Министерство образования Республики Беларусь

Учреждение образования «Белорусский Государственный Университет Информатики и Радиоэлектроники»

Кафедра «Информатика»

Лабораторная работа № 4

«Простейший компилятор»

по курсу «Теория трансляции»

**Выполнил:**

Волчецкий А. М. гр. 253504

**Проверила:**

Шиманский В.В.

Минск 2015

**Задание**

Написать простейший компилятор множества операторов языка С#.

**Теоретические сведения**

**Ассе́мблер** (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *assembler* — сборщик) — транслятор исходного текста программы, написанной на [языке ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0), в программу на [машинном языке](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4).

Как и сам язык, ассемблеры, как правило, специфичны для конкретной [архитектуры](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%B0), [операционной системы](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) и варианта синтаксиса языка. Вместе с тем существуют мультиплатформенные или вовсе универсальные (точнее, ограниченно-универсальные, потому что на языке низкого уровня нельзя написать аппаратно-независимые программы) ассемблеры, которые могут работать на разных платформах и операционных системах. Среди последних можно также выделить группу *кросс-ассемблеров*, способных собирать машинный код и исполняемые модули (файлы) для других архитектур и ОС.

Ассемблирование может быть не первым и не последним этапом на пути получения исполнимого модуля программы. Так, многие компиляторы с языков программирования высокого уровня выдают результат в виде программы на языке ассемблера, которую в дальнейшем обрабатывает ассемблер. Также результатом ассемблирования может быть не исполнимый, а [*объектный* модуль](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C), содержащий разрозненные блоки машинного кода и данных программы, из которого (или из нескольких объектных модулей) в дальнейшем с помощью [редактора связей](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B4%D0%B0%D0%BA%D1%82%D0%BE%D1%80_%D1%81%D0%B2%D1%8F%D0%B7%D0%B5%D0%B9) может быть получен [исполнимый файл](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D0%BC%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C).

Common Intermediate Language (сокращённо CIL) — «высокоуровневый ассемблер» виртуальной машины [.NET](https://ru.wikipedia.org/wiki/Microsoft_.NET). Промежуточный язык, разработанный фирмой «[Microsoft](https://ru.wikipedia.org/wiki/Microsoft)» для платформы [.NET Framework](https://ru.wikipedia.org/wiki/Microsoft_.NET). [JIT](https://ru.wikipedia.org/wiki/JIT)-компилятор CIL является частью [CLR](https://ru.wikipedia.org/wiki/Common_Language_Runtime) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *common language runtime*) — общей [среды выполнения](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B5%D0%B4%D0%B0_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F) программ, написанных на [языках .NET](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_.NET-%D1%8F%D0%B7%D1%8B%D0%BA%D0%BE%D0%B2). Ранее язык назывался «*Microsoft Intermediate Language* *(MSIL)*», однако был переименован для создания стандарта «ECMA-335».

Все [компиляторы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80), поддерживающие платформу [.NET](https://ru.wikipedia.org/wiki/Microsoft_.NET), должны транслировать код с [языков высокого уровня платформы .NET](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_.NET-%D1%8F%D0%B7%D1%8B%D0%BA%D0%BE%D0%B2) на язык CIL. В частности, код на языке CIL генерируют все компиляторы .NET фирмы «[Microsoft](https://ru.wikipedia.org/wiki/Microsoft)», входящие в [среду разработки](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) «[Microsoft Visual Studio](https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio)» ([C#](https://ru.wikipedia.org/wiki/C_Sharp), [Managed C++](https://ru.wikipedia.org/wiki/Managed_C_Plus_Plus), [Visual Basic .NET](https://ru.wikipedia.org/wiki/Visual_Basic_.NET), [Visual J# .NET](https://ru.wikipedia.org/wiki/Visual_J_Sharp_.NET)).

По синтаксису и мнемонике язык CIL напоминает [язык ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0). Его можно рассматривать как ассемблер виртуальной машины .NET. В то же время язык CIL содержит некоторые достаточно высокоуровневые конструкции, повышающие его уровень по сравнению с ассемблером для любой реально существующей машины, и писать код непосредственно на CIL легче, чем на ассемблере для реальных машин. Поэтому CIL можно рассматривать как своеобразный «высокоуровневый ассемблер».

Язык CIL также нередко называют просто IL от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *intermediate language* — буквально «промежуточный язык».

Синтаксис и мнемоника языка CIL описываются стандартом «ECMA-335». Спецификация CIL является составной частью более общей спецификации — спецификации [CLI](https://ru.wikipedia.org/wiki/Common_Language_Infrastructure) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *common language infrastructure*).

**Код программы**

Был написан код, который принимает информацию от синтаксического и семантического анализаторов и при отсутствии ошибок преобразовывает код согласно дереву разбора в IL команды, выполнение которых в дальнейшем и является результатом работы программы.

interface ICodeGenContext : IMemberInfo, ISignatureGen, IDelayedDefinition, IDelayedCompletion

{

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Design", "CA1024:UsePropertiesWhereAppropriate", Justification = "Typical implementation invokes XxxBuilder.GetILGenerator() which is a method as well.")]

ILGenerator GetILGenerator();

Type OwnerType { get; }

}

public partial class CodeGen

{

ILGenerator il;

ICodeGenContext context;

ConstructorGen cg;

bool chainCalled = false;

bool reachable = true;

bool hasRetVar = false, hasRetLabel = false;

LocalBuilder retVar = null;

Label retLabel;

Stack<Block> blocks = new Stack<Block>();

internal ILGenerator IL { get { return il; } }

internal ICodeGenContext Context { get { return context; } }

internal CodeGen(ICodeGenContext context)

{

this.context = context;

this.cg = context as ConstructorGen;

il = context.GetILGenerator();

}

#region Arguments

public Operand This()

{

if (context.IsStatic)

throw new InvalidOperationException(Properties.Messages.ErrCodeStaticThis);

return new \_Arg(0, context.OwnerType);

}

public Operand Base()

{

if (context.IsStatic)

return new \_StaticTarget(context.OwnerType.BaseType);

else

return new \_Base(context.OwnerType.BaseType);

}

int \_ThisOffset { get { return context.IsStatic ? 0 : 1; } }

public Operand PropertyValue()

{

Type[] parameterTypes = context.ParameterTypes;

return new \_Arg(\_ThisOffset + parameterTypes.Length - 1, parameterTypes[parameterTypes.Length - 1]);

}

public Operand Arg(string name)

{

ParameterGen param = context.GetParameterByName(name);

return new \_Arg(\_ThisOffset + param.Position - 1, param.Type);

}

#endregion

#region Locals

public Operand Local()

{

return new \_Local(this);

}

public Operand Local(Operand init)

{

Operand var = Local();

Assign(var, init);

return var;

}

public Operand Local(Type type)

{

return new \_Local(this, type);

}

public Operand Local(Type type, Operand init)

{

Operand var = Local(type);

Assign(var, init);

return var;

}

#endregion

bool HasReturnValue

{

get

{

Type returnType = context.ReturnType;

return returnType != null && returnType != typeof(void);

}

}

void EnsureReturnVariable()

{

if (hasRetVar)

return;

retLabel = il.DefineLabel();

if (HasReturnValue)

retVar = il.DeclareLocal(context.ReturnType);

hasRetVar = true;

}

public bool IsCompleted

{

get

{

return blocks.Count == 0 && !reachable && hasRetVar == hasRetLabel;

}

}

internal void Complete()

{

if (blocks.Count > 0)

throw new InvalidOperationException(Properties.Messages.ErrOpenBlocksRemaining);

if (reachable)

{

if (HasReturnValue)

throw new InvalidOperationException(string.Format(null, Properties.Messages.ErrMethodMustReturnValue, context));

else

Return();

}

if (hasRetVar && !hasRetLabel)

{

il.MarkLabel(retLabel);

if (retVar != null)

il.Emit(OpCodes.Ldloc, retVar);

il.Emit(OpCodes.Ret);

hasRetLabel = true;

}

}

class \_Base : \_Arg

{

public \_Base(Type type) : base(0, type) { }

internal override bool SuppressVirtual

{

get

{

return true;

}

}

}

class \_Arg : Operand

{

ushort index;

Type type;

public \_Arg(int index, Type type)

{

this.index = checked((ushort)index);

this.type = type;

}

internal override void EmitGet(CodeGen g)

{

g.EmitLdargHelper(index);

if (IsReference)

g.EmitLdindHelper(Type);

}

internal override void EmitSet(CodeGen g, Operand value, bool allowExplicitConversion)

{

if (IsReference)

{

g.EmitLdargHelper(index);

g.EmitStindHelper(Type, value, allowExplicitConversion);

}

else

{

g.EmitGetHelper(value, Type, allowExplicitConversion);

g.EmitStargHelper(index);

}

}

internal override void EmitAddressOf(CodeGen g)

{

if (IsReference)

{

g.EmitLdargHelper(index);

}

else

{

if (index <= byte.MaxValue)

g.il.Emit(OpCodes.Ldarga\_S, (byte)index);

else

g.il.Emit(OpCodes.Ldarga, index);

}

}

bool IsReference { get { return type.IsByRef; } }

public override Type Type

{

get

{

return IsReference ? type.GetElementType() : type;

}

}

internal override bool TrivialAccess

{

get

{

return true;

}

}

}

internal class \_Local : Operand

{

CodeGen owner;

LocalBuilder var;

Block scope;

Type t, tHint;

public \_Local(CodeGen owner)

{

this.owner = owner;

scope = owner.GetBlockForVariable();

}

public \_Local(CodeGen owner, Type t)

{

this.owner = owner; this.t = t;

scope = owner.GetBlockForVariable();

}

public \_Local(CodeGen owner, LocalBuilder var)

{

this.owner = owner;

this.var = var;

this.t = var.LocalType;

}

void CheckScope(CodeGen g)

{

if (g != owner)

throw new InvalidOperationException(Properties.Messages.ErrInvalidVariableContext);

if (scope != null && !owner.blocks.Contains(scope))

throw new InvalidOperationException(Properties.Messages.ErrInvalidVariableScope);

}

internal override void EmitGet(CodeGen g)

{

CheckScope(g);

if (var == null)

throw new InvalidOperationException(Properties.Messages.ErrUninitializedVarAccess);

g.il.Emit(OpCodes.Ldloc, var);

}

internal override void EmitSet(CodeGen g, Operand value, bool allowExplicitConversion)

{

CheckScope(g);

if (t == null)

t = value.Type;

if (var == null)

var = g.il.DeclareLocal(t);

g.EmitGetHelper(value, t, allowExplicitConversion);

g.il.Emit(OpCodes.Stloc, var);

}

internal override void EmitAddressOf(CodeGen g)

{

CheckScope(g);

if (var == null)

{

RequireType();

var = g.il.DeclareLocal(t);

}

g.il.Emit(OpCodes.Ldloca, var);

}

public override Type Type

{

get

{

RequireType();

return t;

}

}

void RequireType()

{

if (t == null)

{

if (tHint != null)

t = tHint;

else

throw new InvalidOperationException(Properties.Messages.ErrUntypedVarAccess);

}

}

internal override bool TrivialAccess

{

get

{

return true;

}

}

internal override void AssignmentHint(Operand op)

{

if (tHint == null)

tHint = Operand.GetType(op);

}

}

class \_StaticTarget : Operand

{

Type t;

public \_StaticTarget(Type t) { this.t = t; }

public override Type Type

{

get

{

return t;

}

}

internal override bool IsStaticTarget

{

get

{

return true;

}

}

}

internal void EmitLdargHelper(ushort index)

{

OpCode opCode;

switch (index)

{

case 0: opCode = OpCodes.Ldarg\_0; break;

case 1: opCode = OpCodes.Ldarg\_1; break;

case 2: opCode = OpCodes.Ldarg\_2; break;

case 3: opCode = OpCodes.Ldarg\_3; break;

default:

if (index <= byte.MaxValue)

il.Emit(OpCodes.Ldarg\_S, (byte)index);

else

il.Emit(OpCodes.Ldarg, index);

return;

}

il.Emit(opCode);

}

internal void EmitStargHelper(ushort index)

{

if (index <= byte.MaxValue)

il.Emit(OpCodes.Starg\_S, (byte)index);

else

il.Emit(OpCodes.Starg, index);

}

internal void EmitLdelemHelper(Type elementType)

{

OpCode op;

if (elementType.IsPrimitive)

{

switch (Type.GetTypeCode(elementType))

{

case TypeCode.SByte:

case TypeCode.Boolean:

op = OpCodes.Ldelem\_I1;

break;

case TypeCode.Byte:

op = OpCodes.Ldelem\_U1;

break;

case TypeCode.Int16:

op = OpCodes.Ldelem\_I2;

break;

case TypeCode.UInt16:

case TypeCode.Char:

op = OpCodes.Ldelem\_U2;

break;

case TypeCode.Int32:

op = OpCodes.Ldelem\_I4;

break;

case TypeCode.UInt32:

op = OpCodes.Ldelem\_U4;

break;

case TypeCode.Int64:

case TypeCode.UInt64:

op = OpCodes.Ldelem\_I8;

break;

case TypeCode.Single:

op = OpCodes.Ldelem\_R4;

break;

case TypeCode.Double:

op = OpCodes.Ldelem\_R8;

break;

default:

throw new NotSupportedException();

}

}

else if (elementType.IsValueType)

{

il.Emit(OpCodes.Ldelema, elementType);

il.Emit(OpCodes.Ldobj, elementType);

return;

}

else

op = OpCodes.Ldelem\_Ref;

il.Emit(op);

}

internal static OpCode GetStelemOpCode(Type elementType)

{

if (elementType.IsPrimitive)

{

switch (Type.GetTypeCode(elementType))

{

case TypeCode.Byte:

case TypeCode.SByte:

case TypeCode.Boolean:

return OpCodes.Stelem\_I1;

case TypeCode.Int16:

case TypeCode.UInt16:

case TypeCode.Char:

return OpCodes.Stelem\_I2;

case TypeCode.Int32:

case TypeCode.UInt32:

return OpCodes.Stelem\_I4;

case TypeCode.Int64:

case TypeCode.UInt64:

return OpCodes.Stelem\_I8;

case TypeCode.Single:

return OpCodes.Stelem\_R4;

case TypeCode.Double:

return OpCodes.Stelem\_R8;

default:

throw new NotSupportedException();

}

}

else if (elementType.IsValueType)

return OpCodes.Stobj;

else

return OpCodes.Stelem\_Ref;

}

internal void EmitStelemHelper(Type elementType, Operand element, bool allowExplicitConversion)

{

OpCode op = GetStelemOpCode(elementType);

if (op == OpCodes.Stobj)

il.Emit(OpCodes.Ldelema, elementType);

EmitGetHelper(element, elementType, allowExplicitConversion);

if (op == OpCodes.Stobj)

il.Emit(OpCodes.Stobj, elementType);

else

il.Emit(op);

}

internal void EmitLdindHelper(Type type)

{

OpCode op;

if (type.IsPrimitive)

{

switch (Type.GetTypeCode(type))

{

case TypeCode.SByte:

case TypeCode.Boolean:

op = OpCodes.Ldind\_I1;

break;

case TypeCode.Byte:

op = OpCodes.Ldind\_U1;

break;

case TypeCode.Int16:

op = OpCodes.Ldind\_I2;

break;

case TypeCode.UInt16:

case TypeCode.Char:

op = OpCodes.Ldind\_U2;

break;

case TypeCode.Int32:

op = OpCodes.Ldind\_I4;

break;

case TypeCode.UInt32:

op = OpCodes.Ldind\_U4;

break;

case TypeCode.Int64:

case TypeCode.UInt64:

op = OpCodes.Ldind\_I8;

break;

case TypeCode.Single:

op = OpCodes.Ldind\_R4;

break;

case TypeCode.Double:

op = OpCodes.Ldind\_R8;

break;

default:

throw new NotSupportedException();

}

}

else if (type.IsValueType)

{

il.Emit(OpCodes.Ldobj, type);

return;

}

else

op = OpCodes.Ldind\_Ref;

il.Emit(op);

}

internal static OpCode GetStindOpCode(Type type)

{

if (type.IsPrimitive)

{

switch (Type.GetTypeCode(type))

{

case TypeCode.Byte:

case TypeCode.SByte:

case TypeCode.Boolean:

return OpCodes.Stind\_I1;

case TypeCode.Int16:

case TypeCode.UInt16:

case TypeCode.Char:

return OpCodes.Stind\_I2;

case TypeCode.Int32:

case TypeCode.UInt32:

return OpCodes.Stind\_I4;

case TypeCode.Int64:

case TypeCode.UInt64:

return OpCodes.Stind\_I8;

case TypeCode.Single:

return OpCodes.Stind\_R4;

case TypeCode.Double:

return OpCodes.Stind\_R8;

default:

throw new NotSupportedException();

}

}

else if (type.IsValueType)

return OpCodes.Stobj;

else

return OpCodes.Stind\_Ref;

}

internal void EmitStindHelper(Type type, Operand value, bool allowExplicitConversion)

{

OpCode op = GetStindOpCode(type);

EmitGetHelper(value, type, allowExplicitConversion);

if (op == OpCodes.Stobj)

il.Emit(OpCodes.Stobj, type);

else

il.Emit(op);

}

internal void EmitI4Helper(int value)

{

OpCode code;

switch (value)

{

case 0: code = OpCodes.Ldc\_I4\_0; break;

case 1: code = OpCodes.Ldc\_I4\_1; break;

case 2: code = OpCodes.Ldc\_I4\_2; break;

case 3: code = OpCodes.Ldc\_I4\_3; break;

case 4: code = OpCodes.Ldc\_I4\_4; break;

case 5: code = OpCodes.Ldc\_I4\_5; break;

case 6: code = OpCodes.Ldc\_I4\_6; break;

case 7: code = OpCodes.Ldc\_I4\_7; break;

case 8: code = OpCodes.Ldc\_I4\_8; break;

case -1: code = OpCodes.Ldc\_I4\_M1; break;

default:

if (value >= sbyte.MinValue && value <= sbyte.MaxValue)

il.Emit(OpCodes.Ldc\_I4\_S, (sbyte)value);

else

il.Emit(OpCodes.Ldc\_I4, value);

return;

}

il.Emit(code);

}

internal void EmitI8Helper(long value, bool signed)

{

if (value >= int.MinValue && value <= uint.MaxValue)

{

EmitI4Helper((int)value);

if (value < 0 && signed)

il.Emit(OpCodes.Conv\_I8);

else

il.Emit(OpCodes.Conv\_U8);

}

else

il.Emit(OpCodes.Ldc\_I8, value);

}

internal void EmitConvHelper(TypeCode to)

{

OpCode op;

switch (to)

{

case TypeCode.SByte:

op = OpCodes.Conv\_I1; break;

case TypeCode.Byte:

op = OpCodes.Conv\_U1; break;

case TypeCode.Int16:

op = OpCodes.Conv\_I2; break;

case TypeCode.UInt16:

case TypeCode.Char:

op = OpCodes.Conv\_U2; break;

case TypeCode.Int32:

op = OpCodes.Conv\_I4; break;

case TypeCode.UInt32:

op = OpCodes.Conv\_U4; break;

case TypeCode.Int64:

op = OpCodes.Conv\_I8; break;

case TypeCode.UInt64:

op = OpCodes.Conv\_U8; break;

case TypeCode.Single:

op = OpCodes.Conv\_R4; break;

case TypeCode.Double:

op = OpCodes.Conv\_R8; break;

default:

throw new NotSupportedException();

}

il.Emit(op);

}

internal void EmitGetHelper(Operand op, Type desiredType, bool allowExplicitConversion)

{

if (desiredType.IsByRef)

{

if (op.Type != desiredType.GetElementType())

throw new InvalidOperationException(Properties.Messages.ErrByRefTypeMismatch);

op.EmitAddressOf(this);

return;

}

if ((object)op == null)

{

if (desiredType.IsValueType)

throw new ArgumentNullException("op");

il.Emit(OpCodes.Ldnull);

return;

}

op.EmitGet(this);

Convert(op, desiredType, allowExplicitConversion);

}

internal void EmitCallHelper(MethodBase mth, Operand target)

{

MethodInfo mi = mth as MethodInfo;

if (mi != null)

{

bool suppressVirtual = ((object)target != null && target.SuppressVirtual) || mi.IsStatic;

if (!suppressVirtual && (object)target != null && target.Type.IsValueType && mi.IsVirtual)

{

il.Emit(OpCodes.Constrained, target.Type);

}

il.Emit(suppressVirtual ? OpCodes.Call : OpCodes.Callvirt, mi);

return;

}

ConstructorInfo ci = mth as ConstructorInfo;

if (ci != null)

{

il.Emit(OpCodes.Call, ci);

return;

}

throw new ArgumentException(Properties.Messages.ErrInvalidMethodBase, "mth");

}

internal void Convert(Operand op, Type to, bool allowExplicit)

{

Conversion conv = allowExplicit ? Conversion.GetExplicit(op, to, false) : Conversion.GetImplicit(op, to, false);

conv.Emit(this, (object)op == null ? null : op.Type, to);

}

}

public void Assign(Operand target, Operand value)

{

Assign(target, value, false);

}

public void Assign(Operand target, Operand value, bool allowExplicitConversion)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.Assign(value, allowExplicitConversion).Emit(this);

}

public void AssignAdd(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignAdd(value).Emit(this);

}

public void AssignSubtract(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignSubtract(value).Emit(this);

}

public void AssignMultiply(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignMultiply(value).Emit(this);

}

public void AssignDivide(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignDivide(value).Emit(this);

}

public void AssignModulus(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignModulus(value).Emit(this);

}

public void AssignAnd(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignAnd(value).Emit(this);

}

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1702:CompoundWordsShouldBeCasedCorrectly", Justification = "Checked, OK")]

public void AssignOr(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignOr(value).Emit(this);

}

public void AssignXor(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignXor(value).Emit(this);

}

public void AssignLeftShift(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignLeftShift(value).Emit(this);

}

public void AssignRightShift(Operand target, Operand value)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.AssignRightShift(value).Emit(this);

}

public void Increment(Operand target)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.Increment().Emit(this);

}

public void Decrement(Operand target)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.Decrement().Emit(this);

}

#endregion

#region Constructor chaining

public void InvokeThis(params Operand[] args)

{

if (cg == null)

throw new InvalidOperationException(Properties.Messages.ErrConstructorOnlyCall);

if (chainCalled)

throw new InvalidOperationException(Properties.Messages.ErrConstructorAlreadyChained);

ApplicableFunction other = TypeInfo.FindConstructor(cg.Type, args);

il.Emit(OpCodes.Ldarg\_0);

other.EmitArgs(this, args);

il.Emit(OpCodes.Call, (ConstructorInfo)other.Method.Member);

chainCalled = true;

}

public void InvokeBase(params Operand[] args)

{

if (cg == null)

throw new InvalidOperationException(Properties.Messages.ErrConstructorOnlyCall);

if (chainCalled)

throw new InvalidOperationException(Properties.Messages.ErrConstructorAlreadyChained);

if (cg.Type.TypeBuilder.IsValueType)

throw new InvalidOperationException(Properties.Messages.ErrStructNoBaseCtor);

ApplicableFunction other = TypeInfo.FindConstructor(cg.Type.BaseType, args);

if (other == null)

throw new InvalidOperationException(Properties.Messages.ErrMissingConstructor);

il.Emit(OpCodes.Ldarg\_0);

other.EmitArgs(this, args);

il.Emit(OpCodes.Call, (ConstructorInfo)other.Method.Member);

chainCalled = true;

il.Emit(OpCodes.Ldarg\_0);

il.Emit(OpCodes.Call, cg.Type.CommonConstructor().GetMethodBuilder());

}

#endregion

void BeforeStatement()

{

if (!reachable)

throw new InvalidOperationException(Properties.Messages.ErrCodeNotReachable);

if (cg != null && !chainCalled && !cg.Type.TypeBuilder.IsValueType)

InvokeBase();

}

void DoInvoke(Operand invocation)

{

BeforeStatement();

invocation.EmitGet(this);

if (invocation.Type != typeof(void))

il.Emit(OpCodes.Pop);

}

#region Invocation

public void Invoke(Type target, string method)

{

Invoke(target, method, Operand.EmptyArray);

}

public void Invoke(Type target, string method, params Operand[] args)

{

DoInvoke(Static.Invoke(target, method, args));

}

public void Invoke(Operand target, string method)

{

Invoke(target, method, Operand.EmptyArray);

}

public void Invoke(Operand target, string method, params Operand[] args)

{

if ((object)target == null)

throw new ArgumentNullException("target");

DoInvoke(target.Invoke(method, args));

}

public void InvokeDelegate(Operand targetDelegate)

{

InvokeDelegate(targetDelegate, Operand.EmptyArray);

}

public void InvokeDelegate(Operand targetDelegate, params Operand[] args)

{

if ((object)targetDelegate == null)

throw new ArgumentNullException("targetDelegate");

DoInvoke(targetDelegate.InvokeDelegate(args));

}

public void WriteLine(params Operand[] args)

{

Invoke(typeof(Console), "WriteLine", args);

}

#endregion

#region Event subscription

public void SubscribeEvent(Operand target, string eventName, Operand handler)

{

if ((object)target == null)

throw new ArgumentNullException("target");

if ((object)handler == null)

throw new ArgumentNullException("handler");

IMemberInfo evt = TypeInfo.FindEvent(target.Type, eventName, target.IsStaticTarget);

MethodInfo mi = ((EventInfo)evt.Member).GetAddMethod();

if (!target.IsStaticTarget)

target.EmitGet(this);

handler.EmitGet(this);

EmitCallHelper(mi, target);

}

public void UnsubscribeEvent(Operand target, string eventName, Operand handler)

{

if ((object)target == null)

throw new ArgumentNullException("target");

if ((object)handler == null)

throw new ArgumentNullException("handler");

IMemberInfo evt = TypeInfo.FindEvent(target.Type, eventName, target.IsStaticTarget);

MethodInfo mi = ((EventInfo)evt.Member).GetRemoveMethod();

if (!target.IsStaticTarget)

target.EmitGet(this);

handler.EmitGet(this);

EmitCallHelper(mi, target);

}

#endregion

public void InitObj(Operand target)

{

if ((object)target == null)

throw new ArgumentNullException("target");

BeforeStatement();

target.EmitAddressOf(this);

il.Emit(OpCodes.Initobj, target.Type);

}

#region Flow Control

interface IBreakable

{

Label GetBreakTarget();

}

interface IContinuable

{

Label GetContinueTarget();

}

public void Break()

{

BeforeStatement();

bool useLeave = false;

foreach (Block blk in blocks)

{

ExceptionBlock xb = blk as ExceptionBlock;

if (xb != null)

{

if (xb.IsFinally)

throw new InvalidOperationException(Properties.Messages.ErrInvalidFinallyBranch);

useLeave = true;

}

IBreakable brkBlock = blk as IBreakable;

if (brkBlock != null)

{

il.Emit(useLeave ? OpCodes.Leave : OpCodes.Br, brkBlock.GetBreakTarget());

reachable = false;

return;

}

}

throw new InvalidOperationException(Properties.Messages.ErrInvalidBreak);

}

public void Continue()

{

BeforeStatement();

bool useLeave = false;

foreach (Block blk in blocks)

{

ExceptionBlock xb = blk as ExceptionBlock;

if (xb != null)

{

if (xb.IsFinally)

throw new InvalidOperationException(Properties.Messages.ErrInvalidFinallyBranch);

useLeave = true;

}

IContinuable cntBlock = blk as IContinuable;

if (cntBlock != null)

{

il.Emit(useLeave ? OpCodes.Leave : OpCodes.Br, cntBlock.GetContinueTarget());

reachable = false;

return;

}

}

throw new InvalidOperationException(Properties.Messages.ErrInvalidContinue);

}

public void Return()

{

if (context.ReturnType != null && context.ReturnType != typeof(void))

throw new InvalidOperationException(Properties.Messages.ErrMethodMustReturnValue);

BeforeStatement();

ExceptionBlock xb = GetAnyTryBlock();

if (xb == null)

{

il.Emit(OpCodes.Ret);

}

else if (xb.IsFinally)

{

throw new InvalidOperationException(Properties.Messages.ErrInvalidFinallyBranch);

}

else

{

EnsureReturnVariable();

il.Emit(OpCodes.Leave, retLabel);

}

reachable = false;

}

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Design", "CA1011:ConsiderPassingBaseTypesAsParameters", Justification = "'Operand' required as type to use provided implicit conversions")]

public void Return(Operand value)

{

if (context.ReturnType == null || context.ReturnType == typeof(void))

throw new InvalidOperationException(Properties.Messages.ErrVoidMethodReturningValue);

BeforeStatement();

EmitGetHelper(value, context.ReturnType, false);

ExceptionBlock xb = GetAnyTryBlock();

if (xb == null)

{

il.Emit(OpCodes.Ret);

}

else if (xb.IsFinally)

{

throw new InvalidOperationException(Properties.Messages.ErrInvalidFinallyBranch);

}

else

{

EnsureReturnVariable();

il.Emit(OpCodes.Stloc, retVar);

il.Emit(OpCodes.Leave, retLabel);

}

reachable = false;

}

public void Throw()

{

BeforeStatement();

il.Emit(OpCodes.Rethrow);

reachable = false;

}

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Design", "CA1011:ConsiderPassingBaseTypesAsParameters", Justification = "'Operand' required as type to use provided implicit conversions")]

public void Throw(Operand exception)

{

BeforeStatement();

EmitGetHelper(exception, typeof(Exception), false);

il.Emit(OpCodes.Throw);

reachable = false;

}

public void For(IStatement init, Operand test, IStatement iterator)

{

Begin(new LoopBlock(init, test, iterator));

}

public void While(Operand test)

{

Begin(new LoopBlock(null, test, null));

}

public Operand ForEach(Type elementType, Operand expression)

{

ForeachBlock fb = new ForeachBlock(elementType, expression);

Begin(fb);

return fb.Element;

}

public void If(Operand condition)

{

Begin(new IfBlock(condition));

}

public void Else()

{

IfBlock ifBlk = GetBlock() as IfBlock;

if (ifBlk == null)

throw new InvalidOperationException(Properties.Messages.ErrElseWithoutIf);

blocks.Pop();

Begin(new ElseBlock(ifBlk));

}

public void Try()

{

Begin(new ExceptionBlock());

}

ExceptionBlock GetTryBlock()

{

ExceptionBlock tryBlk = GetBlock() as ExceptionBlock;

if (tryBlk == null)

throw new InvalidOperationException(Properties.Messages.ErrInvalidExceptionStatement);

return tryBlk;

}

ExceptionBlock GetAnyTryBlock()

{

foreach (Block blk in blocks)

{

ExceptionBlock tryBlk = blk as ExceptionBlock;

if (tryBlk != null)

return tryBlk;

}

return null;

}

public Operand Catch(Type exceptionType)

{

return GetTryBlock().BeginCatch(exceptionType);

}

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Naming", "CA1702:CompoundWordsShouldBeCasedCorrectly", Justification = "Intentional")]

public void CatchAll()

{

GetTryBlock().BeginCatchAll();

}

public void Fault()

{

GetTryBlock().BeginFault();

}

public void Finally()

{

GetTryBlock().BeginFinally();

}

public void Switch(Operand expression)

{

Begin(new SwitchBlock(expression));

}

SwitchBlock GetSwitchBlock()

{

SwitchBlock switchBlk = GetBlock() as SwitchBlock;

if (switchBlk == null)

throw new InvalidOperationException(Properties.Messages.ErrInvalidSwitchStatement);

return switchBlk;

}

public void Case(object value)

{

IConvertible conv = value as IConvertible;

if (conv == null)

throw new ArgumentException(Properties.Messages.ErrArgMustImplementIConvertible, "value");

GetSwitchBlock().Case(conv);

}

public void DefaultCase()

{

GetSwitchBlock().Case(null);

}

#endregion

Block GetBlock()

{

if (blocks.Count == 0)

return null;

return blocks.Peek();

}

Block GetBlockForVariable()

{

if (blocks.Count == 0)

return null;

Block b = blocks.Peek();

b.EnsureScope();

return b;

}

void Begin(Block b)

{

blocks.Push(b);

b.g = this;

b.Begin();

}

public void End()

{

if (blocks.Count == 0)

throw new InvalidOperationException(Properties.Messages.ErrNoOpenBlocks);

blocks.Peek().End();

blocks.Pop();

}

abstract class Block

{

bool hasScope;

internal CodeGen g;

public void EnsureScope()

{

if (!hasScope)

{

g.il.BeginScope();

hasScope = true;

}

}

protected void EndScope()

{

if (hasScope)

{

g.il.EndScope();

hasScope = false;

}

}

public void Begin()

{

BeginImpl();

}

public void End()

{

EndImpl();

EndScope();

}

protected abstract void BeginImpl();

protected abstract void EndImpl();

}

class IfBlock : Block

{

Operand condition;

public IfBlock(Operand condition)

{

if (condition.Type != typeof(bool))

this.condition = condition.IsTrue();

else

this.condition = condition;

}

Label lbSkip;

protected override void BeginImpl()

{

g.BeforeStatement();

lbSkip = g.il.DefineLabel();

condition.EmitBranch(g, BranchSet.Inverse, lbSkip);

}

protected override void EndImpl()

{

g.il.MarkLabel(lbSkip);

g.reachable = true;

}

}

class ElseBlock : Block

{

IfBlock ifBlk;

Label lbSkip;

bool canSkip;

public ElseBlock(IfBlock ifBlk)

{

this.ifBlk = ifBlk;

}

protected override void BeginImpl()

{

if (canSkip = g.reachable)

{

lbSkip = g.il.DefineLabel();

g.il.Emit(OpCodes.Br, lbSkip);

}

ifBlk.End();

}

protected override void EndImpl()

{

if (canSkip)

{

g.il.MarkLabel(lbSkip);

g.reachable = true;

}

}

}

class LoopBlock : Block, IBreakable, IContinuable

{

IStatement init;

Operand test;

IStatement iter;

public LoopBlock(IStatement init, Operand test, IStatement iter)

{

this.init = init;

this.test = test;

this.iter = iter;

if (test.Type != typeof(bool))

test = test.IsTrue();

}

Label lbLoop, lbTest, lbEnd, lbIter;

bool endUsed = false, iterUsed = false;

protected override void BeginImpl()

{

g.BeforeStatement();

lbLoop = g.il.DefineLabel();

lbTest = g.il.DefineLabel();

if (init != null)

init.Emit(g);

g.il.Emit(OpCodes.Br, lbTest);

g.il.MarkLabel(lbLoop);

}

protected override void EndImpl()

{

if (iter != null)

{

if (iterUsed)

g.il.MarkLabel(lbIter);

iter.Emit(g);

}

g.il.MarkLabel(lbTest);

test.EmitBranch(g, BranchSet.Normal, lbLoop);

if (endUsed)

g.il.MarkLabel(lbEnd);

g.reachable = true;

}

public Label GetBreakTarget()

{

if (!endUsed)

{

lbEnd = g.il.DefineLabel();

endUsed = true;

}

return lbEnd;

}

public Label GetContinueTarget()

{

if (iter == null)

return lbTest;

if (!iterUsed)

{

lbIter = g.il.DefineLabel();

iterUsed = true;

}

return lbIter;

}

}

class ForeachBlock : Block, IBreakable, IContinuable

{

Type elementType;

Operand collection;

public ForeachBlock(Type elementType, Operand collection)

{

this.elementType = elementType;

this.collection = collection;

}

Operand enumerator, element;

Label lbLoop, lbTest, lbEnd;

bool endUsed = false;

public Operand Element { get { return element; } }

protected override void BeginImpl()

{

g.BeforeStatement();

enumerator = g.Local();

lbLoop = g.il.DefineLabel();

lbTest = g.il.DefineLabel();

if (typeof(IEnumerable).IsAssignableFrom(collection.Type))

collection = collection.Cast(typeof(IEnumerable));

g.Assign(enumerator, collection.Invoke("GetEnumerator"));

g.il.Emit(OpCodes.Br, lbTest);

g.il.MarkLabel(lbLoop);

element = g.Local(elementType);

g.Assign(element, enumerator.Property("Current"), true);

}

protected override void EndImpl()

{

g.il.MarkLabel(lbTest);

enumerator.Invoke("MoveNext").EmitGet(g);

g.il.Emit(OpCodes.Brtrue, lbLoop);

if (endUsed)

g.il.MarkLabel(lbEnd);

g.reachable = true;

}

public Label GetBreakTarget()

{

if (!endUsed)

{

lbEnd = g.il.DefineLabel();

endUsed = true;

}

return lbEnd;

}

public Label GetContinueTarget()

{

return lbTest;

}

}

class ExceptionBlock : Block

{

bool endReachable = false;

bool isFinally = false;

protected override void BeginImpl()

{

g.il.BeginExceptionBlock();

}

public void BeginCatchAll()

{

EndScope();

if (g.reachable)

endReachable = true;

g.il.BeginCatchBlock(typeof(object));

g.il.Emit(OpCodes.Pop);

g.reachable = true;

}

public Operand BeginCatch(Type t)

{

EndScope();

if (g.reachable)

endReachable = true;

g.il.BeginCatchBlock(t);

LocalBuilder lb = g.il.DeclareLocal(t);

g.il.Emit(OpCodes.Stloc, lb);

g.reachable = true;

return new \_Local(g, lb);

}

public void BeginFault()

{

EndScope();

g.il.BeginFaultBlock();

g.reachable = true;

isFinally = true;

}

public void BeginFinally()

{

EndScope();

g.il.BeginFinallyBlock();

g.reachable = true;

isFinally = true;

}

protected override void EndImpl()

{

g.il.EndExceptionBlock();

g.reachable = endReachable;

}

public bool IsFinally { get { return isFinally; } }

}

class SwitchBlock : Block, IBreakable

{

static Type[] validTypes = {

typeof(sbyte), typeof(byte), typeof(short), typeof(ushort), typeof(int), typeof(uint), typeof(long), typeof(ulong), typeof(char), typeof(string)

};

static MethodInfo strCmp = typeof(string).GetMethod("Equals", BindingFlags.Public | BindingFlags.Static,

null, new Type[] { typeof(string), typeof(string) }, null);

Operand expression;

Conversion conv;

Type govType;

Label lbDecision;

Label lbEnd;

Label lbDefault;

LocalBuilder exp;

bool defaultExists = false;

bool endReachable = false;

SortedList<IComparable, Label> cases = new SortedList<IComparable, Label>();

public SwitchBlock(Operand expression)

{

this.expression = expression;

Type exprType = expression.Type;

if (Array.IndexOf(validTypes, exprType) != -1)

govType = exprType;

else if (exprType.IsEnum)

govType = Enum.GetUnderlyingType(exprType);

else

{

foreach (Type t in validTypes)

{

Conversion tmp = Conversion.GetImplicit(expression, t, false);

if (tmp.IsValid)

{

if (conv == null)

{

conv = tmp;

govType = t;

}

else

throw new AmbiguousMatchException(Properties.Messages.ErrAmbiguousSwitchExpression);

}

}

}

}

protected override void BeginImpl()

{

lbDecision = g.il.DefineLabel();

lbDefault = lbEnd = g.il.DefineLabel();

expression.EmitGet(g);

if (conv != null)

conv.Emit(g, expression.Type, govType);

exp = g.il.DeclareLocal(govType);

g.il.Emit(OpCodes.Stloc, exp);

g.il.Emit(OpCodes.Br, lbDecision);

g.reachable = false;

}

public void Case(IConvertible value)

{

bool duplicate;

IComparable val = value == null ? null : (IComparable)value.ToType(govType, System.Globalization.CultureInfo.InvariantCulture);

if (value == null)

duplicate = defaultExists;

else

duplicate = cases.ContainsKey(val);

if (duplicate)

throw new InvalidOperationException(Properties.Messages.ErrDuplicateCase);

if (g.reachable)

g.il.Emit(OpCodes.Br, lbEnd);

EndScope();

Label lb = g.il.DefineLabel();

g.il.MarkLabel(lb);

if (value == null)

{

defaultExists = true;

lbDefault = lb;

}

else

{

cases[val] = lb;

}

g.reachable = true;

}

static int Diff(IConvertible val1, IConvertible val2)

{

ulong diff;

switch (val1.GetTypeCode())

{

case TypeCode.UInt64:

diff = val2.ToUInt64(null) - val1.ToUInt64(null);

break;

case TypeCode.Int64:

diff = (ulong)(val2.ToInt64(null) - val1.ToInt64(null));

break;

case TypeCode.UInt32:

diff = val2.ToUInt32(null) - val1.ToUInt32(null);

break;

default:

diff = (ulong)(val2.ToInt32(null) - val1.ToInt32(null));

break;

}

if (diff >= int.MaxValue)

return int.MaxValue;

else

return (int)diff;

}

void Finish(List<Label> labels)

{

switch (labels.Count)

{

case 0: break;

case 1:

g.il.Emit(OpCodes.Beq, labels[0]);

break;

default:

g.il.Emit(OpCodes.Sub);

g.il.Emit(OpCodes.Switch, labels.ToArray());

break;

}

}

void EmitValue(IConvertible val)

{

switch (val.GetTypeCode())

{

case TypeCode.UInt64:

g.EmitI8Helper(unchecked((long)val.ToUInt64(null)), false);

break;

case TypeCode.Int64:

g.EmitI8Helper(val.ToInt64(null), true);

break;

case TypeCode.UInt32:

g.EmitI4Helper(unchecked((int)val.ToUInt64(null)));

break;

default:

g.EmitI4Helper(val.ToInt32(null));

break;

}

}

protected override void EndImpl()

{

if (g.reachable)

{

g.il.Emit(OpCodes.Br, lbEnd);

endReachable = true;

}

EndScope();

g.il.MarkLabel(lbDecision);

if (govType == typeof(string))

{

foreach (KeyValuePair<IComparable, Label> kvp in cases)

{

g.il.Emit(OpCodes.Ldloc, exp);

g.il.Emit(OpCodes.Ldstr, kvp.Key.ToString());

g.il.Emit(OpCodes.Call, strCmp);

g.il.Emit(OpCodes.Brtrue, kvp.Value);

}

}

else

{

bool first = true;

IConvertible prev = null;

List<Label> labels = new List<Label>();

foreach (KeyValuePair<IComparable, Label> kvp in cases)

{

IConvertible val = (IConvertible)kvp.Key;

if (prev != null)

{

int diff = Diff(prev, val);

if (diff > 3)

{

Finish(labels);

labels.Clear();

prev = null;

first = true;

}

else

while (diff-- > 1)

labels.Add(lbDefault);

}

if (first)

{

g.il.Emit(OpCodes.Ldloc, exp);

EmitValue(val);

first = false;

}

labels.Add(kvp.Value);

prev = val;

}

Finish(labels);

}

if (lbDefault != lbEnd)

g.il.Emit(OpCodes.Br, lbDefault);

g.il.MarkLabel(lbEnd);

g.reachable = endReachable;

}

public Label GetBreakTarget()

{

endReachable = true;

return lbEnd;

}

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 преобразования в IL и вывод программы:

.maxstack 2

.locals init ([0] int32 i,

[1] string s,

[2] int32 j,

[3] bool result)

nop

ldc.i4.1

stloc.0

ldloc.0

call string [mscorlib]System.Convert::ToString(int32)

stloc.1

ldloc.1

call void [mscorlib]System.Console::WriteLine(string)

nop

ldloc.1

call int32 [mscorlib]System.Int32::Parse(string)

stloc.2

ldloca.s j

ldloc.0

call instance bool [mscorlib]System.Int32::Equals(int32)

stloc.3

ldloc.3

call void [mscorlib]System.Console::Write(bool)

nop

call valuetype [mscorlib]System.ConsoleKeyInfo [mscorlib]System.Console::ReadKey()

pop

ret

