Министерство образования Республики Беларусь

Учреждение образования «Белорусский Государственный Университет Информатики и Радиоэлектроники»

Кафедра «Информатика»

Лабораторная работа № 3

«Семантический анализатор»

по курсу «Теория трансляции»

**Выполнил:**

Волчецкий А. М. гр. 253504

**Проверила:**

Шиманский В.В.

Минск 2015

**Задание**

Написать семантический анализатор для ограниченного множества операторов языка С#.

**Теоретические сведения**

### **Особенности семантики и семантического анализа**

Следующий шаг анализа текста программы – семантический, существенно отличается от двух предыдущих – лексического и синтаксического. И дело не столько в том, что фаза семантического анализа реализуется не формальными, а **содержательными**методами (т.е. на данный момент нет универсальных математических моделей и формальных средств описания «смысла» программы). Лексический и синтаксический анализ имеют дело со **структурными,** т.е. внешними, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе. Для любого, имеющего опыт практического программирования, ясно, что формальные конструкции языка дают описание свойств и действий над**внутренними объектами**, с которыми имеет дело программа. Для начала перечислим все, что их касается и лежит на поверхности:

* ·большинство объектов  являются **именованными.**Имя объекта позволяет его идентифицировать, существуют различные области действия имен, соглашения об именах, различные умолчания и т.п.. Все это относится к семантике;
* виды, сложность и набор характеристик объектов различаются в разных языках программирования и сильно зависят от области приложения языка (в этом смысле семантика языков программирования более разнообразна, нежели синтаксис и лексика). Например, классический Си, ориентированный на максимальное приближение к архитектуре компьютера, работает с такими объектами, как **типы данных, переменные, функции**. Все они имеют различные свойства и характеристики. Например, переменная характеризуется именем, типом данных, размерностью, областью действия, временем жизни, текущим значением;
* объекты связаны между собой (ссылаются друг на друга). В том же Си переменная ссылается на описание того типа данных, к которому она относится, далее производный тип данных ссылается на базовый и т.п.. Можно сказать, что семантика программы во внутреннем представлении выглядит как система взаимосвязанных объектов;
* внутреннее представление семантики программы не совсем удачно называется **семантическими таблицами.**На самом деле структура данных, соответствующая представлению семантики, может быть любой. Термин**«таблицы»** говорит о том, что имеются множества объектов различных типов, для каждого из которых заведена отдельная таблица, но нельзя забывать, что элементы различных таблиц связаны между собой. Наиболее близкий термин для описания подобной системы – **база данных.**

**Семантика программы** – внутренняя модель (база данных) множества именованных объектов, с которыми работает программа, с описанием их свойств,  характеристик и связей.

Теперь, когда у нас есть представление о синтаксической фазе, можно оценить ее центральную роль в организации процесса трансляции. Только на уровне синтаксиса текст программы представляет собой **единое структурное целое –**любое предложение языка сводится к единственному начальному нетерминалу **Z.** Лексические единицы, как известно, вообще независимы друг от друга и являются терминальными символами синтаксиса. Семантика программы тоже не обладает структурной целостностью и представлена фрагментарно, но при этом связана с синтаксисом следующим образом:

* один и тот же семантический объект (например, переменная) может встречаться в различных, синтаксически несвязанных частях программы;
* синтаксические конструкции *описаний, определений и объявлений* являются **источником семантики**объектов программы, они «заявляют» о существовании объектов и задают их свойства;
* синтаксические конструкции, связанные с *действиями, выполняемыми над объектами*, являются **потребителями семантики,**их интерпретация, корректность, «смысл» зависят от семантических свойств объекта. Забегая вперед, можно заметить, что заключительная фаза трансляции (генерация кода, интерпретация) может рассматриваться как особые семантические действия, производимые над объектами;
* первичным источником семантики является лексический анализ. Некоторые из лексем (например, идентификаторы и константы) наряду с **классом лексемы**(обозначение выходной единицы лексического анализа), т.е. **символом**(обозначение той же единицы на входе синтаксического анализатора) имеют **значение.**Значением лексемы является сама распознанная цепочка литер, она и представляет**семантическую составляющую лексемы,**которая попадает в семантические таблицы;
* лексемы, или то же самое, что терминальные символы входной строки (в терминах синтаксического анализа), ссылаются в семантические таблицах на свою семантику. В дальнейшем каждый промежуточный нетерминал также ссылается на собственную семантику. При этом любое правило преобразует семантику терминалов и нетерминалов правой части в семантику нетерминала левой части при помощи назначенной правилу **семантической процедуры.**Таким образом**,**формирование семантической составляющей связано с движением **снизу вверх**по синтаксическому дереву, от вершин – потомков к предкам.
* семантическая процедура, получая ссылки на семантику терминальных и нетерминальных символов правой части, формирует семантику результата и размещает ее в семантических таблицах, связывая ее через ссылку с нетерминалом левой части. Таким образом, семантическая составляющая транслятора тоже является  фрагментарной (набор семантических процедур, соответствующих правилам грамматики) и объединяется в единое целое только в рамках синтаксического дерева.

Задача семантического анализа, т.е. «описания смысла» фразы относится скорее к области  искусственного интеллекта. Ее неформализуемость означает, что она  **не имеет формальных средств описания,**например, языков**.**Следовательно, семантическая модель языка разрабатывается в каждом случае уникально, здесь отсутствует общий подход, а имеет место набор частных решений и рекомендаций. Отсюда и уникальность семантики языка.

**Код программы**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Diagnostics;

namespace Irony.Parsing.Construction {

// Methods constructing LALR automaton.

// See \_about\_parser\_construction.txt file in this folder for important comments

internal partial class ParserDataBuilder {

LanguageData \_language;

ParserData \_data;

Grammar \_grammar;

ParserStateHash \_stateHash = new ParserStateHash();

internal ParserDataBuilder(LanguageData language) {

\_language = language;

\_grammar = \_language.Grammar;

}

public void Build() {

\_stateHash.Clear();

\_data = \_language.ParserData;

CreateParserStates();

var itemsNeedLookaheads = GetReduceItemsInInadequateState();

ComputeTransitions(itemsNeedLookaheads);

ComputeLookaheads(itemsNeedLookaheads);

ComputeStatesExpectedTerminals();

ComputeConflicts();

ApplyHints();

HandleUnresolvedConflicts();

CreateRemainingReduceActions();

//Create error action - if it is not created yet by some hint or custom code

if (\_data.ErrorAction == null)

\_data.ErrorAction = new ErrorRecoveryParserAction();

}//method

#region Creating parser states

private void CreateParserStates() {

var grammarData = \_language.GrammarData;

//1. Base automaton: create states for main augmented root for the grammar

\_data.InitialState = CreateInitialState(grammarData.AugmentedRoot);

ExpandParserStateList(0);

CreateAcceptAction(\_data.InitialState, grammarData.AugmentedRoot);

//2. Expand automaton: add parser states from additional roots

foreach(var augmRoot in grammarData.AugmentedSnippetRoots) {

var initialState = CreateInitialState(augmRoot);

ExpandParserStateList(\_data.States.Count - 1); //start with just added state - it is the last state in the list

CreateAcceptAction(initialState, augmRoot);

}

}

private void CreateAcceptAction(ParserState initialState, NonTerminal augmentedRoot) {

var root = augmentedRoot.Productions[0].RValues[0];

var shiftAction = initialState.Actions[root] as ShiftParserAction;

var shiftOverRootState = shiftAction.NewState;

shiftOverRootState.Actions[\_grammar.Eof] = new AcceptParserAction();

}

private ParserState CreateInitialState(NonTerminal augmentedRoot) {

//for an augmented root there is an initial production "Root' -> .Root"; so we need the LR0 item at 0 index

var iniItemSet = new LR0ItemSet();

iniItemSet.Add(augmentedRoot.Productions[0].LR0Items[0]);

var initialState = FindOrCreateState(iniItemSet);

var rootNt = augmentedRoot.Productions[0].RValues[0] as NonTerminal;

\_data.InitialStates[rootNt] = initialState;

return initialState;

}

private void ExpandParserStateList(int initialIndex) {

// Iterate through states (while new ones are created) and create shift transitions and new states

for (int index = initialIndex; index < \_data.States.Count; index++) {

var state = \_data.States[index];

//Get all possible shifts

foreach (var term in state.BuilderData.ShiftTerms) {

var shiftItems = state.BuilderData.ShiftItems.SelectByCurrent(term);

//Get set of shifted cores and find/create target state

var shiftedCoreItems = shiftItems.GetShiftedCores();

var newState = FindOrCreateState(shiftedCoreItems);

//Create shift action

var newAction = new ShiftParserAction(term, newState);

state.Actions[term] = newAction;

//Link items in old/new states

foreach (var shiftItem in shiftItems) {

shiftItem.ShiftedItem = newState.BuilderData.AllItems.FindByCore(shiftItem.Core.ShiftedItem);

}//foreach shiftItem

}//foreach term

} //for index

}//method

private ParserState FindOrCreateState(LR0ItemSet coreItems) {

string key = ComputeLR0ItemSetKey(coreItems);

ParserState state;

if (\_stateHash.TryGetValue(key, out state))

return state;

//create new state

state = new ParserState("S" + \_data.States.Count);

state.BuilderData = new ParserStateData(state, coreItems);

\_data.States.Add(state);

\_stateHash[key] = state;

return state;

}

#endregion

#region Compute transitions, lookbacks, lookaheads

//We compute only transitions that are really needed to compute lookaheads in inadequate states.

// We start with reduce items in inadequate state and find their lookbacks - this is initial list of transitions.

// Then for each transition in the list we check if it has items with nullable tails; for those items we compute

// lookbacks - these are new or already existing transitons - and so on, we repeat the operation until no new transitions

// are created.

private void ComputeTransitions(LRItemSet forItems) {

var newItemsNeedLookbacks = forItems;

while(newItemsNeedLookbacks.Count > 0) {

var newTransitions = CreateLookbackTransitions(newItemsNeedLookbacks);

newItemsNeedLookbacks = SelectNewItemsThatNeedLookback(newTransitions);

}

}

private LRItemSet SelectNewItemsThatNeedLookback(TransitionList transitions) {

//Select items with nullable tails that don't have lookbacks yet

var items = new LRItemSet();

foreach(var trans in transitions)

foreach(var item in trans.Items)

if (item.Core.TailIsNullable && item.Lookbacks.Count == 0) //only if it does not have lookbacks yet

items.Add(item);

return items;

}

private LRItemSet GetReduceItemsInInadequateState() {

var result = new LRItemSet();

foreach(var state in \_data.States) {

if (state.BuilderData.IsInadequate)

result.UnionWith(state.BuilderData.ReduceItems);

}

return result;

}

private TransitionList CreateLookbackTransitions(LRItemSet sourceItems) {

var newTransitions = new TransitionList();

//Build set of initial cores - this is optimization for performance

//We need to find all initial items in all states that shift into one of sourceItems

// Each such initial item would have the core from the "initial" cores set that we build from source items.

var iniCores = new LR0ItemSet();

foreach(var sourceItem in sourceItems)

iniCores.Add(sourceItem.Core.Production.LR0Items[0]);

//find

foreach(var state in \_data.States) {

foreach(var iniItem in state.BuilderData.InitialItems) {

if (!iniCores.Contains(iniItem.Core)) continue;

var iniItemNt = iniItem.Core.Production.LValue; // iniItem's non-terminal (left side of production)

Transition lookback = null; // local var for lookback - transition over iniItemNt

var currItem = iniItem; // iniItem is initial item for all currItem's in the shift chain.

while (currItem != null) {

if(sourceItems.Contains(currItem)) {

// We create transitions lazily, only when we actually need them. Check if we have iniItem's transition

// in local variable; if not, get it from state's transitions table; if not found, create it.

if(lookback == null && !state.BuilderData.Transitions.TryGetValue(iniItemNt, out lookback)) {

lookback = new Transition(state, iniItemNt);

newTransitions.Add(lookback);

}

//Now for currItem, either add trans to Lookbacks, or "include" it into currItem.Transition

// We need lookbacks ONLY for final items; for non-Final items we need proper Include lists on transitions

if (currItem.Core.IsFinal)

currItem.Lookbacks.Add(lookback);

else // if (currItem.Transition != null)

// Note: looks like checking for currItem.Transition is redundant - currItem is either:

// - Final - always the case for the first run of this method;

// - it has a transition after the first run, due to the way we select sourceItems list

// in SelectNewItemsThatNeedLookback (by transitions)

currItem.Transition.Include(lookback);

}//if

//move to next item

currItem = currItem.ShiftedItem;

}//while

}//foreach iniItem

}//foreach state

return newTransitions;

}

private void ComputeLookaheads(LRItemSet forItems) {

foreach(var reduceItem in forItems) {

// Find all source states - those that contribute lookaheads

var sourceStates = new ParserStateSet();

foreach(var lookbackTrans in reduceItem.Lookbacks) {

sourceStates.Add(lookbackTrans.ToState);

sourceStates.UnionWith(lookbackTrans.ToState.BuilderData.ReadStateSet);

foreach(var includeTrans in lookbackTrans.Includes) {

sourceStates.Add(includeTrans.ToState);

sourceStates.UnionWith(includeTrans.ToState.BuilderData.ReadStateSet);

}//foreach includeTrans

}//foreach lookbackTrans

//Now merge all shift terminals from all source states

foreach(var state in sourceStates)

reduceItem.Lookaheads.UnionWith(state.BuilderData.ShiftTerminals);

//Remove SyntaxError - it is pseudo terminal

if (reduceItem.Lookaheads.Contains(\_grammar.SyntaxError))

reduceItem.Lookaheads.Remove(\_grammar.SyntaxError);

//Sanity check

if (reduceItem.Lookaheads.Count == 0)

\_language.Errors.Add(GrammarErrorLevel.InternalError, reduceItem.State, "Reduce item '{0}' in state {1} has no lookaheads.", reduceItem.Core, reduceItem.State);

}//foreach reduceItem

}//method

#endregion

#region Analyzing and resolving conflicts

private void ComputeConflicts() {

foreach(var state in \_data.States) {

if(!state.BuilderData.IsInadequate)

continue;

//first detect conflicts

var stateData = state.BuilderData;

stateData.Conflicts.Clear();

var allLkhds = new BnfTermSet();

//reduce/reduce --------------------------------------------------------------------------------------

foreach(var item in stateData.ReduceItems) {

foreach(var lkh in item.Lookaheads) {

if(allLkhds.Contains(lkh))

state.BuilderData.Conflicts.Add(lkh);

allLkhds.Add(lkh);

}//foreach lkh

}//foreach item

//shift/reduce ---------------------------------------------------------------------------------------

foreach(var term in stateData.ShiftTerminals)

if(allLkhds.Contains(term)) {

stateData.Conflicts.Add(term);

}

}

}//method

private void ApplyHints() {

foreach (var state in \_data.States) {

var stateData = state.BuilderData;

//Add automatic precedence hints

if (stateData.Conflicts.Count > 0)

foreach (var conflict in stateData.Conflicts.ToList())

if (conflict.Flags.IsSet(TermFlags.IsOperator)) {

//Find any reduce item with this lookahead and add PrecedenceHint

var reduceItem = stateData.ReduceItems.SelectByLookahead(conflict).First();

var precHint = new PrecedenceHint();

reduceItem.Core.Hints.Add(precHint);

}

// Apply (activate) hints - these should resolve conflicts as well

foreach (var item in state.BuilderData.AllItems)

foreach (var hint in item.Core.Hints)

hint.Apply(\_language, item);

}//foreach

}//method

//Resolve to default actions

private void HandleUnresolvedConflicts() {

foreach (var state in \_data.States) {

if (state.BuilderData.Conflicts.Count == 0)

continue;

var shiftReduceConflicts = state.BuilderData.GetShiftReduceConflicts();

var reduceReduceConflicts = state.BuilderData.GetReduceReduceConflicts();

var stateData = state.BuilderData;

if (shiftReduceConflicts.Count > 0)

\_language.Errors.Add(GrammarErrorLevel.Conflict, state, Resources.ErrSRConflict, state, shiftReduceConflicts.ToString());

if (reduceReduceConflicts.Count > 0)

\_language.Errors.Add(GrammarErrorLevel.Conflict, state, Resources.ErrRRConflict, state, reduceReduceConflicts.ToString());

//Create default actions for these conflicts. For shift-reduce, default action is shift, and shift action already

// exist for all shifts from the state, so we don't need to do anything, only report it

//For reduce-reduce create reduce actions for the first reduce item (whatever comes first in the set).

foreach (var conflict in reduceReduceConflicts) {

var reduceItems = stateData.ReduceItems.SelectByLookahead(conflict);

var firstProd = reduceItems.First().Core.Production;

var action = new ReduceParserAction(firstProd);

state.Actions[conflict] = action;

}

//stateData.Conflicts.Clear(); -- do not clear them, let the set keep the auto-resolved conflicts, may find more use for this later

}

}

#endregion

#region final actions: creating remaining reduce actions, computing expected terminals, cleaning up state data

//Create reduce actions for states with a single reduce item (and no shifts)

private void CreateRemainingReduceActions() {

foreach (var state in \_data.States) {

if (state.DefaultAction != null) continue;

var stateData = state.BuilderData;

if (stateData.ShiftItems.Count == 0 && stateData.ReduceItems.Count == 1) {

state.DefaultAction = ReduceParserAction.Create(stateData.ReduceItems.First().Core.Production);

continue; //next state; if we have default reduce action, we don't need to fill actions dictionary for lookaheads

}

//create actions

foreach (var item in state.BuilderData.ReduceItems) {

var action = ReduceParserAction.Create(item.Core.Production);

foreach (var lkh in item.Lookaheads) {

if (state.Actions.ContainsKey(lkh)) continue;

state.Actions[lkh] = action;

}

}//foreach item

}//foreach state

}

//Note that for states with a single reduce item the result is empty

private void ComputeStatesExpectedTerminals() {

foreach (var state in \_data.States) {

state.ExpectedTerminals.UnionWith(state.BuilderData.ShiftTerminals);

//Add lookaheads from reduce items

foreach (var reduceItem in state.BuilderData.ReduceItems)

state.ExpectedTerminals.UnionWith(reduceItem.Lookaheads);

RemoveTerminals(state.ExpectedTerminals, \_grammar.SyntaxError, \_grammar.Eof);

}//foreach state

}

private void RemoveTerminals(TerminalSet terms, params Terminal[] termsToRemove) {

foreach(var termToRemove in termsToRemove)

if (terms.Contains(termToRemove)) terms.Remove(termToRemove);

}

public void CleanupStateData() {

foreach (var state in \_data.States)

state.ClearData();

}

#endregion

#region Utilities: ComputeLR0ItemSetKey

//Parser states are distinguished by the subset of kernel LR0 items.

// So when we derive new LR0-item list by shift operation,

// we need to find out if we have already a state with the same LR0Item list.

// We do it by looking up in a state hash by a key - [LR0 item list key].

// Each list's key is a concatenation of items' IDs separated by ','.

// Before producing the key for a list, the list must be sorted;

// thus we garantee one-to-one correspondence between LR0Item sets and keys.

// And of course, we count only kernel items (with dot NOT in the first position).

public static string ComputeLR0ItemSetKey(LR0ItemSet items) {

if (items.Count == 0) return string.Empty;

//Copy non-initial items to separate list, and then sort it

LR0ItemList itemList = new LR0ItemList();

foreach (var item in items)

itemList.Add(item);

//quick shortcut

if (itemList.Count == 1)

return itemList[0].ID.ToString();

itemList.Sort(CompareLR0Items); //Sort by ID

//now build the key

StringBuilder sb = new StringBuilder(100);

foreach (LR0Item item in itemList) {

sb.Append(item.ID);

sb.Append(",");

}//foreach

return sb.ToString();

}

private static int CompareLR0Items(LR0Item x, LR0Item y) {

if (x.ID < y.ID) return -1;

if (x.ID == y.ID) return 0;

return 1;

}

#endregion

internal static StringSet ComputeGroupedExpectedSetForState(Grammar grammar, ParserState state) {

var terms = new TerminalSet();

terms.UnionWith(state.ExpectedTerminals);

var result = new StringSet();

//Eliminate no-report terminals

foreach (var group in grammar.TermReportGroups)

if (group.GroupType == TermReportGroupType.DoNotReport)

terms.ExceptWith(group.Terminals);

//Add normal and operator groups

foreach (var group in grammar.TermReportGroups)

if ((group.GroupType == TermReportGroupType.Normal || group.GroupType == TermReportGroupType.Operator) &&

terms.Overlaps(group.Terminals)) {

result.Add(group.Alias);

terms.ExceptWith(group.Terminals);

}

//Add remaining terminals "as is"

foreach (var terminal in terms)

result.Add(terminal.ErrorAlias);

return result;

}

}//class

}//namespace

Содержание анализируемого кода:

using System;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int i = 1;

string s = Convert.ToString(i);

Console.WriteLine(s);

int j = int.Parse(s);

bool result = j.Equals(i);

Console.Write(result);

Console.ReadKey();

}

}

}

Результат выполнения семантического анализа

Обработка ошибок несоответсвия типов:

