переходы между этими состояниями.

Матрица смежности этого графа будет иметь вид.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Состояние | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 0 | - | D | i | - | - | - | - | - | - | - | - | - |
| 1 | - | - | - | - | - | - | - | - | - | - | - | - |
| 2 | - | - | - | ( | - | - | - | - | - | - | - | - |
| 3 | - | - | - | - | P | i | - | - | - | - | - | - |
| 4 | - | - | - | - | - | - | ) | - | - | - | - | - |
| 5 | - | - | - | - | - | - | - | F | , | - | - | - |
| 6 | - | - | - | - | - | - | - | - | - | ; | - | - |
| 7 | - | - | - | - | - | - | - | - | - | - | - | - |
| 8 | - | - | - | - | - | - | - | - | - | - | i | - |
| 9 | - | - | - | - | - | - | - | - | - | - | - | - |
| 10 | - | - | - | - | - | - | - | - | , | - | - | F |
| 11 | - | - | - | - | - | - | - | - | - | - | - | - |

Диаграмма переходов, построенная на основании приведенной выше матрицы смежности, имеет вид:



1. Строим таблицу SLR-анализатора.

Рассматриваем последовательно каждое состояние, заполняя функции ***action*** и ***goto*** в таблице.

А) Для множества пунктов I0 получаем action[I0, i] = s2 по правилу 2.1. алгоритма 4.2., goto[I0, D] = 1 по правилу 3 алгоритма 4.2..

Б) Для множества пунктов I1 получаем action[I1, $] = accept по правилу 5 алгоритма 4.2.

В) Для множества пунктов I2 получаем action[I2, (] = s3 по правилу 2.1. алгоритма 4.2.

Г) Для множества пунктов I3 получаем action[I3, )] = r3 по правилу 2.2. алгоритма 4.2., action[I3, i] = s5 по правилу 2.1. алгоритма 4.2., goto[I3, P] = 4 по правилу 3 алгоритма 4.2.

Д) Для множества пунктов I4 получаем action[I4, )] = s6 по правилу 2.1. алгоритма 4.2.

Е) Для множества пунктов I5 получаем action[I5, )] = r5 по правилу 2.2. алгоритма 4.2., action[I5, ‘,’] = s8 по правилу 2.1. алгоритма 4.2., goto[I5, F] = 7

Ж) Для множества пунктов I6 получаем action[I6, ;] = s9 по правилу 2.1. алгоритма 4.2..

З) Для множества пунктов I7 получаем action[I7, )] = r2 по правилу 2.2. алгоритма 4.2..

И) Для множества пунктов I8 получаем action[I8, i] = s10 по правилу 2.1. алгоритма 4.2.

К) Для множества пунктов I9 получаем action[I9, $] = r1 по правилу 2.2. алгоритма 4.2..

Л) Для множества пунктов I10 получаем action[I10, )] = r5 по правилу 2.2. алгоритма 4.2., action[I10, ‘,’] = s8 по правилу 2.1. алгоритма 4.2., goto[I10, F] = 11 по правилу 3 алгоритма 4.2.

М) Для множества пунктов I11 получаем action[I10, )] = r4 по правилу 2.2. алгоритма 4.2.

Опишем работу подпрограмм, обеспечивающих обработку ошибок:

**e1:** Эта подпрограмма вызывается из состояния 0. Ожидает идентификатор функции. Вместо этого обнаруживает любой другой символ грамматики.

Помещает в стек символ i, а также размещает над ним состояние 2.

Выдает сообщение об ошибке «Нет названия функции»

**e2:** Эта подпрограмма вызывается из состояний 1, 2, 6, 7, 9 при обнаружении правой скобки во входном потоке.

Удаляет символ ‘)’ из входного потока.

Выдает сообщение об ошибке «Несбалансированная правая скобка»

**e3:** Эта подпрограмма вызывается из состояния 2 при обнаружении символа, отличного от левой скобки, во входном потоке.

Помещает в стек символ ‘(’, а так же размещает над ним состояние 3.

Выдает сообщение об ошибке «Отсутствует открывающаяся скобка при описании параметров»

**e4:** Эта подпрограмма вызывается из состояния 3 при обнаружении символа, отличного от i или правой скобки, во входном потоке.

Удаляет из входного потока неправильный символ.

Выдает сообщение об ошибке «Неправильное описание параметров»

**e5:** Эта подпрограмма вызывается из состояний 4, 8, 11 при обнаружении символа, отличного от правой скобки, во входном потоке. В состоянии 5 только при обнаружении символа ‘;’.

Помещает в стек символ ‘)’, а так же размещает над ним состояние 6.

**e6:** Эта подпрограмма вызывается из состояний 5, 10 при обнаружении символа, отличного от правой скобки или запятой, во входном потоке.

Удаляет из входного потока неправильный символ.

Выдает сообщение об ошибке «Неправильное перечисление параметров»

**e7:** Эта подпрограмма вызывается из состояния 6 при обнаружении символа, отличного от точки с запятой, во входном потоке.

Помещает в стек символ ‘;’, а так же размещает над ним состояние 9.

Выдает сообщение об ошибке «Отсутствует точка с запятой после описания функции»

**e8:** Эта подпрограмма вызывается из состояний 1, 9 при обнаружении любого символа данной грамматики во входом потоке.

Удаляет неправильный символ из входного потока.

Выдает сообщение об ошибке «Лишние символы после описания функции»

**e9:** Эта подпрограмма вызывается из состояния 7 при обнаружении символа, отличного от i, во входном потоке.

Помещает в стек символ i, а также размещает над ним состояние 10.

Выдает сообщение об ошибке «Отсутствует идентификатор параметра»

В итоге таблица имеет следующий вид:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Состояние | action | | | | | | goto | | |
| ( | ) | , | ; | i | $ | D | F | P |
| 0 | e1 | e1 | e1 | e1 | s2 | e1 | 1 |  |  |
| 1 | e8 | e2 | e8 | e8 | e8 | accept |  |  |  |
| 2 | s3 | e2 | e3 | e3 | e3 | e3 |  |  |  |
| 3 | e4 | r3 | e4 | e4 | s5 | e4 |  |  | 4 |
| 4 | e5 | s6 | e5 | e5 | e5 | e5 |  |  |  |
| 5 | e6 | r5 | s7 | e5 | e6 | e6 |  | 8 |  |
| 6 | e7 | e2 | e7 | s9 | e7 | e7 |  |  |  |
| 7 | e9 | e2 | e9 | e9 | s10 | e9 |  |  |  |
| 8 | e5 | r2 | e5 | e5 | e5 | e5 |  |  |  |
| 9 | e8 | e2 | e8 | e8 | e8 | r1 |  |  |  |
| 10 | e6 | r5 | s7 | e6 | e6 | e6 |  | 11 |  |
| 11 | e5 | r4 | e5 | e5 | e5 | e5 |  |  |  |

1. Примеры разбора текстов, порожденных данной грамматикой с помощью построенной таблицы SLR-анализатора.

***Пример1.*** i(i,i,i);

|  |  |  |
| --- | --- | --- |
| Стек | Входной поток | Комментарии |
| $0 | i(i,i,i);$ | *action[I0, i] = s2*  Осуществляем перенос i в стек. А так же добавляем в него состояние 2. |
| $0i2 | (i,i,i);$ | *action[I2, (] = s3*  Осуществляем перенос ‘(‘ в стек. А так же добавляем в него состояние 3. |
| $0i2(3 | i,i,i);$ | *action[I3, i] = s5*  Осуществляем перенос i в стек. А так же добавляем в него состояние 5. |
| $0i2(3i5 | ,i,i);$ | *action[I5, ‘,’] = s7*  Осуществляем перенос ‘,’ в стек. А так же добавляем в него состояние 7. |
| $0i2(3i5,7 | i,i);$ | *action[I7, i] = s10*  Осуществляем перенос i в стек. А так же добавляем в него состояние 10. |
| $0i2(3i5,7i10 | ,i);$ | *action[I10, ‘,’] = s7*  Осуществляем перенос ‘,’ в стек. А так же добавляем в него состояние 7. |
| $0i2(3i5,7i10,7 | i);$ | *action[I7, i] = s10*  Осуществляем перенос i в стек. А так же добавляем в него состояние 10. |
| $0i2(3i5,7i10,7i10 | );$ | *action[I10, )] = r5*  Осуществляем свертку по правилу грамматики номер 5. Снимаем со стека 2\*0 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I10,F] = 11* (10 – текущее состояние вершины стека) |
| $0i2(3i5,7i10,7i10F11 | );$ | *action[I11, )] = r4*  Осуществляем свертку по правилу грамматики номер 4. Снимаем со стека 2\*3 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I10,F] = 11* (10 – текущее состояние вершины стека) |
| $0i2(3i5,7i10F11 | );$ | *action[I11, )] = r4*  Осуществляем свертку по правилу грамматики номер 4. Снимаем со стека 2\*3 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I5,F] = 8* (5 – текущее состояние вершины стека) |
| $0i2(3i5F8 | );$ | *action[I8, )] = r2*  Осуществляем свертку по правилу грамматики номер 2. Снимаем со стека 2\*2 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I2,P] = 8* (2 – текущее состояние вершины стека) |
| $0i2(3P4 | );$ | *action[I4, )] = s6*  Осуществляем перенос ‘)’ в стек. А так же добавляем в него состояние 6. |
| $0i2(3P4)6 | ;$ | *action[I6, ;] = s9*  Осуществляем перенос ‘;’ в стек. А так же добавляем в него состояние 9. |
| $0i2(3P4)6;9 | $ | *action[I9, $] = r1*  Осуществляем свертку по правилу грамматики номер 1. Снимаем со стека 2\*5 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I0,D] = 1* (0 – текущее состояние вершины стека) |
| $0D1 | $ | *action[I1, $] = accept*  Попадаем в ячейку «accept». Строка успешно разобрана |

***Пример 2.*** i();i

|  |  |  |
| --- | --- | --- |
| Стек | Входной поток | Комментарии |
| $0 | i();i$ | *action[I0, i] = s2*  Осуществляем перенос i в стек. А так же добавляем в него состояние 2. |
| $0i2 | ();i$ | *action[I2, (] = s2*  Осуществляем перенос ‘(‘ в стек. А так же добавляем в него состояние 3. |
| $0i2(3 | );i$ | *action[I3, )] = r3*  Осуществляем свертку по правилу грамматики номер 3. Снимаем со стека 2\*0 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I3,P] = 4* (3 – текущее состояние вершины стека) |
| $0i2(3P4 | );i$ | *action[I4, )] = s6*  Осуществляем перенос ‘)‘ в стек. А так же добавляем в него состояние 6. |
| $0i2(3P4)6 | ;i$ | *action[I6, ;] = s9*  Осуществляем перенос ‘;‘ в стек. А так же добавляем в него состояние 9. |
| $0i2(3P4)6;9 | i$ | *action[I9, i] = e8*  Возникла ошибочная ситуация. В соответствии с правилом удаляем символ i из входного потока. |
| $0i2(3P4)6;9 | $ | *action[I9, $] = r1*  Осуществляем свертку по правилу грамматики номер 1. Снимаем со стека 2\*5 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I0,D] = 1* (0 – текущее состояние вершины стека) |
| $0D1 | $ | *action[I1, $] = accept*  Попадаем в ячейку «accept». Строка успешно разобрана |

***Пример 3.*** ,i;

|  |  |  |
| --- | --- | --- |
| Стек | Входной поток | Комментарии |
| $0 | ,i;$ | *action[I0, ‘,’] = e1*  Возникла ошибочная ситуация. В соответствии с правилом добавляем в стек символ i, а так же состояние 2. |
| $0i2 | ,i;$ | *action[I2, ‘,’] = e3*  Возникла ошибочная ситуация. В соответствии с правилом добавляем в стек символ ‘(‘, а так же состояние 3. |
| $0i2(3 | ,i;$ | *action[I3, ‘,’] = e4*  Возникла ошибочная ситуация. В соответствии с правилом удаляем из входного потока символ ‘,’. |
| $0i2(3 | i;$ | *action[I3, i] = s5*  Осуществляем перенос i в стек. А так же добавляем в него состояние 5. |
| $0i2(3i5 | ;$ | *action[I5, ;] = e5*  Возникла ошибочная ситуация. В соответствии с правилом добавляем в стек символ ‘)‘, а так же состояние 6. |
| $0i2(3i5)6 | ;$ | *action[I6, ;] = s9*  Осуществляем перенос ‘;’ в стек. А так же добавляем в него состояние 59. |
| $0i2(3i5)6;9 | $ | *action[I9, $] = r1*  Осуществляем свертку по правилу грамматики номер 1. Снимаем со стека 2\*5 символов, добавляем в него сворачиваемый нетерминал, а так же состояние, которое указано в ячейке *goto[I0,D] = 1* (0 – текущее состояние вершины стека) |
| $0D1 | $ | *action[I1, $] = accept*  Попадаем в ячейку «accept». Строка успешно разобрана |

# Задача «Построение операторной грамматики»

***Теоретическая часть:***

**Определение:** Грамматика, в которой нет продукций, правые части которых представляют собой ε или имеют два соседних нетерминала, называются **операторными грамматиками (operator grammar).**

Опишем простую в реализации технологию синтаксического анализа, называемую **анализом приоритета операторов**. Однако в силу своей простоты технология имеет массу недостатков. Например, с ее помощью сложно обрабатывать токены вроде знака “минус”, который имеет два разных приоритета – в зависимости от того, является ли он унарным или бинарным. Более того, поскольку связь между грамматикой анализируемого языка и синтаксическим анализатором приоритета операторов весьма слабая, нельзя быть уверенным, что синтаксический анализатор допускает именно тот язык, который нас интересует. Наконец, класс грамматик, с которыми может работать такой анализатор, весьма невелик.

**Определение:** При синтаксическом анализе приоритета операторов мы определим три непересекающихся ***отношения приоритетов (predence relations)*** - ∙>, <∙,  - между терминалами. Эти отношения приоритетов управляют выбором основ и имеют следующее содержание.

|  |  |
| --- | --- |
| Отношение | Значение |
| a <∙ b | a “уступает приоритет” b |
| a  b | a имеет тот же приоритет, что и b |
| a ∙> b | a “забирает приоритет у” b |

В отличие от арифметических отношений “меньше”, “равно”, “больше”, отношения приоритетов имеют совершенно иные свойства. Например, в одном и том же языке может быть так, что a <∙ b, и a ∙> b, или для некоторых терминалов a и b не выполняется ни одно из отношений a <∙ b, a ∙> b и a  b.

Имеется два обычных способа определения отношений приоритетов, которые должны выполняться между терминалами. Первый метод – интуитивный. Он основан на традиционных понятиях ассоциативности и приоритета операторов. Например, если \* имеет приоритет выше, чем +, то

\* ∙> +. Такой подход разрешает неоднозначности грамматики и позволяет написать для нее анализатор приоритета операторов (хотя унарный минус и вызывает определенные проблемы).

По второму методу выбора отношений приоритетов операторов вначале строится однозначная грамматика языка, которая отражает правильную ассоциативность и приоритеты операторов в своих деревьях разбора. Получив однозначную грамматику, можно воспользоваться механическим методом построения на ее основе отношений приоритетов операторов. Эти отношения могут пересекаться и задавать язык, отличающийся от порождаемого исходной грамматикой.

В данной задаче будет рассмотрен первый метод.

Рассмотрим **алгоритм синтаксического анализа приоритета операторов**.

***Вход.*** Входная строка w и таблица отношений приоритетов.

***Выход.*** В случае корректной строки w – скелетное дерево разбора с маркером – нетерминалом E, помечающим внутренние узлы; в противном случае – сообщение об ошибке.

***Алгоритм.*** Вначале стек содержит $, а входной буфер – w$.

Устанавливаем указатель ip на первый символ w$.

**repeat forever**

**if** на вершине стека находится $ и ip указывает на $

**then return**

**else begin**

Пусть a – терминал на вершине стека, а b – символ, на который указывает ip.

**if** a <∙ b или а  b **then begin**

Поместить b в стек

Переместить ip к следующему входному символу

**end**

**else if** a ∙> b **then**

**repeat**

Снять со стека

**until** Терминал на вершине стека связан отношением <∙ с последним снятым терминалом

**else** error()

**end**

**Алгоритм .1.**

Приоритет операторов может быть задан любым подходящим способом, однако это не всегда гарантирует корректную работу синтаксического анализатора с данной грамматикой. Для грамматик наподобие грамматики арифметических выражений можно использовать следующие эвристические правила, которые задают корректное множество отношений приоритетов.

**Правила построения таблицы приоритетов операторов.**

1. Если оператор θ1 имеет более высокий приоритет, чем оператор θ1, определим, что θ1 ∙> θ2 и θ2 <∙ θ1.
2. Если θ1 и θ2 представляют собой операторы равного приоритета (в частности, это может быть один и тот же оператор), то устанавливаем, что θ1 ∙> θ2 и θ2 ∙> θ1, если операторы левоассоциативны, и θ1 <∙ θ2 и θ2 <∙ θ1 в случае правоассоциативности.
3. Для всех операторов θ определяем отношения θ <∙ id, id ∙> θ, θ <∙ (, ( <∙ θ, ) ∙> θ, θ ∙> ), θ ∙> $, $ <∙ θ. Кроме того считаем, что (  ), $ <∙ (, $ <∙ id, ( <∙ (, id ∙> $, ) ∙> $, ( <∙ id, id ∙> ), ) ∙> ).

**Обработка унарных операторов.**

Если в грамматике имеется унарный оператор типа ¬ (логическое отрицание), который не имеет бинарного аналога, то его можно внести в описанную выше схему для создания отношений приоритета операторов. Предполагая, что ¬ - унарный префиксный оператор, мы определяем, что θ ∙> ¬ для любого оператора θ, независимо от того, унарный он или бинарный. Кроме того, устанавливаем, что ¬ ∙> θ, если ¬ имеет более высокий приоритет, чем θ, и ¬ <∙ θ в противном случае.

Если же в грамматике присутствует унарный оператор, который имеет бинарный аналог, то наилучшим решением было бы распознавание унарного оператора лексическим анализатором и замена на какой-то другой символ. Далее действовать так же, как описано выше.

**Функции приоритета.**

Компиляторы, использующие синтаксические анализаторы приоритета операторов не нуждаются в хранении отношений приоритетов. В большинстве случаев таблица может быть закодирована двумя ***функциями приоритетов (precedence functions)*** f и g, отображающими терминальные символы в целые числа. Мы пытаемся выбрать f и g такими, чтобы для символов a и b было справедливо:

* f(a) < g(b) при a <∙ b
* f(a) = g(b) при a  b
* f(a) > g(b) при a ∙> b

Таким образом, отношение приоритетов между a и b можно определить сравнением числовых значений f(a) и g(b). Заметим, однако, что при этом теряется смысл пустых ячеек таблицы, информирующих об ошибке, поскольку всегда выполняется одно из трех приведенных условий, независимо от того, чему равны f(a) и g(b). Потеря возможности обнаружения ошибок, вообще говоря, не считается достаточно серьезной, чтобы отказаться от использования функций приоритета там, где это возможно; ошибки могут быть обнаружены при вызове процедуры свертки для необнаруженной основы.

Не всякая таблица отношений приоритетов имеет функции приоритета для ее кодирования, но обычно на практике эти функции существуют.

**Алгоритм построения функций приоритетов.**

***Вход.*** Матрица приоритетов операторов.

***Выход.*** Функции приоритетов, представляющие входную матрицу, либо сообщение, что такие функции не существуют.

***Алгоритм.***

1. Создаем символы fa и ga для каждого символа a, являющегося терминалом или $.
2. Разделим созданные символа на как можно большее число групп таким образом, что если a  b, то fa и gb принадлежат одной группе. Возможно, нам придется поместить в одну группу символы, даже не связанные отношением . Например, если a  b и b  с, то fa и fc должны находиться в одной и той же группе, поскольку они оба находятся в той же группе, что и gb. Если ,кроме того, c  d, то fa­ и gd находятся в одной группе, даже если не выполняется условие a  d
3. Создадим ориентированный граф, узлы которого представляют собой группы, найденные в (2). Для всех a и b, если a <∙ b, проведем дугу из группы fa­ в gb. Отметим, что дуга (или путь) от fa к gb означает, что f(a) превосходит g(b); путь от gb к fa означает, что g(b) должно превосходить f(a).
4. Если построенный в (3) граф имеет циклы, это говорит о том, что функции приоритетов для данной таблицы не существует. Если циклов нет, то значением f(a) служит длина самого длинного пути, начинающегося в группе fa; соответственно, значение g(a) равно длине самого длинного пути из группы ga.

**Алгоритм .2.**

***Условие:***

1. По операторной грамматике и описанию приоритетов и ассоциативности операторов построить таблицу приоритетов.
2. Построить функции приоритетов для данной грамматики.
3. Разобрать 3 выражения с помощью таблицы приоритетов. (1 правильное выражение, 2 неправильных)

## Варианты:

Опишем операторную грамматику:

1. E -> E + E
2. E -> E – E
3. E -> E \* E
4. E -> E / E
5. E -> E % E
6. E -> E ^ E
7. E -> E & E
8. E -> E # E
9. E -> + E
10. E -> - E
11. E -> \* E
12. E -> / E
13. E -> % E
14. E -> ^ E
15. E -> & E
16. E -> # E
17. E -> (E)
18. E -> id

Приоритеты операторов возрастают снизу вверх:

+, -

\*, /

%, ^

&, #

Ассоциативность операторов:

Левоассоциативные: +, -, \*, /, %

Правоассоциативные: ^, #, &

1. acegirq
2. bcfhjrq
3. adfgkrq
4. bcfhlrq
5. acfgmrq
6. abcgnrq
7. abfgorq
8. cdegprq
9. abegirq
10. bcegjrq
11. acehkrq
12. bcehlrq
13. adehmrq
14. abchnrq
15. cdehorq
16. aefgprq
17. acefirq
18. cdfhjrq
19. abehkrq
20. adeglrq
21. bcfgmrq
22. adfhnrq
23. bdfgorq
24. abdgprq
25. abfhirq
26. cdfgjrq
27. abdhkrq
28. bdfhlrq
29. acfhmrq
30. bdegnrq

***Пример построения операторной грамматики:***

1. abcdfjqr:

* E -> E + E (a)
* E -> E – E (b)
* E -> E \* E (c)
* E -> E / E (d)
* E -> E ^ E (f)
* E -> - E (j)
* E -> (E) (q)
* E -> id (r)

1. Для того, чтобы избежать конфликтов заменим правило E -> -E на E -> uE, где u обозначает унарный минус.

Построим таблицу приоритетов операторов для данной грамматики.

По условию:

+ <∙ \*, - <∙ \*, + <∙ /, - <∙ /, \* <∙ ^, / <∙ ^, + <∙ ^, - <∙ ^, + <∙ u, - <∙ u, \* <∙ u, / <∙ u, ^ <∙ u. Поэтому в силу правила 1\* ∙> +, \* ∙> -, / ∙> +, / ∙> -, ^ ∙> \*, ^ ∙> /, ^ ∙> +, ^ ∙> -, u ∙> +, u ∙> -, u ∙> \*, u ∙> /, u ∙> ^.

Операторы +, - имеют одинаковый приоритет, а также левоассоциативные, поэтому в силу правила 2 + ∙> +, + ∙> -, - ∙> +, - ∙> -.

Операторы \*, / имеют одинаковый приоритет, а также левоассоциативные, поэтому в силу правила 2 \* ∙> \*, \* ∙> /, / ∙> \*, / ∙> /.

Оператор u является унарным, поэтому он не может встретиться подряд два раза. В силу этого оставляем ячейку [u,u] таблицы пустой.

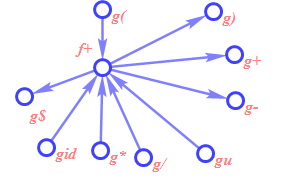
Оператор ^ является правоассоциативным, поэтому в силу правила 2 ^ <∙ ^.

Остальные ячейки таблицы заполняются в соответствии с правилом 3.

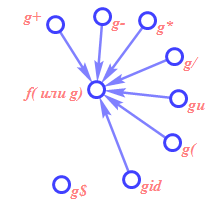
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | + | - | \* | / | ^ | u | ( | ) | id | $ |
| + | ∙> | ∙> | <∙ | <∙ | <∙ | <∙ | <∙ | ∙> | <∙ | ∙> |
| - | ∙> | ∙> | <∙ | <∙ | <∙ | <∙ | <∙ | ∙> | <∙ | ∙> |
| \* | ∙> | ∙> | ∙> | ∙> | <∙ | <∙ | <∙ | ∙> | <∙ | ∙> |
| / | ∙> | ∙> | ∙> | ∙> | <∙ | <∙ | <∙ | ∙> | <∙ | ∙> |
| ^ | ∙> | ∙> | ∙> | ∙> | <∙ | <∙ | <∙ | ∙> | <∙ | ∙> |
| u | ∙> | ∙> | ∙> | ∙> | ∙> |  | <∙ | ∙> | <∙ | ∙> |
| ( | <∙ | <∙ | <∙ | <∙ | <∙ | <∙ | <∙ |  | <∙ |  |
| ) | ∙> | ∙> | ∙> | ∙> | ∙> | ∙> |  | ∙> |  | ∙> |
| id | ∙> | ∙> | ∙> | ∙> | ∙> | ∙> |  | ∙> |  | ∙> |
| $ | <∙ | <∙ | <∙ | <∙ | <∙ | <∙ | <∙ |  | <∙ |  |

1. В соответствии с алгоритмом .2. и таблицей, построенной в предыдущем пункте задания, построим ориентированный граф функций приоритетов. Найдя в нем наибольшие пути для каждого символа, получим значения для функций приоритетов.

Весь граф будет выглядеть ненаглядно, поэтому приведем только его часть.



Данный подграф отражает первую строку таблицы. Остальные строки строятся аналогично. В случае равенства приоритетов две функции соединяются в одну вершину. Как будет показано далее.



Далее в полученном графе находим самые длинные пути от каждой группы f и g. Они и будут являться соответствующими значениями функций приоритетов.

В итоге получаем таблицу значений функций приоритетов.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | + | - | \* | / | ^ | u | ( | ) | id | $ |
| f | 2 | 2 | 4 | 4 | 6 | 7 | 0 | 10 | 12 | 0 |
| g | 1 | 1 | 3 | 3 | 5 | 8 | 9 | 0 | 11 | 0 |

1. Приведем пример разбора трех операторных выражений.

Подчеркиванием обозначается символ, на котором установлен указатель.

А) (id+id\*id)^id^id-id

|  |  |  |  |
| --- | --- | --- | --- |
| Входной поток | Стек | Результат | Комментарий |
| **(**id+id\*id)^id^id-id$ | $ |  | Помещаем в стек ‘(‘, т.к. $ <∙ ‘(‘ |
| **id**+id\*id)^id^id-id$ | ($ |  | Помещаем в стек id, т.к. ‘(‘<∙ id |
| **+**id\*id)^id^id-id$ | id($ | id | Снимаем id со стека, т.к. id ∙> +, затем устанавливаем, что ‘(‘ <∙ id, поэтому больше символов со стека не снимаем |
| **+**id\*id)^id^id-id$ | ($ | id | Помещаем в стек +, т.к. ‘(‘ <∙ + |
| **id**\*id)^id^id-id$ | +($ | id | Помещаем в стек id, т.к. + <∙ id |
| **\***id)^id^id-id$ | id+($ | id id | Снимаем id со стека, т.к. id ∙> \*, затем устанавливаем, что + <∙ id, поэтому больше символов со стека не снимаем |
| **\***id)^id^id-id$ | +($ | id id | Помещаем в стек \*, т.к. + <∙ \* |
| **id**)^id^id-id$ | \*+($ | id id | Помещаем в стек id, т.к. \* <∙ id |
| **)**^id^id-id$ | id\*+($ | id id id | Снимаем id со стека, т.к. id ∙> ‘)‘, затем устанавливаем, что \* <∙ id, поэтому больше символов со стека не снимаем |
| **)**^id^id-id$ | \*+($ | id id id \* | Снимаем \* со стека, т.к. \* ∙> ‘)‘, затем устанавливаем, что + <∙ \*, поэтому больше символов со стека не снимаем |
| **)**^id^id-id$ | +($ | id id id \* + ( | Снимаем + со стека, т.к. + ∙> ‘)‘, затем устанавливаем, что ‘(‘ ‘)’, поэтому снимаем ‘(’ со стека. Далее устанавливаем, что $ <∙ ‘(‘, поэтому больше символов со стека не снимаем |
| **)**^id^id-id$ | $ | id id id \* + ( | Помещаем в стек ‘)’, т.к. $ <∙ ‘)’ |
| **^**id^id-id$ | )$ | id id id \* + ( ) | Снимаем ‘)’ со стека, т.к. ‘)’ ∙> ^, затем устанавливаем, что $ <∙ ‘)’, поэтому больше символов со стека не снимаем |
| **^**id^id-id$ | $ | id id id \* + ( ) | Помещаем в стек ^, т.к. $ <∙ ^ |
| **id**^id-id$ | ^$ | id id id \* + ( ) | Помещаем в стек id, т.к. ^ <∙ id |
| **^**id-id$ | id^$ | id id id \* + ( ) id | Снимаем id со стека, т.к. id ∙> ^, затем устанавливаем, что ^ <∙ id, поэтому больше символов со стека не снимаем |
| **^**id-id$ | ^$ | id id id \* + ( ) id | Помещаем в стек ^, т.к. ^ <∙ ^ |
| **id**-id$ | ^^$ | id id id \* + ( ) id | Помещаем в стек id, т.к. ^ <∙ id |
| **-**id$ | id^^$ | id id id \* + ( ) id id | Снимаем id со стека, т.к. id ∙> -, затем устанавливаем, что ^ <∙ id, поэтому больше символов со стека не снимаем |
| **-**id$ | ^^$ | id id id \* + ( ) id id ^ | Снимаем ^ со стека, т.к. ^ ∙> -, затем устанавливаем, что ^ <∙ ^, поэтому больше символов со стека не снимаем |
| **-**id$ | ^$ | id id id \* + ( ) id id ^ ^ | Снимаем ^ со стека, т.к. ^ ∙> -, затем устанавливаем, что $ <∙ ^, поэтому больше символов со стека не снимаем |
| **-**id$ | $ | id id id \* + ( ) id id ^ ^ | Помещаем в стек -, т.к. $ <∙ - |
| **id**$ | -$ | id id id \* + ( ) id id ^ ^ | Помещаем в стек id, т.к. - <∙ id |
| **$** | id-$ | id id id \* + ( ) id id ^ ^ id | Снимаем id со стека, т.к. id ∙> $, затем устанавливаем, что - <∙ id, поэтому больше символов со стека не снимаем |
| **$** | -$ | id id id \* + ( ) id id ^ ^ id - | Снимаем - со стека, т.к. - ∙> $, затем устанавливаем, что $ <∙ -, поэтому больше символов со стека не снимаем |
| **$** | $ | id id id \* + ( ) id id ^ ^ id - | Символ на вершине стека и текущий символ входного потока равны $, таким образом выражение успешно разобрано. |

Б) (id+id\*id)id^id-id

|  |  |  |  |
| --- | --- | --- | --- |
| Входной поток | Стек | Результат | Комментарий |
| **(**id+id\*id)id^id-id$ | $ |  | Помещаем в стек ‘(‘, т.к. $ <∙ ‘(‘ |
| **id**+id\*id)id^id-id$ | ($ |  | Помещаем в стек id, т.к. ‘(‘<∙ id |
| **+**id\*id)id^id-id$ | id($ | id | Снимаем id со стека, т.к. id ∙> +, затем устанавливаем, что ‘(‘ <∙ id, поэтому больше символов со стека не снимаем |
| **+**id\*id) id^id-id$ | ($ | id | Помещаем в стек +, т.к. ‘(‘ <∙ + |
| **id**\*id)id^id-id$ | +($ | id | Помещаем в стек id, т.к. + <∙ id |
| **\***id)id^id-id$ | id+($ | id id | Снимаем id со стека, т.к. id ∙> \*, затем устанавливаем, что + <∙ id, поэтому больше символов со стека не снимаем |
| **\***id)id^id-id$ | +($ | id id | Помещаем в стек \*, т.к. + <∙ \* |
| **id**)id^id-id$ | \*+($ | id id | Помещаем в стек id, т.к. \* <∙ id |
| **)**id^id-id$ | id\*+($ | id id id | Снимаем id со стека, т.к. id ∙> ‘)‘, затем устанавливаем, что \* <∙ id, поэтому больше символов со стека не снимаем |
| **)**id^id-id$ | \*+($ | id id id \* | Снимаем \* со стека, т.к. \* ∙> ‘)‘, затем устанавливаем, что + <∙ \*, поэтому больше символов со стека не снимаем |
| **)**id^id-id$ | +($ | id id id \* + ( | Снимаем + со стека, т.к. + ∙> ‘)‘, затем устанавливаем, что ‘(‘ ‘)’, поэтому снимаем ‘(’ со стека. Далее устанавливаем, что $ <∙ ‘(‘, поэтому больше символов со стека не снимаем |
| **)**id^id-id$ | $ | id id id \* + ( | Помещаем в стек ‘)’, т.к. $ <∙ ‘)’ |
| **id**^id-id$ | )$ | id id id \* + ( ) | Обнаруживаем пустую ячейку в таблице приоритетов. (Ячейка [‘)’,id]. Анализ завершился с ошибкой. |

B) (id+idid)^id^id-id

|  |  |  |  |
| --- | --- | --- | --- |
| Входной поток | Стек | Результат | Комментарий |
| **(**id+idid)^id^id-id$ | $ |  | Помещаем в стек ‘(‘, т.к. $ <∙ ‘(‘ |
| **id**+idid)^id^id-id$ | ($ |  | Помещаем в стек id, т.к. ‘(‘<∙ id |
| **+**idid)^id^id-id$ | id($ | id | Снимаем id со стека, т.к. id ∙> +, затем устанавливаем, что ‘(‘ <∙ id, поэтому больше символов со стека не снимаем |
| **+**idid)^id^id-id$ | ($ | id | Помещаем в стек +, т.к. ‘(‘ <∙ + |
| **id**id)^id^id-id$ | +($ | id | Помещаем в стек id, т.к. + <∙ id |
| **id**)^id^id-id$ | id+($ | id id | Обнаруживаем пустую ячейку в таблице приоритетов. (Ячейка [id, id]. Анализ завершился с ошибкой. |

# Задача 5. «Синтаксически управляемая трансляция»

***Теоретическая часть:***

Промежуточный код может иметь три типа представления:

* Синтаксические деревья
* Постфиксная запись
* Трехадресный код

Далее подробнее о каждом типе представления. Постфиксная запись в данной задаче рассматриваться не будет.

**Синтаксические деревья**

Синтаксические деревья бывают двух видов: **собственно синтаксическое дерево** и **даг**.Синтаксическое дерево изображает естественную иерархическую структуру исходной программы. В свою очередь, даг дает ту же информацию, но в более компактном виде (одинаковые подвыражения в нем объединены).

Приведем пример графического представления синтаксического дерева и дага для инструкции присваивания ***a := b \* -c + b \* -c***. Данный пример не описывает всю специфику представления промежуточного кода в виде синтаксических деревьев. Более подробный пример будет приведен в разборе задачи.

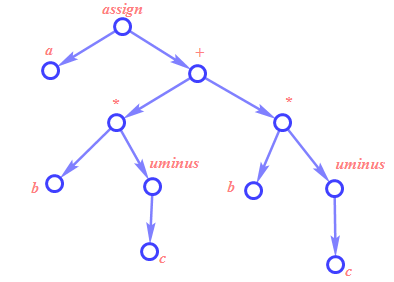


Рис. 5.1. Синтаксическое дерево для инструкции a := b \* -c + b \* -c.

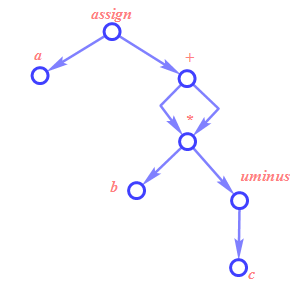


Рис. 5.2. Даг для инструкции a := b \* -c + b \* -c.

Опишем примерные семантические правила, которые необходимы для построения промежуточного кода в виде синтаксического дерева для грамматики арифметических выражений.

|  |  |
| --- | --- |
| Продукция | Семантическое правило |
| S -> **id** := E | S.nptr := mknode(‘assign’, mkleaf(**id**, **id**.place), E.nptr) |
| E -> E1 + E2 | E.nptr := mknode(‘+’, E1.nptr, E2.nptr) |
| E -> E1 \* E2 | E.nptr := mknode(‘\*’, E1.nptr, E2.nptr) |
| E -> - E1 | E.nptr := mkunode(‘uminus’, E1.nptr) |
| E -> (E1) | E.nptr := E1.nptr |
| E -> **id** | E.nptr := mkleaf(**id**, **id**.place) |

Поле нетерминального символа ***nptr*** обозначает указатель на узел некоторый дерева.

Операция ***mknode(name, ptr1, ptr2)*** позволяет создать узел дерева, который имеет название name и ссылается на узлы, обозначаемые указателями *ptr1* и *ptr2* соответственно.

Операция ***mkunode(name, ptr)*** позволяет создать узел дерева, который имеет название *name* и ссылается только на узел, обозначаемый указателем *ptr*.

Операция ***mkleaf(name, value)*** позволяет создать лист дерева, который имеет название name и значение value.

Построение дага будет осуществлено в том случае, если модифицировать правила, описанные выше, таким образом, что операции ***mknode(name, ptr1, ptr2)*** и ***mkunode(name, ptr)*** будут не сразу создавать новый узел дерева, а проверять, существует ли такой узел, и если существует, то возвращать указатель на него.

Приведем возможный вариант реализации структуры хранения для синтаксического дерева в компиляторе.

Первый вариант реализации в виде записей с указателями:

assign

id

a

+

\*

id

b

uminus

id

c

\*

id

b

uminus

id

c

Второй вариант реализации в виде массива:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | id | b |  |
| 1 | id | c |  |
| 2 | uminus | 1 |  |
| 3 | \* | 0 | 2 |
| 4 | id | b |  |
| 5 | id | c |  |
| 6 | uminus | 5 |  |
| 7 | \* | 4 | 6 |
| 8 | + | 3 | 7 |
| 9 | id | a |  |
| 10 | assign | 9 | 8 |

Приведем так же аналогичные варианты хранения дага.

Первый вариант:

assign

id

a

+

\*

id

b

uminus

id

c

Второй вариант :

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | id | b |  |
| 1 | id | c |  |
| 2 | uminus | 1 |  |
| 3 | \* | 0 | 2 |
| 4 | + | 3 | 3 |
| 5 | id | a |  |
| 6 | assign | 5 | 4 |

Как видно из размера структур данных, даг занимает меньше памяти, чем синтаксическое дерево.

**Трехадресный код**

Трехадресный код представляет собой последовательность инструкций вида x := y op z, где x, y, z – имена, константы, временные переменные, генерируемые компилятором; op означает некоторый оператор, например, арифметический оператор для работы с логическими значениями. Заметим, что не разрешены никакие встроенные арифметические выражения, и в правой части инструкции имеется только один оператор. Следовательно, выражение исходного языка наподобие x + y \* z может быть транслировано в следующую последовательность.

t1 := y \* z

t2 := x + t1

Здесь t1 и t2 – сгенерированные компилятором временные имена. Такое разложение сложных арифметических выражений и вложенных инструкций потока управления делает трехадресный код подходящим для генерации целевого кода и оптимизации.

Трехадресный код является линеаризованным представлением синтаксического дерева или дага, в котором внутренним узлам графа соответствуют явные имена. Покажем, как при помощи трехадресного кода представить синтаксическое дерево и даг, представленные на рис. 5.1. и 5.2., соответственно.

Код для синтаксического дерева:

t1 := - с

t2 := b \* t1

t3 := - c

t4 := b \* t3

a := t2 + t4

Код для дага:

t1 := - c

t2 := b \* t

t3 := t2 + t2

a := t3

**Типы трехадресных инструкций**

Трехадресные инструкции похожи на ассемблерный код.

Ниже приведен список основных трехадресных инструкций, предлагаемых для использования в данной задаче:

1. Инструкция присваивания вида x := y op z, где op – бинарная арифметическая операция или логическая операция.
2. Инструкция присваивания вида x := op y, где op – унарная операция. Основные унарные операции включают унарный минус, логическое отрицание, операторы сдвига и операторы преобразования, например, преобразуют число с фиксированной точкой в число с плавающей точкой.
3. Инструкции копирования вида x := y, в которых значение y присваивается x.
4. Безусловный переход goto L. После этой инструкции будет выполнена трехадресная инструкция с меткой L.
5. Условный переход типа if x relop y goto L. Эта инструкция применяет оператор отношения relop (<, >, = и т.п.) к x и y, и следующей выполняется инструкция с меткой L, если соотношение x relop y верно. В противном случае выполняется следующая за условным переходом инструкция.
6. Инструкции param x и call p, n для вызова процедур и return y для возврата из них, где y обозначает необязательное возвращаемое значение. Обычно они используются в виду следующей последовательности трехадресных инструкций.

param x1,

param x2,

param x3,

…,

param xn

call p, n

Данная последовательность генерируется в качестве части вызова процедуры p(x1, x2, …, xn). В инструкции call p, n целое число n, указывающее количество действительных параметров, не является излишним в силу того, что вызовы могут быть вложенными.

1. Индексированные присвоения типа x := y[i] и x[i] := y. Первая инструкция присваивает x значение, находящееся в i-й ячейке памяти по отношению к y. Инструкция x[i] := y заносит в i-ю ячейку памяти по отношению к x значение y. В обеих инструкциях x, y и i ссылаются на объекты данных.
2. Присваивание адресов и указателей вида x := &y, x := \*y и \*x := y. Первая инструкция устанавливает значение x равным положению y в памяти. Предположительно, y представляет собой имя, возможно временное, обозначающее выражение с l-значением типа A[i,j], а x – имя указателя или временное имя. Таким образом, r-значение x становится равным содержимому этой ячейки. И наконец, инструкция, \*x := y устанавливает r-значение объекта, указываемого x, равным r-значению y.

Опишем примерные семантические правила, которые необходимы для построения промежуточного кода в виде трехадресного кода для грамматики арифметических выражений.

|  |  |
| --- | --- |
| Продукция | Семантическое правило |
| S -> **id** := E | S.code := E.code || gen(id.place ‘:=’ E.place) |
| E -> E1 + E2 | E.place := newtemp;  E.code := E1.code || E2.code || gen(E.place ‘:=’ E1.place ‘+’ E2.place) |
| E -> E1 \* E2 | E.place := newtemp;  E.code := E1.code || E2.code || gen(E.place ‘:=’ E1.place ‘\*’ E2.place) |
| E -> - E1 | E.place := newtemp;  E.code := E1.code || gen(E.place ‘:=’‘uminus’ E1.place) |
| E -> (E1) | E.place := E1.place;  E.code := E1.code |
| E -> **id** | E.place := id.placel  E.code := ‘’ |

Поле нетерминального символа ***place*** обозначает имя, которое хранит нетерминал.

Поле нетерминального символа ***code*** обозначает последовательность трехадресных инструкций, вычисляющих E.

Операция ***newtemp*** последовательно возвращает имена временных переменных t1­, t2, …, .

Операция ***||*** позволяет добавить к последовательности трехадресных инструкций новые, сгенерированные при срабатывании данного правила.

Операция ***gen(instruction)*** позволяет генерировать трехадресную инструкцию, которая является конкатенацией строк, записанных в параметрах.

**Реализация трехадресных инструкций**

Трехадресные инструкции представляют собой абстрактный вид промежуточного кода. В компиляторе эти инструкции могут быть реализованы как записи с полями для операторов и операндов. Три возможных представления – это *четверки*, *тройки* и *косвенные тройки*.

**Четверки.**

Четверка (quadruple) представляет собой запись с четырьмя полями, которые назовем op, arg1, arg2, result. Поле op содержит внутренний код оператора. Трехадресная инструкция x := y op z представляется размещением y в arg1, z – в arg2 и x – в result. Инструкции с унарным оператором наподобие x := -y или x := y не используют arg2. Операторы типа param не используют ни arg2, ни result. Условные и безусловные переходы помещают в result целевую метку.

Покажем представление в виде четверок операции присваивания a := b \* -c + b \* -c.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | op | arg1 | arg2 | result |
| (0) | uminus | c |  | t1 |
| (1) | \* | b | t1 | t2 |
| (2) | uminus | c |  | t3 |
| (3) | \* | b | t3 | t4 |
| (4) | + | t2 | t4 | t5 |
| (5) | assign | t5 |  | a |

**Тройки.**

Для того, чтобы избежать вставки временных имен в таблицу символов, мы можем ссылаться на временные значения по номеру инструкции, которая вычисляет значение, соответствующее этому имени. Если мы поступим таким образом, трехадресные инструкции можно будет представить записями только с тремя полями: op, arg1 и arg2. Поля arg1 и arg2 для аргументов op представляют собой либо указатели в таблицу символов, либо указатели на тройки. Поскольку здесь используется три поля, этот вид промежуточного кода известен как тройки (triple). За исключением представления имен, определенных в программе, тройки соответствуют представлению синтаксического дерева или графа в виде массива вершин.

Покажем представление в виде троек операции присваивания a := b \* -c + b \* -c.

|  |  |  |  |
| --- | --- | --- | --- |
|  | op | arg1 | arg2 |
| (0) | uminus | c |  |
| (1) | \* | b | (0) |
| (2) | uminus | c |  |
| (3) | \* | b | (2) |
| (4) | + | (1) | (3) |
| (5) | assign | а | (4) |

**Косвенные тройки.**

Еще одно представление трехадресного кода состоит в использовании списка указателей на тройки вместо списка самих троек. Естественно, такая реализация названа косвенными тройками (indirect triples).

Ниже приведена таблица, содержащая непосредственно тройки.

|  |  |  |  |
| --- | --- | --- | --- |
|  | op | arg1 | arg2 |
| (14) | uminus | c |  |
| (15) | \* | b | (14) |
| (16) | uminus | c |  |
| (17) | \* | b | (16) |
| (18) | + | (15) | (17) |
| (19) | assign | а | (18) |

Далее таблица, содержащая ссылки на эти тройки

|  |  |
| --- | --- |
|  | statement |
| (0) | (14) |
| (1) | (15) |
| (2) | (16) |
| (3) | (17) |
| (4) | (18) |
| (5) | (19) |

**Оптимизация**

Существует ряд способов, которыми компилятор может улучшить программу без изменения вычисляемой функции. Устранение общих подвыражений, размножение копий, удаление недоступного кода и дублирование констант – вот основные примеры таких преобразований, сохраняющих функции.

**Общие подвыражения.**

Выражение E называется **общим подвыражением**, если E было ранее вычислено и значения переменных в E с того времени не изменились. Повторного вычисления можно избежать, если использовать ранее вычисленные значения.

Приведем фрагмент кода с подвыражениями, которые будут устранены в результате оптимизации.

t6 := 4 \* i

x := a[t6]

t7 := 4 \* i

t8 := 4 \* j

t9 := a[t8]

a[t7] := t9

t10 := 4 \* j

a[t10] := x

t6 := 4 \* i

x := a[t6]

t8 := 4 \* j

t9 := a[t8]

a[t6] := t9

a[t8] := x

**Распространение копий.**

Идея преобразования распространения копий заключается в использовании после присваивания f := g переменной g вместо f везде, где это только возможно.

Сам по себе этот прием не несет в себе улучшения качества кода, однако совместно с удалением бесполезного кода, которое будет описано ниже.

Приведем фрагмент кода, в котором проведем распространение копий.

x := t3

a[t2] := t5

a[t4] := t3

x := t3

a[t2] := t5

a[t4] := x

**Удаление бесполезного кода.**

Переменная в некоторой точке программы считается “живой”, или активной, если ее значение будет использовано в программе в последующем; в противном случае она считается “мертвой”. Очередная оптимизация касается “мертвого”, или бесполезного кода, т.е. кода, вычисляющего значения, которые никогда не будут использованы.

Таким образом, данная оптимизация совместно с распространением копий превращает инструкции копирования в мертвый код, который в последствие удаляется.

Приведем фрагмент кода, в котором удалим бесполезный код.

x := t3

a[t2] := t5

a[t4] := t3

a[t2] := t5

a[t4] := t3

**Построение формальной КС-грамматики, а так же ее атрибутивной грамматики с семантическими действиями более подробно описано в лабораторном практикуме по курсу.**

***Условие:***

1. Построить промежуточный код в указанном формате (С?) для фрагмента исходного кода (F?) на языке C.
2. Описать типологию команд промежуточного кода. Она должна соответствовать той, что описана выше в теории.
3. Над полученным промежуточным кодом провести оптимизацию. Записать протокол оптимизирующих действий.
4. Полученный после оптимизации код записать в другом формате (-> C?).
5. По исходному фрагменту кода построить формальную КС-грамматику, которая задавала бы схожие фрагменты.
6. На основе полученной КС-грамматики построить атрибутивную грамматику, в которой описываются действия по проверке семантических ошибок и по генерации промежуточного кода.

## Варианты:

|  |  |
| --- | --- |
| № | Формулировка варианта задания |
|  | F1,C1 -> C3 |
|  | F1,C2 -> C3 |
|  | F1,C3 -> C1 |
|  | F1,C4 -> C1 |
|  | F1,C5 -> C1 |
|  | F2,C1 -> C3 |
|  | F2,C2 -> C3 |
|  | F2,C3 -> C1 |
|  | F2,C4 -> C1 |
|  | F2,C5 -> C1 |
|  | F3,C1 -> C3 |
|  | F3,C2 -> C3 |
|  | F3,C3 -> C1 |
|  | F3,C4 -> C1 |
|  | F3,C5 -> C1 |
|  | F4,C1 -> C3 |
|  | F4,C2 -> C3 |
|  | F4,C3 -> C1 |
|  | F4,C4 -> C1 |
|  | F4,C5 -> C1 |
|  | F5,C1 -> C3 |
|  | F5,C2 -> C3 |
|  | F5,C3 -> C1 |
|  | F5,C4 -> C1 |
|  | F5,C5 -> C1 |
|  | F6,C1 -> C3 |
|  | F6,C2 -> C3 |
|  | F6,C3 -> C1 |
|  | F6,C4 -> C1 |
|  | F6,C5 -> C1 |

|  |  |
| --- | --- |
| F1 | int a, b, c = 1, d = c \* 3;  char \* s = “строка символов”;  if (s[c + d] == 0) {  a = c + 1;  b = c + 2;  } else {  a = d + 1;  b = d + 2;  }  printf(“Ответ: %d”, a + b); |
| F2 | int f(int n) { return n \* 3 + 1;}  int main(void) {  int d = f (4);  return f(5) + d;  } |
| F3 | for (int i = 0; i < 12; i++) {  scanf(“%d”, f);  int d = i \* f \* 3;  printf(“%d”, d);  } |
| F4 | int a, b;  scanf(“%d”, a);  switch(a) {  case 1: b = a + 1;  case 2: b = a + 2; break;  default: b = 0;  }  exit(b); |
| F5 | struct {int a; char b} s, f;  s.a = 1;  s.b = ‘3’;  strcpy(&f, &s, sizeof(s));  printf(“%d - %d”, f.a, f.b); |
| F6 | char[] str = “string”;  int a, b, c;  a = 3;  if (str[a] == ‘\n’) {  b = str[a-1]; c = 0;  } else {  b = str[a+1]; c = 1;  }  str[b] = 0;  printf (“%s”, str); |

|  |  |
| --- | --- |
| C1 | Синтаксическое дерево |
| C2 | Даг |
| C3 | Трехадресный код. Четверки |
| C4 | Трехадресный код. Тройки |
| C5 | Трехадресный код. Косвенные тройки |

***Пример решения задачи:***

1. Фрагмент кода

|  |
| --- |
| char[] str = “something”;  int f = 4;  char[] useless = “bad string”;  int x = f;  int j;  for (int i = 0; i < strlen(str); i++) {  scanf(“%d”, &j);  int k = x \* j;  printf(“%c - %d”, str[x \* j], k);  } |

1. Построение промежуточного кода.
   1. В виде синтаксического дерева.

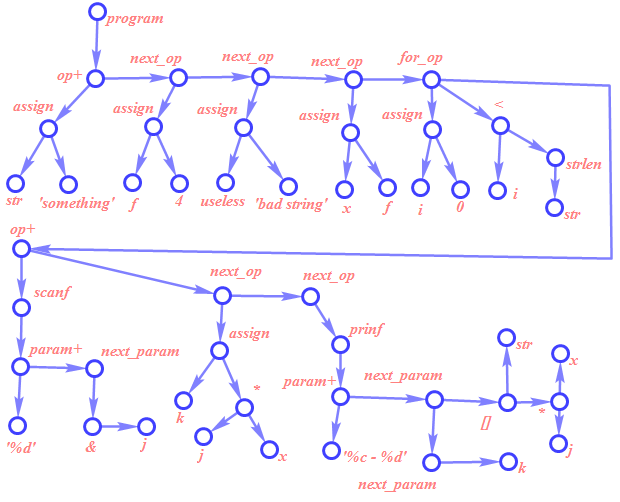


Рис. 5.3. Представление промежуточного кода в виде синтаксического дерева

* 1. В виде дага.

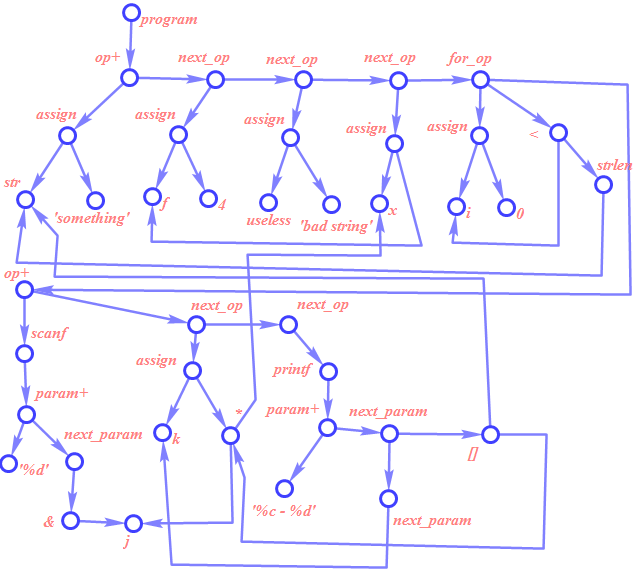


Рис. 5.4. Представление промежуточного кода в виде дага

* 1. В виде четверок.

Ячейки памяти для констант

|  |  |
| --- | --- |
| Номер ячейки | Значение |
| D01 | “something” |
| D02 | “bad string” |
| D03 | “%d” |
| D04 | “%c - %d” |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | op | arg1 | arg2 | result |
| (0) | assign | D01 |  | str |
| (1) | assign | 4 |  | f |
| (2) | assign | D02 |  | useless |
| (3) | assign | f |  | x |
| (4) | assign | 0 |  | i |
| (5) | param | str |  |  |
| (6) | call | strlen | 1 | t1 |
| (7) | < | i | t1 | t2 |
| (8) | if | t2 | (10) |  |
| (9) | goto | (25) |  |  |
| (10) | param | D03 |  |  |
| (11) | & | j |  | t3 |
| (12) | param | t3 |  |  |
| (13) | call | scanf | 2 |  |
| (14) | \* | j | x | t4 |
| (15) | assign | t4 |  | k |
| (16) | param | D04 |  |  |
| (17) | \* | j | x | t5 |
| (18) | [] | str | t5 | t6 |
| (19) | param | t6 |  |  |
| (20) | param | k |  |  |
| (21) | call | printf | 3 |  |
| (22) | + | i | 1 | t­7 |
| (23) | assign | t7 |  | i |
| (24) | goto | (5) |  |  |
| (25) |  |  |  |  |

* 1. В виде троек.

Ячейки памяти для констант

|  |  |
| --- | --- |
| Номер ячейки | Значение |
| D01 | “something” |
| D02 | “bad string” |
| D03 | “%d” |
| D04 | “%c - %d” |

|  |  |  |  |
| --- | --- | --- | --- |
|  | op | arg1 | arg2 |
| (0) | assign | str | D01 |
| (1) | assign | f | 4 |
| (2) | assign | useless | D02 |
| (3) | assign | x | f |
| (4) | assign | i | 0 |
| (5) | param | str |  |
| (6) | call | strlen | 1 |
| (7) | < | i | (6) |
| (8) | if | (7) | (10) |
| (9) | goto | (25) |  |
| (10) | param | D03 |  |
| (11) | & | j |  |
| (12) | param | (11) |  |
| (13) | call | scanf | 2 |
| (14) | \* | j | x |
| (15) | assign | k | (14) |
| (16) | param | D04 |  |
| (17) | \* | j | x |
| (18) | [] | str | (17) |
| (19) | param | (18) |  |
| (20) | param | k |  |
| (21) | call | printf | 3 |
| (22) | + | i | 1 |
| (23) | assign | i | (22) |
| (24) | goto | (5) |  |
| (25) |  |  |  |

* 1. В виде косвенных троек.

Ячейки памяти для констант

|  |  |
| --- | --- |
| Номер ячейки | Значение |
| D01 | “something” |
| D02 | “bad string” |
| D03 | “%d” |
| D04 | “%c - %d” |

Непосредственно тройки.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Op | arg1 | arg2 |
| (30) | assign | str | D01 |
| (31) | assign | f | 4 |
| (32) | assign | useless | D02 |
| (33) | assign | x | f |
| (34) | assign | i | 0 |
| (35) | param | str |  |
| (36) | call | strlen | 1 |
| (37) | < | i | (36) |
| (38) | if | (37) | (40) |
| (39) | goto | (55) |  |
| (40) | param | D03 |  |
| (41) | & | j |  |
| (42) | param | (41) |  |
| (43) | call | scanf | 2 |
| (44) | \* | j | x |
| (45) | assign | k | (44) |
| (46) | param | D04 |  |
| (47) | \* | j | x |
| (48) | [] | str | (47) |
| (49) | param | (48) |  |
| (50) | param | k |  |
| (51) | call | printf | 3 |
| (52) | + | i | 1 |
| (53) | assign | i | (52) |
| (54) | goto | (35) |  |
| (55) |  |  |  |

Таблица, содержащая ссылки на тройки.

|  |  |
| --- | --- |
|  | statement |
| (0) | (30) |
| (1) | (31) |
| (2) | (32) |
| (3) | (33) |
| (4) | (34) |
| (5) | (35) |
| (6) | (36) |
| (7) | (37) |
| (8) | (38) |
| (9) | (39) |
| (10) | (40) |
| (11) | (41) |
| (12) | (42) |
| (13) | (43) |
| (14) | (44) |
| (15) | (45) |
| (16) | (46) |
| (17) | (47) |
| (18) | (48) |
| (19) | (49) |
| (20) | (50) |
| (21) | (51) |
| (22) | (52) |
| (23) | (53) |
| (24) | (54) |
| (25) | (55) |

1. Типология команд промежуточного кода.
   1. Для синтаксического дерева.

* program – программа. Содержит список операторов программы.
* op+ - список операторов программы. Содержит оператор и ссылку на следующий.
* next\_op – ссылка на следующий оператор списка. Содержит оператор и ссылку на следующий (ссылка может отсутствовать).
* assign – оператор присваивания. Содержит объект, которому присваивается значение и значение, которое присваивается.
* for\_op – оператор итерационного цикла. Состоит из оператора присваивания, условного оператора и списка операторов тела цикла.
* < - логическая операция «меньше». Состоит из двух объектов, которые сравниваются этим оператором.
* strlen – оператор нахождения длины строки. Содержит строку, длину которой находит.
* scanf – оператор ввода. Содержит список параметров.
* param+ - список параметров функции. Содержит параметр и ссылку на следующий параметр.
* next\_param - ссылка на следующий параметр функции. Содержит параметр функции и ссылку на следующий (ссылка может отсутствовать).
* & - оператор взятия адреса. Содержит объект, адрес которого берется.
* \* - оператор умножения. Содержит умножаемые числа.
* printf – оператор вывода. Содержит список параметров.
* [] – оператор обращения к элементу массива по индексу. Содержит массив, к которому осуществляется обращение и индекс запрашиваемого элемента.
  1. Для дага.

Номенклатура операторов для дага ничем не отличается от описанной выше для синтаксического дерева.

* 1. Для четверок.
* assign – оператор присваивания. Содержит объект, которому присваивается значение и значение, которое присваивается.
* param – оператор занесения параметра в стек. Содержит параметр, заносимый в стек.
* call – вызов функции. Содержит имя вызываемой функции и число параметров, извлекаемых из стека.
* < - логическая операция «меньше». Состоит из двух объектов, которые сравниваются этим оператором.
* if – оператор условного перехода. Содержит условие перехода и метку, по которой осуществляется переход. В случае выполнения условия осуществляется переход по метке, в противном случае осуществляется переход к следующей инструкции.
* goto – оператор безусловного перехода. Содержит метку, по которой осуществляется переход.
* & - оператор взятия адреса. Содержит объект, адрес которого берется.
* \* - оператор умножения. Содержит умножаемые числа.
* [] – оператор обращения к элементу массива по индексу. Содержит массив, к которому осуществляется обращение и индекс запрашиваемого элемента.
* + - оператор сложения. Содержит складываемые числа.
  1. Для троек.

Номенклатура операторов для троек ничем не отличается от описанной выше для четверок.

* 1. Для косвенных троек.

Номенклатура операторов для косвенных троек ничем не отличается от описанной выше для четверок и троек.

1. Оптимизация промежуточного кода.

Для наглядности оптимизацию проведем над исходным кодом, затем покажем, как она отразилась на каждом виде промежуточного кода.

* Найдем общие подвыражения и заменим их, где это возможно.

char[] str = “something”;

int f = 4;

char[] useless = “bad string”;

int x = f;

int j;

for (int i = 0; i < strlen(str); i++) {

scanf(“%d”, &j);

int k = x \* j;

printf(“%c - %d”, str[x \* j], k);

}

char[] str = “something”;

int f = 4;

char[] useless = “bad string”;

int x = f;

int j;

for (int i = 0; i < strlen(str); i++) {

scanf(“%d”, &j);

int k = x \* j;

printf(“%c - %d”, str[k], k);

}

* Проведем распространение копий для последующей оптимизации путем удаления бесполезного кода

char[] str = “something”;

int f = 4;

char[] useless = “bad string”;

int x = f;

int j;

for (int i = 0; i < strlen(str); i++) {

scanf(“%d”, &j);

int k = x \* j;

printf(“%c - %d”, str[k], k);

}

char[] str = “something”;

int f = 4;

char[] useless = “bad string”;

int x = f;

int j;

for (int i = 0; i < strlen(str); i++) {

scanf(“%d”, &j);

int k = f \* j;

printf(“%c - %d”, str[k], k);

}

* Удалим бесполезный код

char[] str = “something”;

int f = 4;

char[] useless = “bad string”;

int x = f;

int j;

for (int i = 0; i < strlen(str); i++) {

scanf(“%d”, &j);

int k = f \* j;

printf(“%c - %d”, str[k], k);

}

char[] str = “something”;

int f = 4;

int j;

for (int i = 0; i < strlen(str); i++) {

scanf(“%d”, &j);

int k = f \* j;

printf(“%c - %d”, str[k], k);

}

Запишем полученный оптимизированный промежуточный код.

* 1. В виде синтаксического дерева.

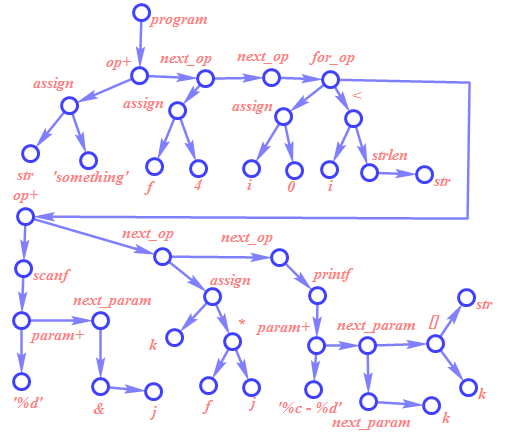


Рис. 5.5. Представление оптимизированного промежуточного кода в виде синтаксического дерева

* 1. В виде дага.

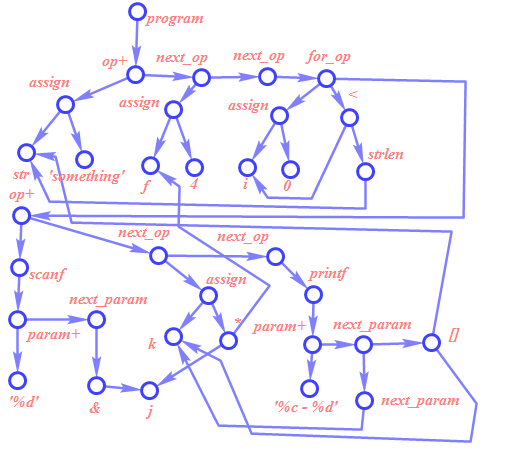


Рис. 5.6. Представление оптимизированного промежуточного кода в виде дага

* 1. В виде четверок.

Ячейки памяти для констант

|  |  |
| --- | --- |
| Номер ячейки | Значение |
| D01 | “something” |
| D02 | “%d” |
| D03 | “%c - %d” |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Op | arg1 | arg2 | result |
| (0) | assign | D01 |  | str |
| (1) | assign | 4 |  | f |
| (2) | assign | 0 |  | i |
| (3) | param | str |  |  |
| (4) | call | strlen | 1 | t1 |
| (5) | < | i | t1 | t2 |
| (6) | if | t2 | (8) |  |
| (7) | goto | (22) |  |  |
| (8) | param | D02 |  |  |
| (9) | & | j |  | t3 |
| (10) | param | t3 |  |  |
| (11) | call | scanf | 2 |  |
| (12) | \* | j | f | t4 |
| (13) | assign | t4 |  | k |
| (14) | param | D03 |  |  |
| (15) | [] | str | k | t5 |
| (16) | param | t5 |  |  |
| (17) | param | k |  |  |
| (18) | call | printf | 3 |  |
| (19) | + | i | 1 | t­6 |
| (20) | assign | t6 |  | i |
| (21) | goto | (3) |  |  |
| (22) |  |  |  |  |

* 1. В виде троек.

Ячейки памяти для констант

|  |  |
| --- | --- |
| Номер ячейки | Значение |
| D01 | “something” |
| D02 | “%d” |
| D03 | “%c - %d” |

|  |  |  |  |
| --- | --- | --- | --- |
|  | op | arg1 | arg2 |
| (0) | assign | str | D01 |
| (1) | assign | f | 4 |
| (2) | assign | i | 0 |
| (3) | param | str |  |
| (4) | call | strlen | 1 |
| (5) | < | i | (4) |
| (6) | if | (5) | (8) |
| (7) | goto | (22) |  |
| (8) | param | D02 |  |
| (9) | & | j |  |
| (10) | param | (9) |  |
| (11) | call | scanf | 2 |
| (12) | \* | j | f |
| (13) | assign | k | (12) |
| (14) | param | D03 |  |
| (15) | [] | str | k |
| (16) | param | (15) |  |
| (17) | param | k |  |
| (18) | call | printf | 3 |
| (19) | + | i | 1 |
| (20) | assign | i | (19) |
| (21) | goto | (3) |  |
| (22) |  |  |  |

* 1. В виде косвенных троек.

Ячейки памяти для констант

|  |  |
| --- | --- |
| Номер ячейки | Значение |
| D01 | “something” |
| D02 | “%d” |
| D03 | “%c - %d” |

Непосредственно тройки.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Op | arg1 | arg2 |
| (30) | assign | str | D01 |
| (31) | assign | f | 4 |
| (32) | assign | i | 0 |
| (33) | param | str |  |
| (34) | call | strlen | 1 |
| (35) | < | i | (34) |
| (36) | If | (35) | (38) |
| (37) | goto | (52) |  |
| (38) | param | D02 |  |
| (39) | & | j |  |
| (40) | param | (39) |  |
| (41) | call | scanf | 2 |
| (42) | \* | j | f |
| (43) | assign | k | (42) |
| (44) | param | D03 |  |
| (45) | [] | str | k |
| (46) | param | (45) |  |
| (47) | param | k |  |
| (48) | call | printf | 3 |
| (49) | + | i | 1 |
| (50) | assign | i | (49) |
| (51) | goto | (33) |  |
| (52) |  |  |  |

Таблица, содержащая ссылки на тройки.

|  |  |
| --- | --- |
|  | statement |
| (0) | (30) |
| (1) | (31) |
| (2) | (32) |
| (3) | (33) |
| (4) | (34) |
| (5) | (35) |
| (6) | (36) |
| (7) | (37) |
| (8) | (38) |
| (9) | (39) |
| (10) | (40) |
| (11) | (41) |
| (12) | (42) |
| (13) | (43) |
| (14) | (44) |
| (15) | (45) |
| (16) | (46) |
| (17) | (47) |
| (18) | (48) |
| (19) | (49) |
| (20) | (50) |
| (21) | (51) |
| (22) | (52) |

1. Запись оптимизированного кода в другом формате.

* Синтаксическое дерево -> даг
* Синтаксическое дерево -> четверки
* Синтаксическое дерево -> тройки
* Синтаксическое дерево -> косвенные тройки
* Даг -> синтаксическое дерево
* Даг -> четверки
* Даг -> тройки
* Даг -> косвенные тройки
* Четверки -> тройки
* Четверки -> косвенные тройки
* Тройки -> четверки
* Тройки -> косвенные тройки
* Косвенные тройки -> четверки
* Косвенные тройки -> тройки

При проведении описанных выше преобразований промежуточный код будет выглядеть точно так же, как и в п.3. задачи.

* Четверки -> синтаксическое дерево

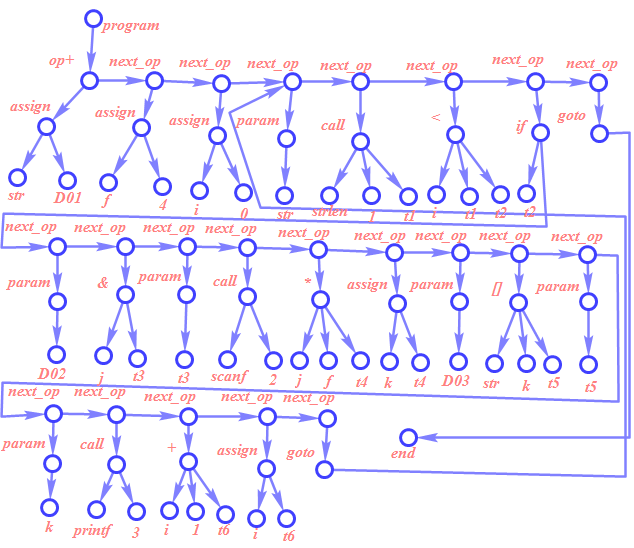


Рис. 5.7. Представление оптимизированного промежуточного кода в виде синтаксического дерева, преобразованного из четверок.

* Тройки -> синтаксическое дерево

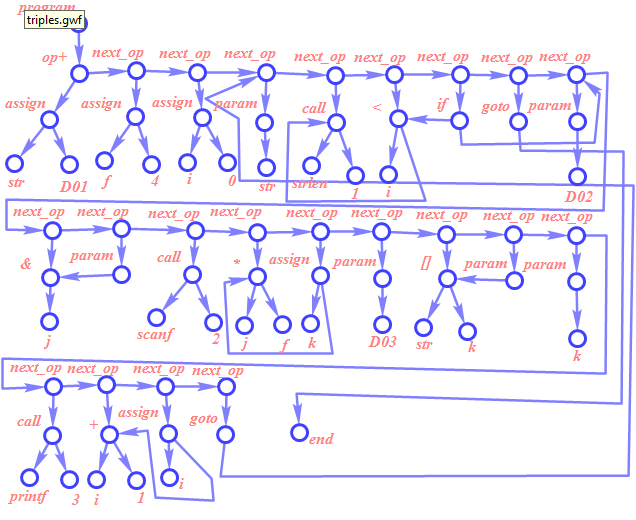


Рис. 5.8. Представление оптимизированного промежуточного кода в виде синтаксического дерева, преобразованного из троек.

* Косвенные тройки -> синтаксическое дерево

Вид данного синтаксического дерева совпадает с тем, что изображено на рис. 5.8.

* Четверки -> даг

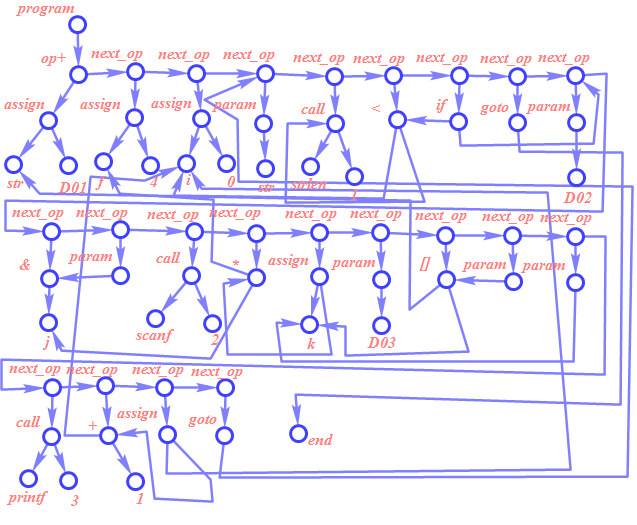


Рис. 5.8. Представление оптимизированного промежуточного кода в виде дага, преобразованного из четверок.

* Тройки -> даг
* Косвенные тройки -> даг

Вид дага совпадает как для троек, так и для косвенных троек в силу специфики собственно дага и его представления в трехадресном коде.

1. Построим формальную КС-грамматику, которая задает похожие фрагменты исходного кода, при помощи среды ***antlrworks***.

**grammar** Task5Grammar;

**options** {

language = Java;

}

*ID* : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\*

;

*DIGIT* : '0'..'9'+

;

*STRING* : '"' ~'"'\* '"'

;

*WS* : ( ' '

| '\t'

| '\r'

| '\n'

) {$channel=HIDDEN;}

;

type : 'int' | 'char[]'

;

assign

: *ID* '=' (expression)

;

assign\_operator

: assign ';'

;

expression

: *ID* |

*STRING* |

*DIGIT* |

arifmetic\_expression |

incrementation |

function\_call |

array\_element |

getting\_address

;

arifmetic\_operand

: *ID* |

*DIGIT*

;

arifmetic\_expression

: arifmetic\_operand arifmetic\_sign arifmetic\_operand

;

arifmetic\_sign

: ('+' | '\*')

;

incrementation

: *ID* '++'

;

array\_element

: *ID* '[' expression ']'

;

getting\_address

: '&' *ID*

;

function\_call

: *ID* '(' parametres ')'

;

parametres

: (expression (',' expression)\*)?

;

variable\_declaration

: type (*ID* | assign) (',' (*ID* | assign))\* ';'

;

for\_operator

: 'for' '(' type? assign ';' logical\_condition ';' expression ')' '{'

operator+

'}'

;

function\_call\_operator

: function\_call ';'

;

logical\_condition

: expression logical\_link expression

;

logical\_link

: '==' | '!=' | '>' | '<' | '>=' | '<='

;

operator : variable\_declaration | assign\_operator | for\_operator | function\_call\_operator

;

program : (operator)+

;

1. Построение атрибутивной грамматики с семантическими проверками и генерацией целевого кода в указанном формате.
   1. Для синтаксического дерева.

**grammar** Task5Grammar;

**options** {

language = Java;

}

@members {

**protected** NamesTable names = **new** NamesTable();

**protected** ArrayList<String> errors = **new** ArrayList<String>();

**protected** ArrayList<String> instructions = **new** ArrayList<String>();

**protected** ArrayList<String> constants = **new** ArrayList<String>();

**protected** **int** instructionNumber = 0;

**protected** String program = **""**;

**public** **static** **void** main(String[] args) **throws** Exception {

Task5GrammarLexer lex = **new** Task5GrammarLexer(**new** ANTLRFileStream(**"test.txt"**));

Task5GrammarParser parser = **new** Task5GrammarParser(**new** CommonTokenStream(lex));

parser.program();

**if** (! parser.errors.isEmpty()) {

System.out.println(**"Found "** + parser.errors.size() + **" errors:"**);

**for** (String m : parser.errors) {

System.out.println(m);

}

}

**else** {

System.out.println(**"Compiled successfully"**);

System.out.println(**"Constants:"**);

**int** i = 1;

**for** (String m : parser.constants) {

System.out.println(**"D0"** + i + **" "** + m);

i++;

}

System.out.println(**"program "** + parser.program);

**for** (String m : parser.instructions) {

System.out.println(m);

}

}

}

**public** String getErrorHeader(RecognitionException e) {

**return** **"line "**+e.line+**":"**;

}

**public** **void** emitErrorMessage(String msg) {

errors.add(msg);

}

**public** **int** getInstructionNumber() {

instructionNumber++;

**return** instructionNumber - 1;

}

}

*ID* : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\*

;

*DIGIT* : '0'..'9'+

;

*STRING* : '"' ~'"'\* '"'

;

*WS* : ( ' '

| '\t'

| '\r'

| '\n'

) {$channel=HIDDEN;}

;

type

: 'int' | 'char[]'

;

assign **returns** [String idName, **int** idLine, String expressionType, **int** instructionNumber]

: *ID* '=' (expression) {

$idName = $ID.text;

$idLine = $ID.line;

$expressionType = $expression.expressionType;

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

//constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + **"("** + ($instructionNumber - 1) + **") "** + **"("** + $expression.instructionNumber + **")"**);

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + **"("** + ($instructionNumber - 1) + **") "** + **"("** + $expression.instructionNumber + **")"**);

}

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + **"("** + ($instructionNumber - 1) + **") ("** + $expression.instructionNumber + **")"**);

}

}

;

assign\_operator **returns** [**int** instructionNumber]

: assign ';' {

**if** (!names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" is not declarated"**);

}

**else** {

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

**else** {

$instructionNumber = $assign.instructionNumber;

}

}

}

;

expression **returns** [String expressionType, String expressionClass, **int** instructionNumber]

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$expressionType = names.get($ID.text).getType();

$expressionClass = **"id"**;

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

}

}

| *STRING* {

$expressionType = **"char[]"**;

$expressionClass = **"const\_string"**;

$instructionNumber = getInstructionNumber();

constants.add($STRING.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"String"** + **" D0"** + constants.size());

}

| *DIGIT* {

$expressionType = **"int"**;

$expressionClass = **"const\_int"**;

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"Number"** + **" "** + $DIGIT.text);

}

| arifmetic\_expression {

$expressionType = **"int"**;

$expressionClass = **"arifmetic\_expression"**;

$instructionNumber = $arifmetic\_expression.instructionNumber;

}

| function\_call[**false**] {

$expressionType = $function\_call.resultType;

$expressionClass = **"function\_call"**;

$instructionNumber = $function\_call.instructionNumber;

}

| array\_element {

$expressionType = **"int"**;

$expressionClass = **"array\_element"**;

$instructionNumber = $array\_element.instructionNumber;

}

| getting\_address {

$expressionType = **"int"**;

$expressionClass = **"getting\_address"**;

$instructionNumber = $getting\_address.instructionNumber;

}

;

arifmetic\_operand **returns** [**int** instructionNumber]

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

}

}

}

| *DIGIT* {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"Number"** + **" "** + $DIGIT.text);

}

;

arifmetic\_expression **returns**[**int** instructionNumber]

: operandFirst = arifmetic\_operand arifmetic\_sign operandSecond = arifmetic\_operand {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + $arifmetic\_sign.text + **" "** + **"("** + $operandFirst.instructionNumber + **")"** + **" "** + **"("** + $operandSecond.instructionNumber + **")"**);

}

;

arifmetic\_sign

: ('+' | '-' | '\*' | '/')

;

incrementation **returns** [String idName, **int** instructionNumber]

: *ID* '++' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

**else** {

$idName = $ID.text;

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"++"** + **" "** + **"("** + ($instructionNumber - 1) + **")"** );

}

}

}

;

array\_element **returns** [**int** instructionNumber]

: *ID* '[' expression ']' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"char[]"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

**if** (!$expression.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": wrong index type"**);

}

**else** {

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

//constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + **"("** + ($instructionNumber - 1) + **") "** + **" "** + **"("** + $expression.instructionNumber + **")"** );

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + **"("** + ($instructionNumber - 1) + **")"** + **" "** + **"("** + $expression.instructionNumber + **")"**);

}

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + **"("** + ($instructionNumber - 1) + **")"** + **" ("** + $expression.instructionNumber + **")"**);

}

}

}

;

getting\_address **returns**[**int** instructionNumber]

: '&' *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"&"** + **" "** + **"("** + ($instructionNumber - 1) + **")"**);

}

}

;

function\_call[**boolean** flag] **returns** [String resultType, **int** instructionNumber]

: *ID* '(' parametres[$ID.line]? ')' {

$instructionNumber = getInstructionNumber();

**if** ($flag) {

$instructionNumber = getInstructionNumber();

}

String resultString = **"("** + $instructionNumber + **") "** + $ID.text + **" "**;

**for** (**int** number : $parametres.instructionsList) {

resultString += **"("** + number + **") "**;

}

**if** ($ID.text.equals(**"strlen"**)) {

$resultType = **"int"**;

}

**else** {

$resultType = **"void"**;

}

instructions.add(resultString);

}

;

parametres [**int** functionLine] **returns** [ArrayList<Integer> instructionsList]

: {

$instructionsList = **new** ArrayList<Integer>();

}

((expressionFirst = expression) {

**if** ($expressionFirst.expressionType != **null**) {

**if** ($expressionFirst.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionFirst.text);

}

**else** {

**int** instructionNumber = getInstructionNumber();

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionFirst.text);

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionFirst.instructionNumber + **")"**);

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionFirst.instructionNumber + **")"**);

}

$instructionsList.add(instructionNumber);

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" ("** + $expressionFirst.instructionNumber + **")"**);

$instructionsList.add(instructionNumber);

}

}

}

} (',' expressionNext = expression {

**if** ($expressionNext.expressionType != **null**) {

**if** ($expressionNext.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionNext.text);

}

**else** {

instructionNumber = getInstructionNumber();

**if** ($expressionNext.expressionClass.equals(**"id"**) ||

$expressionNext.expressionClass.equals(**"const\_string"**) ||

$expressionNext.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionNext.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionNext.text);

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionNext.instructionNumber + **")"** );

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionNext.instructionNumber + **")"**);

}

$instructionsList.add(instructionNumber);

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" ("** + $expressionNext.instructionNumber + **")"**);

$instructionsList.add(instructionNumber);

}

}

}

})\*)

;

variable\_declaration **returns** [**int** instructionNumber]

: type (idParam = *ID* | assignParam = assign) ';' {

**if** (names.isExist($idParam.text)) {

errors.add(**"line "** + $idParam.line + **": name "** + $idParam.text + **" duplicated"**);

}

**else** {

**if** ($idParam.text != **null**) {

names.add(names.**new** Name($idParam.text, $type.text, $idParam.line));

}

}

**if** (names.isExist($assignParam.idName)) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" duplicated"**);

}

**else** {

**if** ($assignParam.idName != **null**) {

names.add(names.**new** Name($assignParam.idName, $type.text, $assignParam.idLine));

$instructionNumber = $assignParam.instructionNumber;

}

}

**if** ($assignParam.expressionType != **null** && $assignParam.idName != **null** && !$assignParam.expressionType.equals(names.get($assignParam.idName).getType())) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" type is mismatched"**);

}

}

;

for\_operator **returns** [**int** instructionNumber]

: {

ArrayList<Integer> forInstructions = **new** ArrayList<Integer>();

}

'for' '(' type? assign {

**if** (names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" duplicated"**);

}

**else** {

**if** ($assign.text != **null**) {

names.add(names.**new** Name($assign.idName, $type.text, $assign.idLine));

}

}

**if** (!$type.text.equals(**"int"**)) {

errors.add(**"line "** + $assign.idLine + **": type of counter is mismatched"**);

}

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

';' logical\_condition[$assign.idLine] ';' incrementation ')' '{' {

$instructionNumber = getInstructionNumber();

String resultString = **"("** + instructionNumber + **") "** + **"for"** + **" ("** + $assign.instructionNumber + **")"** + **" ("** + $logical\_condition.instructionNumber + **") "** + **"("** + $incrementation.instructionNumber + **") "**;

instructions.add(resultString);

}

(operator {

forInstructions.add($operator.instructionNumber);

})+ '}' {

**for** (**int** i = instructions.size() - 1; i > 0; i--) {

**if** (instructions.get(i).indexOf(**"for"**) != -1) {

**for** (**int** number : forInstructions) {

String temp = instructions.get(i);

temp += **"("** + number + **") "**;

instructions.set(i, temp);

}

}

}

}

;

function\_call\_operator **returns**[**int** instructionNumber]

: function\_call[**true**] ';' {

$instructionNumber = $function\_call.instructionNumber;

}

;

logical\_condition [**int** operatorLine] **returns** [**int** instructionNumber]

: expressionFirst = expression logical\_link expressionSecond = expression {

**if** ($expressionFirst.expressionType != **null**) {

**if** (!$expressionFirst.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionSecond.expressionType != **null**) {

**if** (!$expressionSecond.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionSecond.text);

}

**else** {

String resultString = **""**;

$instructionNumber = getInstructionNumber();

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionFirst.text);

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + **"D0"** + (constants.indexOf($expressionFirst.text) + 1) + **" "**;

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + $expressionFirst.text + **" "**;

}

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" ("** + $expressionFirst.instructionNumber + **") "**;

}

**if** ($expressionSecond.expressionClass.equals(**"id"**) ||

$expressionSecond.expressionClass.equals(**"const\_string"**) ||

$expressionSecond.expressionClass.equals(**"const\_int"**)) {

$instructionNumber = getInstructionNumber();

**if** ($expressionSecond.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionSecond.text);

resultString += **"D0"** + (constants.indexOf($expressionSecond.text) + 1);

}

**else** {

resultString += $expressionSecond.text;

}

instructions.add(resultString);

}

**else** {

resultString += **"("** + $expressionSecond.instructionNumber + **")"**;

instructions.add(resultString);

}

}

}

}

}

}

;

logical\_link

: '==' | '!=' | '>' | '<' | '>=' | '<='

;

operator **returns** [**int** instructionNumber]

: variable\_declaration {

$instructionNumber = $variable\_declaration.instructionNumber;

}

| assign\_operator {

$instructionNumber = $assign\_operator.instructionNumber;

}

| for\_operator {

$instructionNumber = $for\_operator.instructionNumber;

}

| function\_call\_operator {

$instructionNumber = $function\_call\_operator.instructionNumber;

}

;

program

: (operator {

**if** ($operator.instructionNumber != 0) {

program += **"("** + $operator.instructionNumber + **") "**;

}

})+

;

* 1. Для дага.

**grammar** Task5Grammar;

**options** {

language = Java;

}

@members {

**protected** NamesTable names = **new** NamesTable();

**protected** ArrayList<String> errors = **new** ArrayList<String>();

**protected** ArrayList<String> instructions = **new** ArrayList<String>();

**protected** ArrayList<String> constants = **new** ArrayList<String>();

**protected** **int** instructionNumber = 0;

**protected** String program = **""**;

**public** **static** **void** main(String[] args) **throws** Exception {

Task5GrammarLexer lex = **new** Task5GrammarLexer(**new** ANTLRFileStream(**"test.txt"**));

Task5GrammarParser parser = **new** Task5GrammarParser(**new** CommonTokenStream(lex));

parser.program();

**if** (! parser.errors.isEmpty()) {

System.out.println(**"Found "** + parser.errors.size() + **" errors:"**);

**for** (String m : parser.errors) {

System.out.println(m);

}

}

**else** {

System.out.println(**"Compiled successfully"**);

System.out.println(**"Constants:"**);

**int** i = 1;

**for** (String m : parser.constants) {

System.out.println(**"D0"** + i + **" "** + m);

i++;

}

System.out.println(**"program "** + parser.program);

**for** (String m : parser.instructions) {

System.out.println(m);

}

}

}

**public** String getErrorHeader(RecognitionException e) {

**return** **"line "**+e.line+**":"**;

}

**public** **void** emitErrorMessage(String msg) {

errors.add(msg);

}

**public** **int** getInstructionNumber() {

instructionNumber++;

**return** instructionNumber - 1;

}

**public** **void** check() {

String instruction = instructions.get(instructions.size() - 1);

instruction = instruction.substring(instruction.indexOf(**")"**) + 1);

**boolean** flag = **false**;

**for** (**int** i = 0; i < instructions.size() - 1 && !flag; i++) {

**if** (instructions.get(i).indexOf(instruction) != -1) {

flag = **true**;

}

}

**if** (flag) {

instructionNumber--;

instructions.remove(instructions.get(instructions.size() - 1) );

}

}

}

*ID* : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\*

;

*DIGIT* : '0'..'9'+

;

*STRING* : '"' ~'"'\* '"'

;

*WS* : ( ' '

| '\t'

| '\r'

| '\n'

) {$channel=HIDDEN;}

;

type

: 'int' | 'char[]'

;

assign **returns** [String idName, **int** idLine, String expressionType, **int** instructionNumber]

: *ID* '=' (expression) {

$idName = $ID.text;

$idLine = $ID.line;

$expressionType = $expression.expressionType;

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

//constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + **"("** + ($instructionNumber - 1) + **") "** + **"("** + $expression.instructionNumber + **")"**);

check();

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + **"("** + ($instructionNumber - 1) + **") "** + **"("** + $expression.instructionNumber + **")"**);

check();

}

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + **"("** + ($instructionNumber - 1) + **") ("** + $expression.instructionNumber + **")"**);

check();

}

}

;

assign\_operator **returns** [**int** instructionNumber]

: assign ';' {

**if** (!names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" is not declarated"**);

}

**else** {

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

**else** {

$instructionNumber = $assign.instructionNumber;

}

}

}

;

expression **returns** [String expressionType, String expressionClass, **int** instructionNumber]

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$expressionType = names.get($ID.text).getType();

$expressionClass = **"id"**;

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

}

}

| *STRING* {

$expressionType = **"char[]"**;

$expressionClass = **"const\_string"**;

$instructionNumber = getInstructionNumber();

constants.add($STRING.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"String"** + **" D0"** + constants.size());

check();

}

| *DIGIT* {

$expressionType = **"int"**;

$expressionClass = **"const\_int"**;

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"Number"** + **" "** + $DIGIT.text);

check();

}

| arifmetic\_expression {

$expressionType = **"int"**;

$expressionClass = **"arifmetic\_expression"**;

$instructionNumber = $arifmetic\_expression.instructionNumber;

}

| function\_call[**false**] {

$expressionType = $function\_call.resultType;

$expressionClass = **"function\_call"**;

$instructionNumber = $function\_call.instructionNumber;

}

| array\_element {

$expressionType = **"int"**;

$expressionClass = **"array\_element"**;

$instructionNumber = $array\_element.instructionNumber;

}

| getting\_address {

$expressionType = **"int"**;

$expressionClass = **"getting\_address"**;

$instructionNumber = $getting\_address.instructionNumber;

}

;

arifmetic\_operand **returns** [**int** instructionNumber]

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

}

}

}

| *DIGIT* {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"Number"** + **" "** + $DIGIT.text);

check();

}

;

arifmetic\_expression **returns**[**int** instructionNumber]

: operandFirst = arifmetic\_operand arifmetic\_sign operandSecond = arifmetic\_operand {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + $arifmetic\_sign.text + **" "** + **"("** + $operandFirst.instructionNumber + **")"** + **" "** + **"("** + $operandSecond.instructionNumber + **")"**);

check();

}

;

arifmetic\_sign

: ('+' | '-' | '\*' | '/')

;

incrementation **returns** [String idName, **int** instructionNumber]

: *ID* '++' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

**else** {

$idName = $ID.text;

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"++"** + **" "** + **"("** + ($instructionNumber - 1) + **")"** );

check();

}

}

}

;

array\_element **returns** [**int** instructionNumber]

: *ID* '[' expression ']' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"char[]"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

**if** (!$expression.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": wrong index type"**);

}

**else** {

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

//constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + **"("** + ($instructionNumber - 1) + **") "** + **" "** + **"("** + $expression.instructionNumber + **")"** );

check();

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + **"("** + ($instructionNumber - 1) + **")"** + **" "** + **"("** + $expression.instructionNumber + **")"**);

check();

}

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + **"("** + ($instructionNumber - 1) + **")"** + **" ("** + $expression.instructionNumber + **")"**);

check();

}

}

}

;

getting\_address **returns**[**int** instructionNumber]

: '&' *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"ID"** + **" "** + $ID.text);

check();

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"&"** + **" "** + **"("** + ($instructionNumber - 1) + **")"**);

check();

}

}

;

function\_call[**boolean** flag] **returns** [String resultType, **int** instructionNumber]

: *ID* '(' parametres[$ID.line]? ')' {

$instructionNumber = getInstructionNumber();

**if** ($flag) {

$instructionNumber = getInstructionNumber();

}

String resultString = **"("** + $instructionNumber + **") "** + $ID.text + **" "**;

**for** (**int** number : $parametres.instructionsList) {

resultString += **"("** + number + **") "**;

}

**if** ($ID.text.equals(**"strlen"**)) {

$resultType = **"int"**;

}

**else** {

$resultType = **"void"**;

}

instructions.add(resultString);

check();

}

;

parametres [**int** functionLine] **returns** [ArrayList<Integer> instructionsList]

: {

$instructionsList = **new** ArrayList<Integer>();

}

((expressionFirst = expression) {

**if** ($expressionFirst.expressionType != **null**) {

**if** ($expressionFirst.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionFirst.text);

}

**else** {

**int** instructionNumber = getInstructionNumber();

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionFirst.text);

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionFirst.instructionNumber + **")"**);

check();

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionFirst.instructionNumber + **")"**);

check();

}

$instructionsList.add(instructionNumber);

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" ("** + $expressionFirst.instructionNumber + **")"**);

check();

$instructionsList.add(instructionNumber);

}

}

}

} (',' expressionNext = expression {

**if** ($expressionNext.expressionType != **null**) {

**if** ($expressionNext.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionNext.text);

}

**else** {

instructionNumber = getInstructionNumber();

**if** ($expressionNext.expressionClass.equals(**"id"**) ||

$expressionNext.expressionClass.equals(**"const\_string"**) ||

$expressionNext.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionNext.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionNext.text);

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionNext.instructionNumber + **")"** );

check();

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" "** + **"("** + $expressionNext.instructionNumber + **")"**);

check();

}

$instructionsList.add(instructionNumber);

}

**else** {

instructions.add(**"("** + instructionNumber + **") "** + **"param"** + **" ("** + $expressionNext.instructionNumber + **")"**);

check();

$instructionsList.add(instructionNumber);

}

}

}

})\*)

;

variable\_declaration **returns** [**int** instructionNumber]

: type (idParam = *ID* | assignParam = assign) ';' {

**if** (names.isExist($idParam.text)) {

errors.add(**"line "** + $idParam.line + **": name "** + $idParam.text + **" duplicated"**);

}

**else** {

**if** ($idParam.text != **null**) {

names.add(names.**new** Name($idParam.text, $type.text, $idParam.line));

}

}

**if** (names.isExist($assignParam.idName)) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" duplicated"**);

}

**else** {

**if** ($assignParam.idName != **null**) {

names.add(names.**new** Name($assignParam.idName, $type.text, $assignParam.idLine));

$instructionNumber = $assignParam.instructionNumber;

}

}

**if** ($assignParam.expressionType != **null** && $assignParam.idName != **null** && !$assignParam.expressionType.equals(names.get($assignParam.idName).getType())) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" type is mismatched"**);

}

}

;

for\_operator **returns** [**int** instructionNumber]

: {

ArrayList<Integer> forInstructions = **new** ArrayList<Integer>();

}

'for' '(' type? assign {

**if** (names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" duplicated"**);

}

**else** {

**if** ($assign.text != **null**) {

names.add(names.**new** Name($assign.idName, $type.text, $assign.idLine));

}

}

**if** (!$type.text.equals(**"int"**)) {

errors.add(**"line "** + $assign.idLine + **": type of counter is mismatched"**);

}

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

';' logical\_condition[$assign.idLine] ';' incrementation ')' '{' {

$instructionNumber = getInstructionNumber();

String resultString = **"("** + instructionNumber + **") "** + **"for"** + **" ("** + $assign.instructionNumber + **")"** + **" ("** + $logical\_condition.instructionNumber + **") "** + **"("** + $incrementation.instructionNumber + **") "**;

instructions.add(resultString);

check();

}

(operator {

forInstructions.add($operator.instructionNumber);

})+ '}' {

**for** (**int** i = instructions.size() - 1; i > 0; i--) {

**if** (instructions.get(i).indexOf(**"for"**) != -1) {

**for** (**int** number : forInstructions) {

String temp = instructions.get(i);

temp += **"("** + number + **") "**;

instructions.set(i, temp);

}

}

}

}

;

function\_call\_operator **returns**[**int** instructionNumber]

: function\_call[**true**] ';' {

$instructionNumber = $function\_call.instructionNumber;

}

;

logical\_condition [**int** operatorLine] **returns** [**int** instructionNumber]

: expressionFirst = expression logical\_link expressionSecond = expression {

**if** ($expressionFirst.expressionType != **null**) {

**if** (!$expressionFirst.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionSecond.expressionType != **null**) {

**if** (!$expressionSecond.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionSecond.text);

}

**else** {

String resultString = **""**;

$instructionNumber = getInstructionNumber();

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionFirst.text);

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + **"D0"** + (constants.indexOf($expressionFirst.text) + 1) + **" "**;

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + $expressionFirst.text + **" "**;

}

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" ("** + $expressionFirst.instructionNumber + **") "**;

}

**if** ($expressionSecond.expressionClass.equals(**"id"**) ||

$expressionSecond.expressionClass.equals(**"const\_string"**) ||

$expressionSecond.expressionClass.equals(**"const\_int"**)) {

$instructionNumber = getInstructionNumber();

**if** ($expressionSecond.expressionClass.equals(**"const\_string"**)) {

//constants.add($expressionSecond.text);

resultString += **"D0"** + (constants.indexOf($expressionSecond.text) + 1);

}

**else** {

resultString += $expressionSecond.text;

}

instructions.add(resultString);

check();

}

**else** {

resultString += **"("** + $expressionSecond.instructionNumber + **")"**;

instructions.add(resultString);

check();

}

}

}

}

}

}

;

logical\_link

: '==' | '!=' | '>' | '<' | '>=' | '<='

;

operator **returns** [**int** instructionNumber]

: variable\_declaration {

$instructionNumber = $variable\_declaration.instructionNumber;

}

| assign\_operator {

$instructionNumber = $assign\_operator.instructionNumber;

}

| for\_operator {

$instructionNumber = $for\_operator.instructionNumber;

}

| function\_call\_operator {

$instructionNumber = $function\_call\_operator.instructionNumber;

}

;

program

: (operator {

**if** ($operator.instructionNumber != 0) {

program += **"("** + $operator.instructionNumber + **") "**;

}

})+

;

* 1. Для четверок.

**grammar** Task5Grammar;

**options** {

language = Java;

}

@members {

**protected** NamesTable names = **new** NamesTable();

**protected** ArrayList<String> errors = **new** ArrayList<String>();

**protected** ArrayList<String> instructions = **new** ArrayList<String>();

**protected** ArrayList<String> constants = **new** ArrayList<String>();

**protected** **int** currentTempNumber = 0;

**protected** **int** instructionNumber = 0;

**public** **static** **void** main(String[] args) **throws** Exception {

Task5GrammarLexer lex = **new** Task5GrammarLexer(**new** ANTLRFileStream(**"test.txt"**));

Task5GrammarParser parser = **new** Task5GrammarParser(**new** CommonTokenStream(lex));

parser.program();

**if** (! parser.errors.isEmpty()) {

System.out.println(**"Found "** + parser.errors.size() + **" errors:"**);

**for** (String m : parser.errors) {

System.out.println(m);

}

}

**else** {

System.out.println(**"Compiled successfully"**);

System.out.println(**"Constants:"**);

**int** i = 1;

**for** (String m : parser.constants) {

System.out.println(**"D0"** + i + **" "** + m);

i++;

}

System.out.println(**"Instructions:"**);

parser.instructions.add(**"("** + (parser.instructions.size()) + **")"**);

**for** (String m : parser.instructions) {

System.out.println(m);

}

}

}

**public** String getErrorHeader(RecognitionException e) {

**return** **"line "**+e.line+**":"**;

}

**public** **void** emitErrorMessage(String msg) {

errors.add(msg);

}

**public** **int** genNewTemp() {

currentTempNumber++;

**return** currentTempNumber;

}

**public** **int** getInstructionNumber() {

instructionNumber++;

**return** instructionNumber - 1;

}

}

*ID* : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\*

;

*DIGIT* : '0'..'9'+

;

*STRING* : '"' ~'"'\* '"'

;

*WS* : ( ' '

| '\t'

| '\r'

| '\n'

) {$channel=HIDDEN;}

;

type

: 'int' | 'char[]'

;

assign **returns** [String idName, **int** idLine, String expressionType, **int** instructionNumber]

: *ID* '=' (expression) {

$idName = $ID.text;

$idLine = $ID.line;

$expressionType = $expression.expressionType;

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + **"D0"** + constants.size() + **" - "** + $ID.text);

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + $expression.text + **" - "** + $ID.text);

}

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" t"** + $expression.tempIndex + **" - "** + $ID.text);

}

}

;

assign\_operator

: assign ';' {

**if** (!names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" is not declarated"**);

}

**else** {

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

}

;

expression **returns** [String expressionType, String expressionClass, **int** tempIndex]

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$expressionType = names.get($ID.text).getType();

$expressionClass = **"id"**;

}

}

| *STRING* {

$expressionType = **"char[]"**;

$expressionClass = **"const\_string"**;

}

| *DIGIT* {

$expressionType = **"int"**;

$expressionClass = **"const\_int"**;

}

| arifmetic\_expression {

$expressionType = **"int"**;

$expressionClass = **"arifmetic\_expression"**;

$tempIndex = $arifmetic\_expression.tempVariableIndex;

}

| function\_call {

$expressionType = $function\_call.resultType;

$expressionClass = **"function\_call"**;

$tempIndex = $function\_call.tempVariableIndex;

}

| array\_element {

$expressionType = **"int"**;

$expressionClass = **"array\_element"**;

$tempIndex = $array\_element.tempVariableIndex;

}

| getting\_address {

$expressionType = **"int"**;

$expressionClass = **"getting\_address"**;

$tempIndex = $getting\_address.tempVariableIndex;

}

;

arifmetic\_operand

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

}

| *DIGIT*

;

arifmetic\_expression **returns**[**int** tempVariableIndex]

: operandFirst = arifmetic\_operand arifmetic\_sign operandSecond = arifmetic\_operand {

$tempVariableIndex = genNewTemp();

instructions.add(**"("** + getInstructionNumber() + **") "** + $arifmetic\_sign.text + **" "** + $operandFirst.text + **" "** + $operandSecond.text + **" t"** + $tempVariableIndex);

}

;

arifmetic\_sign

: ('+' | '-' | '\*' | '/')

;

incrementation **returns** [String idName]

: *ID* '++' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

**else** {

$idName = $ID.text;

}

}

}

;

array\_element **returns** [**int** tempVariableIndex]

: *ID* '[' expression ']' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"char[]"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

**if** (!$expression.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": wrong index type"**);

}

**else** {

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

$tempVariableIndex = genNewTemp();

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

constants.add($expression.text);

instructions.add(**"("** + getInstructionNumber() + **") "** + **"[]"** + **" "** + $ID.text + **" "** + **"D0"** + constants.size() + **" - "** + **"t"** + $tempVariableIndex);

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"[]"** + **" "** + $ID.text + **" "** + $expression.text + **" - "** + **"t"** + $tempVariableIndex);

}

}

**else** {

$tempVariableIndex = genNewTemp();

instructions.add(**"("** + getInstructionNumber() + **") "** + **"[]"** + **" "** + $ID.text + **" t"** + $expression.tempIndex + **" - "** + **"t"** + $tempVariableIndex);

}

}

}

;

getting\_address **returns**[**int** tempVariableIndex]

: '&' *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$tempVariableIndex = genNewTemp();

instructions.add(**"("** + getInstructionNumber() + **") "** + **"&"** + **" "** + $ID.text + **" - "** + **" t"** + $tempVariableIndex);

}

}

;

function\_call **returns** [String resultType, **int** tempVariableIndex]

: *ID* '(' parametres[$ID.line] ')' {

**if** ($ID.text.equals(**"strlen"**)) {

$resultType = **"int"**;

$tempVariableIndex = genNewTemp();

instructions.add(**"("** + getInstructionNumber() + **") "** + **"call"** + **" strlen"** + **" 1"** + **" t"** + $tempVariableIndex);

}

**else** {

$resultType = **"void"**;

instructions.add(**"("** + getInstructionNumber() + **") "** + **"call"** + **" "** + $ID.text + **" "** + $parametres.parametresCount);

}

}

;

parametres [**int** functionLine] **returns** [**int** parametresCount]

: {

$parametresCount = 0;

}

((expressionFirst = expression) {

**if** ($expressionFirst.expressionType != **null**) {

**if** ($expressionFirst.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionFirst.text);

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + $expressionFirst.text);

}

$parametresCount++;

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" t"** + $expressionFirst.tempIndex);

$parametresCount++;

}

}

}

} (',' expressionNext = expression {

**if** ($expressionNext.expressionType != **null**) {

**if** ($expressionNext.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionNext.text);

}

**else** {

**if** ($expressionNext.expressionClass.equals(**"id"**) ||

$expressionNext.expressionClass.equals(**"const\_string"**) ||

$expressionNext.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionNext.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionNext.text);

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + $expressionNext.text);

}

$parametresCount++;

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" t"** + $expressionNext.tempIndex);

$parametresCount++;

}

}

}

})\*)

;

variable\_declaration

: type (idParam = *ID* | assignParam = assign) (',' (idParamNext = *ID* | assignParamNext = assign))\* ';' {

**if** (names.isExist($idParam.text)) {

errors.add(**"line "** + $idParam.line + **": name "** + $idParam.text + **" duplicated"**);

}

**else** {

**if** ($idParam.text != **null**) {

names.add(names.**new** Name($idParam.text, $type.text, $idParam.line));

}

}

**if** (names.isExist($assignParam.idName)) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" duplicated"**);

}

**else** {

**if** ($assignParam.idName != **null**) {

names.add(names.**new** Name($assignParam.idName, $type.text, $assignParam.idLine));

}

}

**if** (names.isExist($idParamNext.text)) {

errors.add(**"line "** + $idParamNext.line + **": name "** + $idParamNext.text + **" duplicated"**);

}

**else** {

**if** ($idParamNext.text != **null**) {

names.add(names.**new** Name($idParamNext.text, $type.text, $idParamNext.line));

}

}

**if** (names.isExist($assignParamNext.idName)) {

errors.add(**"line "** + $assignParamNext.idLine + **": name "** + $assignParamNext.idName + **" duplicated"**);

}

**else** {

**if** ($assignParamNext.idName != **null**) {

names.add(names.**new** Name($assignParamNext.idName, $type.text, $assignParamNext.idLine));

}

}

**if** ($assignParam.expressionType != **null** && $assignParam.idName != **null** && !$assignParam.expressionType.equals(names.get($assignParam.idName).getType())) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" type is mismatched"**);

}

**if** ($assignParam.expressionType != **null** && $assignParamNext.idName != **null** && !$assignParamNext.expressionType.equals(names.get($assignParamNext.idName).getType())) {

errors.add(**"line "** + $assignParamNext.idLine + **": name "** + $assignParamNext.idName + **" type is mismatched"**);

}

}

;

for\_operator

: 'for' '(' type? assign {

**if** (names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" duplicated"**);

}

**else** {

**if** ($assign.text != **null**) {

names.add(names.**new** Name($assign.idName, $type.text, $assign.idLine));

}

}

**if** (!$type.text.equals(**"int"**)) {

errors.add(**"line "** + $assign.idLine + **": type of counter is mismatched"**);

}

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

';' logical\_condition[$assign.idLine] ';' incrementation ')' '{' {

**int** instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"if"** + **" t"** + $logical\_condition.tempVariableIndex + **" ("** + (instructionNumber + 2) + **")"**);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"goto"** + **" "** + **"("** + **"X"** + **")"**);

}

operator+

'}' {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"+"** + **" "** + $incrementation.idName + **" 1 t"** + genNewTemp());

instructions.add(**"("** + getInstructionNumber() + **") "** + **"assign"** + **" t"** + currentTempNumber + **" "** + **"- "** + $incrementation.idName);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"goto"** + **" ("** + ($assign.instructionNumber + 1) + **")"**);

**for** (**int** i = instructions.size() - 1; i > 0; i--) {

**if** (instructions.get(i).indexOf(**"X"**) != -1) {

String temp = instructions.get(i);

**int** position = temp.indexOf(**"X"**);

temp = temp.substring(0, position) + Integer.toString(instructionNumber + 1) + temp.substring(position + 1);

instructions.set(i, temp);

}

}

}

;

function\_call\_operator

: function\_call ';'

;

logical\_condition [**int** operatorLine] **returns** [**int** tempVariableIndex]

: expressionFirst = expression logical\_link expressionSecond = expression {

**if** ($expressionFirst.expressionType != **null**) {

**if** (!$expressionFirst.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionSecond.expressionType != **null**) {

**if** (!$expressionSecond.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionSecond.text);

}

**else** {

String resultString = **""**;

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionFirst.text);

resultString = **"("** + getInstructionNumber() + **") "** + $logical\_link.text + **" "** + **"D0"** + constants.size() + **" "**;

}

**else** {

resultString = **"("** + getInstructionNumber() + **") "** + $logical\_link.text + **" "** + $expressionFirst.text + **" "**;

}

}

**else** {

resultString = **"("** + getInstructionNumber() + **") "** + $logical\_link.text + **" t"** + $expressionFirst.tempIndex + **" "**;

}

**if** ($expressionSecond.expressionClass.equals(**"id"**) ||

$expressionSecond.expressionClass.equals(**"const\_string"**) ||

$expressionSecond.expressionClass.equals(**"const\_int"**)) {

$tempVariableIndex = genNewTemp();

**if** ($expressionSecond.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionSecond.text);

resultString += **"D0"** + constants.size() + **" t"** + $tempVariableIndex;

}

**else** {

resultString += $expressionSecond.text + **" t"** + $tempVariableIndex;

}

instructions.add(resultString);

}

**else** {

$tempVariableIndex = genNewTemp();

resultString += **"t"** + $expressionSecond.tempIndex + **" t"** + $tempVariableIndex;

instructions.add(resultString);

}

}

}

}

}

}

;

logical\_link

: '==' | '!=' | '>' | '<' | '>=' | '<='

;

operator : variable\_declaration | assign\_operator | for\_operator | function\_call\_operator

;

program : (operator)+

;

* 1. Для троек.

**grammar** Task5Grammar;

**options** {

language = Java;

}

@members {

**protected** NamesTable names = **new** NamesTable();

**protected** ArrayList<String> errors = **new** ArrayList<String>();

**protected** ArrayList<String> instructions = **new** ArrayList<String>();

**protected** ArrayList<String> constants = **new** ArrayList<String>();

**protected** **int** instructionNumber = 0;

**public** **static** **void** main(String[] args) **throws** Exception {

Task5GrammarLexer lex = **new** Task5GrammarLexer(**new** ANTLRFileStream(**"test.txt"**));

Task5GrammarParser parser = **new** Task5GrammarParser(**new** CommonTokenStream(lex));

parser.program();

**if** (! parser.errors.isEmpty()) {

System.out.println(**"Found "** + parser.errors.size() + **" errors:"**);

**for** (String m : parser.errors) {

System.out.println(m);

}

}

**else** {

System.out.println(**"Compiled successfully"**);

System.out.println(**"Constants:"**);

**int** i = 1;

**for** (String m : parser.constants) {

System.out.println(**"D0"** + i + **" "** + m);

i++;

}

System.out.println(**"Instructions:"**);

parser.instructions.add(**"("** + (parser.instructions.size()) + **")"**);

**for** (String m : parser.instructions) {

System.out.println(m);

}

}

}

**public** String getErrorHeader(RecognitionException e) {

**return** **"line "**+e.line+**":"**;

}

**public** **void** emitErrorMessage(String msg) {

errors.add(msg);

}

**public** **int** getInstructionNumber() {

instructionNumber++;

**return** instructionNumber - 1;

}

}

*ID* : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\*

;

*DIGIT* : '0'..'9'+

;

*STRING* : '"' ~'"'\* '"'

;

*WS* : ( ' '

| '\t'

| '\r'

| '\n'

) {$channel=HIDDEN;}

;

type

: 'int' | 'char[]'

;

assign **returns** [String idName, **int** idLine, String expressionType, **int** instructionNumber]

: *ID* '=' (expression) {

$idName = $ID.text;

$idLine = $ID.line;

$expressionType = $expression.expressionType;

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + $ID.text + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + $ID.text + **" "** + $expression.text);

}

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + $ID.text + **" ("** + $expression.instructionNumber + **")"**);

}

}

;

assign\_operator

: assign ';' {

**if** (!names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" is not declarated"**);

}

**else** {

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

}

;

expression **returns** [String expressionType, String expressionClass, **int** instructionNumber]

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$expressionType = names.get($ID.text).getType();

$expressionClass = **"id"**;

}

}

| *STRING* {

$expressionType = **"char[]"**;

$expressionClass = **"const\_string"**;

}

| *DIGIT* {

$expressionType = **"int"**;

$expressionClass = **"const\_int"**;

}

| arifmetic\_expression {

$expressionType = **"int"**;

$expressionClass = **"arifmetic\_expression"**;

$instructionNumber = $arifmetic\_expression.instructionNumber;

}

| function\_call {

$expressionType = $function\_call.resultType;

$expressionClass = **"function\_call"**;

$instructionNumber = $function\_call.instructionNumber;

}

| array\_element {

$expressionType = **"int"**;

$expressionClass = **"array\_element"**;

$instructionNumber = $array\_element.instructionNumber;

}

| getting\_address {

$expressionType = **"int"**;

$expressionClass = **"getting\_address"**;

$instructionNumber = $getting\_address.instructionNumber;

}

;

arifmetic\_operand

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

}

| *DIGIT*

;

arifmetic\_expression **returns**[**int** instructionNumber]

: operandFirst = arifmetic\_operand arifmetic\_sign operandSecond = arifmetic\_operand {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + $arifmetic\_sign.text + **" "** + $operandFirst.text + **" "** + $operandSecond.text);

}

;

arifmetic\_sign

: ('+' | '-' | '\*' | '/')

;

incrementation **returns** [String idName]

: *ID* '++' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

**else** {

$idName = $ID.text;

}

}

}

;

array\_element **returns** [**int** instructionNumber]

: *ID* '[' expression ']' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"char[]"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

**if** (!$expression.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": wrong index type"**);

}

**else** {

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + $ID.text + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + $ID.text + **" "** + $expression.text);

}

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + $ID.text + **" ("** + $expression.instructionNumber + **")"**);

}

}

}

;

getting\_address **returns**[**int** instructionNumber]

: '&' *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"&"** + **" "** + $ID.text);

}

}

;

function\_call **returns** [String resultType, **int** instructionNumber]

: *ID* '(' parametres[$ID.line] ')' {

$instructionNumber = getInstructionNumber();

**if** ($ID.text.equals(**"strlen"**)) {

$resultType = **"int"**;

instructions.add(**"("** + $instructionNumber + **") "** + **"call"** + **" strlen"** + **" 1"**);

}

**else** {

$resultType = **"void"**;

instructions.add(**"("** + $instructionNumber + **") "** + **"call"** + **" "** + $ID.text + **" "** + $parametres.parametresCount);

}

}

;

parametres [**int** functionLine] **returns** [**int** parametresCount]

: {

$parametresCount = 0;

}

((expressionFirst = expression) {

**if** ($expressionFirst.expressionType != **null**) {

**if** ($expressionFirst.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionFirst.text);

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + $expressionFirst.text);

}

$parametresCount++;

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" ("** + $expressionFirst.instructionNumber + **")"**);

$parametresCount++;

}

}

}

} (',' expressionNext = expression {

**if** ($expressionNext.expressionType != **null**) {

**if** ($expressionNext.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionNext.text);

}

**else** {

**if** ($expressionNext.expressionClass.equals(**"id"**) ||

$expressionNext.expressionClass.equals(**"const\_string"**) ||

$expressionNext.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionNext.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionNext.text);

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + $expressionNext.text);

}

$parametresCount++;

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" ("** + $expressionNext.instructionNumber + **")"**);

$parametresCount++;

}

}

}

})\*)

;

variable\_declaration

: type (idParam = *ID* | assignParam = assign) (',' (idParamNext = *ID* | assignParamNext = assign))\* ';' {

**if** (names.isExist($idParam.text)) {

errors.add(**"line "** + $idParam.line + **": name "** + $idParam.text + **" duplicated"**);

}

**else** {

**if** ($idParam.text != **null**) {

names.add(names.**new** Name($idParam.text, $type.text, $idParam.line));

}

}

**if** (names.isExist($assignParam.idName)) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" duplicated"**);

}

**else** {

**if** ($assignParam.idName != **null**) {

names.add(names.**new** Name($assignParam.idName, $type.text, $assignParam.idLine));

}

}

**if** (names.isExist($idParamNext.text)) {

errors.add(**"line "** + $idParamNext.line + **": name "** + $idParamNext.text + **" duplicated"**);

}

**else** {

**if** ($idParamNext.text != **null**) {

names.add(names.**new** Name($idParamNext.text, $type.text, $idParamNext.line));

}

}

**if** (names.isExist($assignParamNext.idName)) {

errors.add(**"line "** + $assignParamNext.idLine + **": name "** + $assignParamNext.idName + **" duplicated"**);

}

**else** {

**if** ($assignParamNext.idName != **null**) {

names.add(names.**new** Name($assignParamNext.idName, $type.text, $assignParamNext.idLine));

}

}

**if** ($assignParam.expressionType != **null** && $assignParam.idName != **null** && !$assignParam.expressionType.equals(names.get($assignParam.idName).getType())) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" type is mismatched"**);

}

**if** ($assignParam.expressionType != **null** && $assignParamNext.idName != **null** && !$assignParamNext.expressionType.equals(names.get($assignParamNext.idName).getType())) {

errors.add(**"line "** + $assignParamNext.idLine + **": name "** + $assignParamNext.idName + **" type is mismatched"**);

}

}

;

for\_operator

: 'for' '(' type? assign {

**if** (names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" duplicated"**);

}

**else** {

**if** ($assign.text != **null**) {

names.add(names.**new** Name($assign.idName, $type.text, $assign.idLine));

}

}

**if** (!$type.text.equals(**"int"**)) {

errors.add(**"line "** + $assign.idLine + **": type of counter is mismatched"**);

}

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

';' logical\_condition[$assign.idLine] ';' incrementation ')' '{' {

**int** instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"if"** + **" ("** + $logical\_condition.instructionNumber + **") ("** + (instructionNumber + 2) + **")"**);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"goto"** + **" "** + **"("** + **"X"** + **")"**);

}

operator+

'}' {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"+"** + **" "** + $incrementation.idName + **" 1"**);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"assign"** + **" "** + $incrementation.idName + **" ("** + (instructionNumber - 1) + **")"**);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"goto"** + **" ("** + ($assign.instructionNumber + 1) + **")"**);

**for** (**int** i = instructions.size() - 1; i > 0; i--) {

**if** (instructions.get(i).indexOf(**"X"**) != -1) {

String temp = instructions.get(i);

**int** position = temp.indexOf(**"X"**);

temp = temp.substring(0, position) + Integer.toString(instructionNumber + 1) + temp.substring(position + 1);

instructions.set(i, temp);

}

}

}

;

function\_call\_operator

: function\_call ';'

;

logical\_condition [**int** operatorLine] **returns** [**int** instructionNumber]

: expressionFirst = expression logical\_link expressionSecond = expression {

**if** ($expressionFirst.expressionType != **null**) {

**if** (!$expressionFirst.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionSecond.expressionType != **null**) {

**if** (!$expressionSecond.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionSecond.text);

}

**else** {

String resultString = **""**;

$instructionNumber = getInstructionNumber();

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionFirst.text);

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + **"D0"** + constants.size() + **" "**;

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + $expressionFirst.text + **" "**;

}

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" ("** + $expressionFirst.instructionNumber + **") "**;

}

**if** ($expressionSecond.expressionClass.equals(**"id"**) ||

$expressionSecond.expressionClass.equals(**"const\_string"**) ||

$expressionSecond.expressionClass.equals(**"const\_int"**)) {

$instructionNumber = getInstructionNumber();

**if** ($expressionSecond.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionSecond.text);

resultString += **"D0"** + constants.size();

}

**else** {

resultString += $expressionSecond.text;

}

instructions.add(resultString);

}

**else** {

resultString += **"("** + $expressionSecond.instructionNumber + **")"**;

instructions.add(resultString);

}

}

}

}

}

}

;

logical\_link

: '==' | '!=' | '>' | '<' | '>=' | '<='

;

operator : variable\_declaration | assign\_operator | for\_operator | function\_call\_operator

;

program : (operator)+

;

* 1. Для косвенных троек.

**grammar** Task5Grammar;

**options** {

language = Java;

}

@members {

**protected** NamesTable names = **new** NamesTable();

**protected** ArrayList<String> errors = **new** ArrayList<String>();

**protected** ArrayList<String> instructions = **new** ArrayList<String>();

**protected** ArrayList<String> constants = **new** ArrayList<String>();

**protected** **int** instructionNumber = 30;

**public** **static** **void** main(String[] args) **throws** Exception {

Task5GrammarLexer lex = **new** Task5GrammarLexer(**new** ANTLRFileStream(**"test.txt"**));

Task5GrammarParser parser = **new** Task5GrammarParser(**new** CommonTokenStream(lex));

parser.program();

**if** (! parser.errors.isEmpty()) {

System.out.println(**"Found "** + parser.errors.size() + **" errors:"**);

**for** (String m : parser.errors) {

System.out.println(m);

}

}

**else** {

System.out.println(**"Compiled successfully"**);

System.out.println(**"Constants:"**);

**int** i = 1;

**for** (String m : parser.constants) {

System.out.println(**"D0"** + i + **" "** + m);

i++;

}

System.out.println(**"Triples:"**);

parser.instructions.add(**"("** + (parser.instructions.size() + 30) + **")"**);

**for** (String m : parser.instructions) {

System.out.println(m);

}

System.out.println(**"Instructions:"**);

**for** (i = 0; i < parser.instructions.size(); i++) {

System.out.println(**"("** + i + **")"** + **"("** + (i + 30) + **")"**);

}

}

}

**public** String getErrorHeader(RecognitionException e) {

**return** **"line "**+e.line+**":"**;

}

**public** **void** emitErrorMessage(String msg) {

errors.add(msg);

}

**public** **int** getInstructionNumber() {

instructionNumber++;

**return** instructionNumber - 1;

}

}

*ID* : ('a'..'z'|'A'..'Z'|'\_') ('a'..'z'|'A'..'Z'|'0'..'9'|'\_')\*

;

*DIGIT* : '0'..'9'+

;

*STRING* : '"' ~'"'\* '"'

;

*WS* : ( ' '

| '\t'

| '\r'

| '\n'

) {$channel=HIDDEN;}

;

type

: 'int' | 'char[]'

;

assign **returns** [String idName, **int** idLine, String expressionType, **int** instructionNumber]

: *ID* '=' (expression) {

$idName = $ID.text;

$idLine = $ID.line;

$expressionType = $expression.expressionType;

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + $ID.text + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + $ID.text + **" "** + $expression.text);

}

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"assign"** + **" "** + $ID.text + **" ("** + $expression.instructionNumber + **")"**);

}

}

;

assign\_operator

: assign ';' {

**if** (!names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" is not declarated"**);

}

**else** {

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

}

;

expression **returns** [String expressionType, String expressionClass, **int** instructionNumber]

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$expressionType = names.get($ID.text).getType();

$expressionClass = **"id"**;

}

}

| *STRING* {

$expressionType = **"char[]"**;

$expressionClass = **"const\_string"**;

}

| *DIGIT* {

$expressionType = **"int"**;

$expressionClass = **"const\_int"**;

}

| arifmetic\_expression {

$expressionType = **"int"**;

$expressionClass = **"arifmetic\_expression"**;

$instructionNumber = $arifmetic\_expression.instructionNumber;

}

| function\_call {

$expressionType = $function\_call.resultType;

$expressionClass = **"function\_call"**;

$instructionNumber = $function\_call.instructionNumber;

}

| array\_element {

$expressionType = **"int"**;

$expressionClass = **"array\_element"**;

$instructionNumber = $array\_element.instructionNumber;

}

| getting\_address {

$expressionType = **"int"**;

$expressionClass = **"getting\_address"**;

$instructionNumber = $getting\_address.instructionNumber;

}

;

arifmetic\_operand

: *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

}

| *DIGIT*

;

arifmetic\_expression **returns**[**int** instructionNumber]

: operandFirst = arifmetic\_operand arifmetic\_sign operandSecond = arifmetic\_operand {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + $arifmetic\_sign.text + **" "** + $operandFirst.text + **" "** + $operandSecond.text);

}

;

arifmetic\_sign

: ('+' | '-' | '\*' | '/')

;

incrementation **returns** [String idName]

: *ID* '++' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

**else** {

$idName = $ID.text;

}

}

}

;

array\_element **returns** [**int** instructionNumber]

: *ID* '[' expression ']' {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

**if** (!names.get($ID.text).getType().equals(**"char[]"**)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" type mismatched"**);

}

}

**if** (!$expression.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $ID.line + **": wrong index type"**);

}

**else** {

**if** ($expression.expressionClass.equals(**"id"**) ||

$expression.expressionClass.equals(**"const\_string"**) ||

$expression.expressionClass.equals(**"const\_int"**)) {

$instructionNumber = getInstructionNumber();

**if** ($expression.expressionClass.equals(**"const\_string"**)) {

constants.add($expression.text);

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + $ID.text + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + $ID.text + **" "** + $expression.text);

}

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"[]"** + **" "** + $ID.text + **" ("** + $expression.instructionNumber + **")"**);

}

}

}

;

getting\_address **returns**[**int** instructionNumber]

: '&' *ID* {

**if** (!names.isExist($ID.text)) {

errors.add(**"line "** + $ID.line + **": name "** + $ID.text + **" is not declarated"**);

}

**else** {

$instructionNumber = getInstructionNumber();

instructions.add(**"("** + $instructionNumber + **") "** + **"&"** + **" "** + $ID.text);

}

}

;

function\_call **returns** [String resultType, **int** instructionNumber]

: *ID* '(' parametres[$ID.line] ')' {

$instructionNumber = getInstructionNumber();

**if** ($ID.text.equals(**"strlen"**)) {

$resultType = **"int"**;

instructions.add(**"("** + $instructionNumber + **") "** + **"call"** + **" strlen"** + **" 1"**);

}

**else** {

$resultType = **"void"**;

instructions.add(**"("** + $instructionNumber + **") "** + **"call"** + **" "** + $ID.text + **" "** + $parametres.parametresCount);

}

}

;

parametres [**int** functionLine] **returns** [**int** parametresCount]

: {

$parametresCount = 0;

}

((expressionFirst = expression) {

**if** ($expressionFirst.expressionType != **null**) {

**if** ($expressionFirst.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionFirst.text);

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + $expressionFirst.text);

}

$parametresCount++;

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" ("** + $expressionFirst.instructionNumber + **")"**);

$parametresCount++;

}

}

}

} (',' expressionNext = expression {

**if** ($expressionNext.expressionType != **null**) {

**if** ($expressionNext.expressionType.equals(**"void"**)) {

errors.add(**"line "** + $functionLine + **": wrong parameter type of "** + $expressionNext.text);

}

**else** {

**if** ($expressionNext.expressionClass.equals(**"id"**) ||

$expressionNext.expressionClass.equals(**"const\_string"**) ||

$expressionNext.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionNext.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionNext.text);

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + **"D0"** + constants.size());

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" "** + $expressionNext.text);

}

$parametresCount++;

}

**else** {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"param"** + **" ("** + $expressionNext.instructionNumber + **")"**);

$parametresCount++;

}

}

}

})\*)

;

variable\_declaration

: type (idParam = *ID* | assignParam = assign) (',' (idParamNext = *ID* | assignParamNext = assign))\* ';' {

**if** (names.isExist($idParam.text)) {

errors.add(**"line "** + $idParam.line + **": name "** + $idParam.text + **" duplicated"**);

}

**else** {

**if** ($idParam.text != **null**) {

names.add(names.**new** Name($idParam.text, $type.text, $idParam.line));

}

}

**if** (names.isExist($assignParam.idName)) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" duplicated"**);

}

**else** {

**if** ($assignParam.idName != **null**) {

names.add(names.**new** Name($assignParam.idName, $type.text, $assignParam.idLine));

}

}

**if** (names.isExist($idParamNext.text)) {

errors.add(**"line "** + $idParamNext.line + **": name "** + $idParamNext.text + **" duplicated"**);

}

**else** {

**if** ($idParamNext.text != **null**) {

names.add(names.**new** Name($idParamNext.text, $type.text, $idParamNext.line));

}

}

**if** (names.isExist($assignParamNext.idName)) {

errors.add(**"line "** + $assignParamNext.idLine + **": name "** + $assignParamNext.idName + **" duplicated"**);

}

**else** {

**if** ($assignParamNext.idName != **null**) {

names.add(names.**new** Name($assignParamNext.idName, $type.text, $assignParamNext.idLine));

}

}

**if** ($assignParam.expressionType != **null** && $assignParam.idName != **null** && !$assignParam.expressionType.equals(names.get($assignParam.idName).getType())) {

errors.add(**"line "** + $assignParam.idLine + **": name "** + $assignParam.idName + **" type is mismatched"**);

}

**if** ($assignParam.expressionType != **null** && $assignParamNext.idName != **null** && !$assignParamNext.expressionType.equals(names.get($assignParamNext.idName).getType())) {

errors.add(**"line "** + $assignParamNext.idLine + **": name "** + $assignParamNext.idName + **" type is mismatched"**);

}

}

;

for\_operator

: 'for' '(' type? assign {

**if** (names.isExist($assign.idName)) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" duplicated"**);

}

**else** {

**if** ($assign.text != **null**) {

names.add(names.**new** Name($assign.idName, $type.text, $assign.idLine));

}

}

**if** (!$type.text.equals(**"int"**)) {

errors.add(**"line "** + $assign.idLine + **": type of counter is mismatched"**);

}

**if** (!$assign.expressionType.equals(names.get($assign.idName).getType())) {

errors.add(**"line "** + $assign.idLine + **": name "** + $assign.idName + **" type is mismatched"**);

}

}

';' logical\_condition[$assign.idLine] ';' incrementation ')' '{' {

**int** instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"if"** + **" ("** + $logical\_condition.instructionNumber + **") ("** + (instructionNumber + 2) + **")"**);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"goto"** + **" "** + **"("** + **"X"** + **")"**);

}

operator+

'}' {

instructions.add(**"("** + getInstructionNumber() + **") "** + **"+"** + **" "** + $incrementation.idName + **" 1"**);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"assign"** + **" "** + $incrementation.idName + **" ("** + (instructionNumber - 1) + **")"**);

instructionNumber = getInstructionNumber();

instructions.add(**"("** + instructionNumber + **") "** + **"goto"** + **" ("** + ($assign.instructionNumber + 1) + **")"**);

**for** (**int** i = instructions.size() - 1; i > 0; i--) {

**if** (instructions.get(i).indexOf(**"X"**) != -1) {

String temp = instructions.get(i);

**int** position = temp.indexOf(**"X"**);

temp = temp.substring(0, position) + Integer.toString(instructionNumber + 1) + temp.substring(position + 1);

instructions.set(i, temp);

}

}

}

;

function\_call\_operator

: function\_call ';'

;

logical\_condition [**int** operatorLine] **returns** [**int** instructionNumber]

: expressionFirst = expression logical\_link expressionSecond = expression {

**if** ($expressionFirst.expressionType != **null**) {

**if** (!$expressionFirst.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionFirst.text);

}

**else** {

**if** ($expressionSecond.expressionType != **null**) {

**if** (!$expressionSecond.expressionType.equals(**"int"**)) {

errors.add(**"line "** + $operatorLine + **": wrong condition operator type of "** + $expressionSecond.text);

}

**else** {

String resultString = **""**;

$instructionNumber = getInstructionNumber();

**if** ($expressionFirst.expressionClass.equals(**"id"**) ||

$expressionFirst.expressionClass.equals(**"const\_string"**) ||

$expressionFirst.expressionClass.equals(**"const\_int"**)) {

**if** ($expressionFirst.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionFirst.text);

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + **"D0"** + constants.size() + **" "**;

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" "** + $expressionFirst.text + **" "**;

}

}

**else** {

resultString = **"("** + $instructionNumber + **") "** + $logical\_link.text + **" ("** + $expressionFirst.instructionNumber + **") "**;

}

**if** ($expressionSecond.expressionClass.equals(**"id"**) ||

$expressionSecond.expressionClass.equals(**"const\_string"**) ||

$expressionSecond.expressionClass.equals(**"const\_int"**)) {

$instructionNumber = getInstructionNumber();

**if** ($expressionSecond.expressionClass.equals(**"const\_string"**)) {

constants.add($expressionSecond.text);

resultString += **"D0"** + constants.size();

}

**else** {

resultString += $expressionSecond.text;

}

instructions.add(resultString);

}

**else** {

resultString += **"("** + $expressionSecond.instructionNumber + **")"**;

instructions.add(resultString);

}

}

}

}

}

}

;

logical\_link

: '==' | '!=' | '>' | '<' | '>=' | '<='

;

operator : variable\_declaration | assign\_operator | for\_operator | function\_call\_operator

;

program : (operator)+

;