UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

ESPECIFICACÃO E COMPILAÇÃO DE UMA LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A PLATAFORMA MICROSOFT .NET

GUSTAVO ZADROZNY LEYENDECKER

BLUMENAU
2005

2005/2-11

GUSTAVO ZADROZNY LEYENDECKER

ESPECIFICAÇÃO E COMPILAÇÃO DE UMA LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A PLATAFORMA MICROSOFT .NET

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Joyce Martins, Mestre – Orientadora

BLUMENAU
2005

2005/2-11

ESPECIFICAÇÃO E COMPILAÇÃO DE UMA LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS PARA A PLATAFORMA MICROSOFT .NET

Por

GUSTAVO ZADROZNY LEYENDECKER

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Joyce Martins Orientadora, FURB
Membro:	Prof. Jomi Fred Hübner – FURB
Membro:	Prof. José Roque Voltolini da Silva – FURB

Blumenau, 15 de dezembro de 2005

Dedico este trabalho à minha família que me apoiou durante a realização do mesmo.

AGRADECIMENTOS

À minha orientadora, Joyce, por ter aceitado auxiliar no desenvolvimento deste trabalho mesmo que em cima da hora.

À minha família, namorada e amigos por terem me apoiado e incentivado.

Applications programming is a race between software engineers, who strive to produce idiot-proof programs, and the Universe which strives to produce bigger idiots. So far the Universe is winning.

Autor desconhecido

RESUMO

Este trabalho apresenta a especificação de uma linguagem de programação orientada a objetos para a plataforma Microsoft .NET e o desenvolvimento de um compilador para esta linguagem. O objetivo do compilador é gerar código *Microsoft Intermediate Language* (MSIL) para que possa ser executado a partir da *Common Language Runtime* (CLR). Para especificar a linguagem são utilizadas definições regulares e a notação *Backus-Naur Form* estendida (EBNF). No desenvolvimento do compilador é utilizada a ferramenta JavaCCCS para fazer as verificações léxicas e sintáticas e construir a *Abstract Syntax Tree* (AST). O tratamento de contexto ou análise semântica é implementado em C# .NET. A geração de código é feita através da varredura da AST criando uma árvore de operações da linguagem MSIL. Por fim, o compilador executa o montador de MSIL da Microsoft (o ILAsm) com o intuito de gerar um arquivo executável ou uma biblioteca de classes para a plataforma .NET.

Palavras-chave: Compilador. *Microsoft Intermediate Language*. Linguagens de programação. Orientação a objetos.

ABSTRACT

This work presents the specification of an object oriented programming language for the .NET platform and the development of a compiler for this language. The goal of the compiler is to generate Microsoft Intermediate Language (MSIL) to run in the Common Language Runtime (CLR). To specify the language regular definitions and the extended Backus-Naur Form (EBNF) notation are used. In the compiler development, the JavaCCCS tool is used to create both the lexical and syntactical analyzers and load the Abstract Syntax Tree (AST). The semantical checks are implemented using C# .NET. The code generation is made by converting the AST to a MSIL operation tree. At last, the compiler executes the MSIL builder from Microsoft (ILAsm) to create a .NET platform executable file or a class library.

Key-words: Compiler. Microsoft Intermediate Language. Programming languages. Object orientation.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

AST – *Abstract Syntax Tree*

BD – Banco de Dados

BNF – *Backus-Naur Form*

CLR – *Common Language Runtime*

CLS – *Common Language Specification*

CTS – *Common Type System*

EA – *Enterprise Architect*

EBNF – *Extended Backus-Naur Form*

GAC – *Global Assembly Cache*

IDE – *Integrated Development Environment*

LALR – *Look Ahead Left-to-right Rightmost derivation*

LL – *Left-to-right Leftmost derivation*

LP – Linguagem de Programação

LR – *Left-to-right Rightmost derivation*

MSIL – *Microsoft Intermediate Language*

NLS - *National Language Support*

PE – *Portable Executable*

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

WMI - *Windows Management Instrumentation*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Uma linguagem de programação (LP) provê ao desenvolvedor de sistemas uma variedade de facilidades para a criação de soluções de tecnologia da informação. Varejão (2004, p. 3) diz que as LPs visam tornar mais produtivo o trabalho dos programadores levando em consideração a manutenção e a qualidade do software.

Segundo Wilson e Clark (1993, p. 14), as primeiras linguagens que surgiram eram bastante primitivas. Utilizavam números para representar operações, locais de armazenamento e registradores especiais. Não possuíam *loops* nem mesmo números em ponto flutuante.

Em virtude da evolução da computação, os sistemas tornaram-se mais complexos e o uso de LPs simples reduzia a produtividade dos programadores. Para resolver este problema, surgiram as linguagens de programação de alto nível. Entre as linguagens de programação modernas, é possível observar o uso do paradigma de orientação a objetos.

Nesse sentido, o objetivo deste trabalho é implementar um compilador para uma linguagem orientada a objetos que gere código para a máquina virtual da plataforma Microsoft .NET. Esta plataforma, de acordo com a Microsoft Corporation (2005), disponibiliza todas as ferramentas e tecnologias necessárias para a construção de aplicações. Ela dispõe de um modelo de programação consistente e independente de linguagem através de todas as camadas de um software.

> Umas das grandes jogadas da .NET é sua capacidade de trabalhar com a multilinguagem permitindo com isso que todos os desenvolvedores possam trabalhar com a linguagem a qual possuem mais afinidade. Isso é possível graças ao fato de que os módulos do sistema implementados em uma determinada linguagem se comunicam tranqüilamente com os implementados em outras, desde que ambas sejam suportadas pela .NET. Pode-se, por exemplo, utilizar um componente do Visual Basic .NET em sistemas desenvolvidos em C#. (BORBA, 2004, p. 1).

Esta independência de linguagem da plataforma .NET deve-se ao fato de todas as LPs utilizarem o mesmo código intermediário, o *Microsoft Intermediate Language* (MSIL). A linguagem proposta não será diferente já que seu compilador irá gerar código MSIL. Em

outras palavras, o compilador irá gerar um *assembly* (módulo executável .NET) que poderá referenciar *assemblies* gerados a partir qualquer linguagem .NET e utilizar as classes contidas dentro dele. Da mesma forma, *assemblies* desenvolvidos nestas linguagens também poderão referenciar o *assembly* gerado e utilizar suas classes. Assim sendo, cada camada de aplicação em um software multi-camadas, ou em inglês *n-tier*, poderá ser implementada em uma linguagem diferente. Por exemplo, em um software de três camadas, as camadas de acesso a dados e interface serão desenvolvidas em C# enquanto a camada de negócio será desenvolvida fazendo uso da LP proposta. Segundo Mahmoodi (2005), o desenvolvimento em três camadas tem como vantagens reusabilidade da lógica de negócio, facilidade de manutenção, entre outras.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um compilador para uma linguagem de programação orientada a objetos, que gere código MSIL.

Os objetivos específicos do trabalho são:

a) especificar uma linguagem de programação orientada a objetos semelhante ao C#;
b) implementar os analisadores léxico, sintático e semântico e o gerador de código intermediário;
c) disponibilizar novas funcionalidades que não existem no C#, como o uso de expressões relacionais e aritméticas para trabalhar com tipos de data e hora, e intervalo de tempo.

1.2 ESTRUTURA DO TRABALHO

O trabalho está dividido em quatro capítulos. No capítulo seguinte é feito um levantamento teórico sobre os assuntos abordados no trabalho. O terceiro capítulo trata da especificação da linguagem e desenvolvimento do compilador. O quarto capítulo contém uma breve discussão sobre o trabalho, incluindo as conclusões sobre o que foi apresentado nos capítulos anteriores e sugestões de extensões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados conceitos, técnicas e ferramentas relevantes para o desenvolvimento de compiladores e especificação de linguagens de programação focando a compilação de linguagens orientadas a objetos. Também é descrita a plataforma Microsoft .NET. Por fim, são apresentados trabalhos correlatos.

2.1 LINGUAGENS DE PROGRAMAÇÃO

Computadores são ferramentas que solucionam problemas através de um programa ou *software* que é escrito em uma linguagem de programação (WILSON; CLARK, 1993, p. 2). As linguagens de programação modernas devem abstrair a complexidade do funcionamento de um computador.

Segundo Lisbôa (2004), as linguagens de programação podem ser classificadas de acordo com os seguintes modelos:

a) imperativo: os comandos são executados de forma seqüencial e definem operações sobre dados. O modelo imperativo é baseado na arquitetura Von Neumann e tem como recursos centrais variáveis e instruções de atribuição e repetição. Pertencem a este modelo as linguagens procedurais e as orientadas a objetos;
b) declarativo: não possuem comandos, apenas “roteiros” que definem o que deve ser computado. Ou seja, os programas são definidos por funções (linguagens funcionais) ou axiomas lógicos (linguagens lógicas).

Varejão (2004, p. 6-12) cita as seguintes propriedades como desejáveis em uma LP:

a) legibilidade: esta propriedade diz respeito à facilidade de ler e entender o que se pretende com determinado programa. O uso do comando *goto*, por exemplo, pode

causar problemas de legibilidade em um código fonte;

b) redigibilidade: é a capacidade que uma LP tem de abstrair certos detalhes de funcionamento de determinado comando ou expressão;
c) confiabilidade: diz respeito aos mecanismos fornecidos por uma determinada LP para construção de programas confiáveis. Isto implica nos mecanismos que uma linguagem fornece para tratar erros em tempo de compilação e execução;
d) eficiência: faz referência à performance da LP. Embora gerar código otimizado seja uma responsabilidade do compilador, algumas definições na linguagem podem impactar negativamente em sua performance;
e) facilidade de aprendizado: linguagens que tem como característica permitir a utilização de diferentes comandos para fazer a mesma coisa tendem a ser difíceis de aprender;
f) ortogonalidade: “[...] uma LP é tão mais ortogonal quanto menor for o número de exceções aos seus padrões regulares.” (VAREJÃO, 2004, p. 10). Isto quer dizer que uma certa sintaxe deve ter o mesmo significado sem importar seu contexto;
g) reusabilidade: esta propriedade diz respeito aos mecanismos que uma LP fornece para o reaproveitamento de rotinas. Uma linguagem pode ou não incentivar o reuso de código permitindo ao programador definir subrotinas ou fazer uso do paradigma de orientação a objetos;
h) modificabilidade: são as facilidades que uma LP fornece ao programador para escrever código de fácil modificação. O uso de constantes simbólicas, separação entre interface e implementação e tipos abstratos são exemplos de práticas para a melhoria na modificabilidade de uma aplicação;
i) portabilidade: esta propriedade diz respeito a facilidade de uma aplicação desenvolvida para uma arquitetura de *hardware* e sistema operacional ser portada

para outra.

2.1.1 Conceitos de orientação a objetos

Nesta seção são abordados alguns dos conceitos principais para a programação orientada a objetos. Algumas linguagens não suportam todos os recursos deste paradigma. Wilson e Clark (1993, p. 192) utilizam como exemplo a linguagem C++ dizendo que esta pode ser utilizada como uma linguagem imperativa convencional ou como uma linguagem orientada a objetos. Os autores ainda complementam citando linguagens como Smalltalk e Eiffel onde a abordagem de orientação a objetos é obrigatória.

Segundo Hunt (1997, p. 18-19), a visão de programadores que utilizam o paradigma de orientação a objetos é de objetos de dados bem definidos que trocam mensagens entre si. A idéia básica é que um sistema deve ser visto como uma coleção de objetos organizados em classes que se comunicam a fim de realizar uma tarefa em conjunto. As características principais da orientação a objetos são:

a) encapsulamento: é o processo de esconder os detalhes de um objeto que não contribuem para suas características principais;
b) herança: em muitos casos objetos tem propriedades semelhantes, sendo que um modo de classificar estas propriedades é através da herança. Duas classes que herdam de uma classe em comum contém propriedades semelhantes;
c) abstração: é a capacidade de fazer uso de uma funcionalidade sem ter a necessidade de ter completo entendimento de seu funcionamento;
d) polimorfirmo: é a habilidade de mandar a mesma mensagem para diferentes instâncias que aparentemente têm a mesma funcionalidade. No entanto, o tratamento dado para tal mensagem cabe à classe a qual a instância pertence.

2.2 PLATAFORMA .NET

A plataforma .NET, segundo a Microsoft Corporation (2005), tem como objetivo conectar informações, sistemas, pessoas e dispositivos baseando-se em padrões abertos já conhecidos como o *eXtensible Markup Language* (XML). Para este fim, a Microsoft disponibiliza todas as ferramentas e tecnologias necessárias para o desenvolvedor construir aplicações. A seguir são listados alguns conceitos necessários para a melhor compreensão do funcionamento da plataforma .NET.

2.2.1 *Common Language Runtime* (CLR)

Segundo Thai e Lam (2001, p. 17), o CLR é o componente mais importante da plataforma .NET. É ele que gerencia a execução de programas escritos em linguagens .NET e é a base para a arquitetura .NET. O CLR fornece um ambiente de execução gerenciado que é responsável por: compilar o código intermediário para código nativo da máquina onde está executando; executar o código nativo; ativar objetos; fazer verificações de segurança e coletar o lixo (*garbage collecting*), entre outras funcionalidades.

2.2.2 *Common Language Specification* (CLS)

De acordo com Thai e Lam (2001, p. 36 - 37), um dos objetivos da plataforma .NET é suportar a integração de diferentes LPs de tal maneira que programas possam ser escritos em qualquer linguagem e que aplicações .NET possam interagir umas com as outras aproveitando os recursos de herança, polimorfismo, tratamento de exceções e outras funcionalidades. No entanto, uma linguagem pode suportar um recurso totalmente diferente de outra linguagem.

Com o intuito de trazer todas as linguagens ao mesmo nível, a Microsoft publicou a CLS. Esta é uma especificação comum para todas as linguagens de programação para a plataforma .NET.

A interoperabilidade de linguagens da plataforma .NET é possível porque toda aplicação .NET, independente da linguagem na qual foi escrita, é convertida para MSIL. Esta é uma linguagem de baixo nível que o CLR pode ler e executar (STOECKER, 2003, p. 3).

2.2.3 *Common Type System* (CTS)

O CTS é a especificação da Micrsoft que garante que todas a linguagens da plataforma .NET tenham um entendimento comum entre seus tipos incluindo classes e interfaces (THAI; LAM, 2001, p. 36-37). Os tipos primitivos também devem corresponder entre as linguagens de programação da plataforma .NET. Por exemplo, o tipo inteiro da linguagem C# representado pela palavra reservada *int*, quando compilado para MSIL, deve ser o mesmo que o tipo inteiro *Integer* da linguagem VB.NET (STOECKER, 2003, p. 3).

2.2.4 *Microsoft Intermediate Language* (MSIL)

É a representação intermediária de aplicações escritas para a plataforma .NET. Esta é utilizada pelo CLR para executar aplicações para esta plataforma. A *Microsoft Intermediate Language* (MSIL) é compilada dentro de uma arquivo que recebe a denominação de *assembly* .NET. Este é uma unidade de código executável pela plataforma .NET, também chamado de .NET *Portable Executable* (PE).

A MSIL é constituída de dois componentes principais: metadados e código gerenciado. Metadados representa os elementos estruturais de um *assembly* .NET, incluindo classes,

atributos e métodos. Código gerenciado representa a funcionalidade dos métodos da aplicação (LIDIN, 2002, p. 5).

Segundo Thai e Lam (2001, p. 31), a MSIL suporta todas as funcionalidades de linguagens orientadas a objetos, incluindo abstração de dados, herança e polimorfismo, além de tratamentos de exceções e eventos. Todas as linguagens da plataforma .NET devem ser convertidas em MSIL. Assim, a plataforma .NET suporta múltiplas linguagens.

2.2.5 *Global Assembly Cache* (GAC)

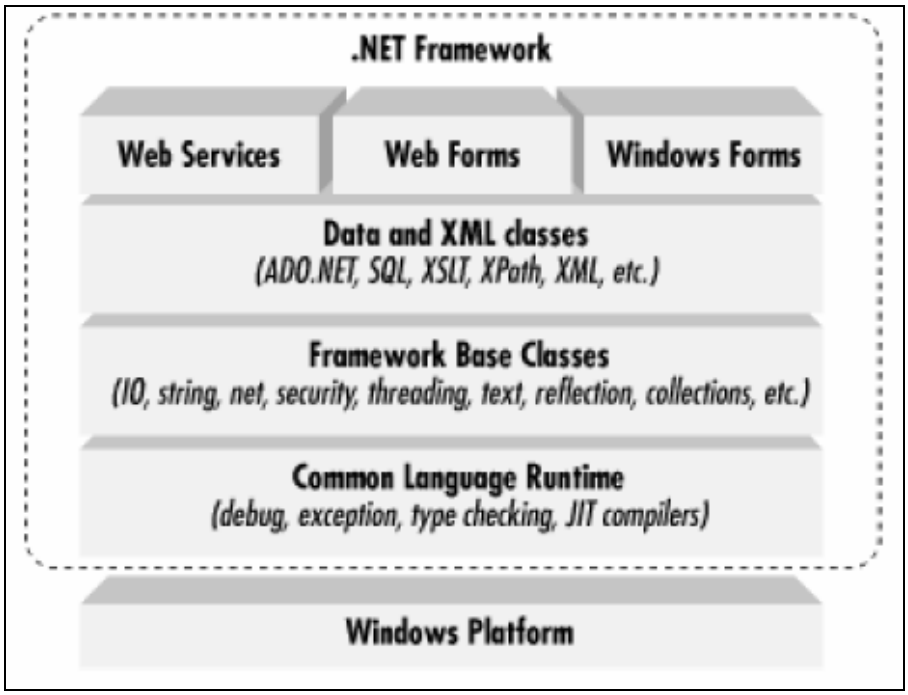
A plataforma .NET permite a criação de *assemblies* executáveis e bibliotecas de classes. *Assemblies* executáveis recebem a extensão .EXE e bibliotecas de classes recebem a extensão .DLL. Ambos podem referenciar outros *assemblies* .DLL. Estes podem estar localizados em um repositório comum de *assemblies* denominado GAC.

O GAC é um diretório de *assemblies* compartilhados pelas aplicações .NET. Serve para organizar os *assemblies* de acordo com suas versões, podendo conter mais de uma versão do mesmo *assembly*. Isto resolve um problema conhecido na arquitetura Windows denominado *DLL Hell*, que é a sobrescrita de versões anteriores de bibliotecas através da instalação de novos programas, trazendo a instabilidade do programas anteriores (THAI; LAM, 2001, p. 14).

2.2.6 *Framework* .NET

O *framework* .NET é o ambiente de desenvolvimento e execução de aplicações .NET. Este *framework* gerencia todos os aspectos da execução da aplicação. É constituído de dois componentes principais, o CLR e a biblioteca de classes .NET (STOECKER, 2003, p. 2).

Segundo Thai e Lam (2001, p. 15-16), o *framework* .NET está no topo do sistema operacional e consiste de vários componentes, sendo o mais importante deles o CLR. A Figura 1 apresenta os principais componentes do *framework* .NET.



Fonte: Thai e Lam (2001, p. 12)

Figura 1 – Componentes do *framework* .NET

2.2.7 Biblioteca de classes do *framework* .NET

É um conjunto de classes e interfaces que fornece ao desenvolvedor uma série de serviços considerados complexos de implementar. Muitas das classes apresentadas por esta biblioteca podem ser estendidas pelo programador, se necessário (STOECKER, 2003, p. 6).

A biblioteca de classes do *framework* .NET é dividida em *namespaces* que são agrupamentos lógicos de classes. Por exemplo, o *namespace System.Windows.Forms* contém todas as classes necessárias para criar formulários para aplicações Windows (STOECKER, 2003, p. 6). O Quadro 1 descreve os principais *namespaces* da biblioteca de classes do *framework* .NET.

NAMESPACE	**DESCRIÇÃO**
System	Todos os tipos básicos
System.Collections	Tipos para auxiliar no gerenciamento de coleções, incluindo pilhas, filas, tabelas *hash*, entre outros
System.Diagnostics	Tipos para auxiliar na depuração de aplicações
System.Drawing	Tipos para manipular gráficos 2D. São usados para aplicações que utilizam formulários do Windows bem como na criação de imagens para páginas web
System.EnterpriseServices	Tipos para gerenciar transações, *pool* de objetos e segurança, entre outras funcionalidades para fazer o uso de código gerenciado mais eficiente no servidor
System.Globalization	Tipos para o *National Language Support* (NLS), como comparação de *strings*
System.IO	Tipos para gerenciar entrada e saída de dados, navegar em diretórios e em arquivos
System.Management	Tipos para gerenciar outros computadores de uma corporação através do *Windows Management Instrumentation* (WMI)
System.Net	Tipos para efetuar comunicação em rede
System.Reflection	Tipos para inspecionar metadados e efetuar a amarração de tipos e seus membros
System.Resources	Tipos para manipular *resources* externos
System.Runtime.InteropServices	Tipos que permitem código gerenciado acessar componentes COM e funções em DLLs Win32

Fonte: adaptado de Richter (2002, p. 30)

Quadro 1 – *Namespaces* da biblioteca de classes do *framework* .NET

2.3 COMPILADORES

Em linhas gerais, um compilador é um programa que aceita como entrada um código de uma linguagem de programação (código fonte) em formato texto e produz como saída outro código em outra linguagem (código objeto) levando em consideração o significado deste texto de entrada (GRUNE et al, 2001, p. 1). Este processo é chamado de tradução quando se trata de linguagens naturais. Para a computação este processo pode ser utilizado para, a partir de um código fonte de uma linguagem qualquer, gerar código para um *hardware* ou máquina virtual.

Outra funcionalidade de um compilador é o tratamento de erros. Em geral o texto ou linguagem de entrada de um compilador é escrito por humanos e freqüentemente contém falhas. Sendo assim, o compilador deve ser projetado de forma a informar ao usuário os erros

cometidos pelo mesmo (LOUDEN, 2004, p. 18).

Segundo Grune et al (2001, p. 23), os compiladores se diferem de acordo com sua arquitetura. Duas questões arquitetônicas devem ser consideradas. A primeira delas é quanto à largura do compilador. Esta define a granularidade dos dados que trafegam entre seus módulos. A segunda diz respeito ao fluxo de controle entre os módulos de um compilador.

Quanto à largura do compilador, ele pode ser estreito ou largo. Um compilador estreito lê uma pequena parte do código fonte, processa as informações obtidas e já produz parte do código objeto. Estes são mais complexos de se desenvolver e são mais apropriados quando se provê de pouca memória principal. Um compilador largo carrega o programa inteiro para a memória e faz uma série de transformações nele para então gerar o código objeto.

O campo de visão[1] de um compilador estreito é limitado, e em muitos casos, este não pode administrar todas as suas transformações durante a execução de apenas uma varredura do programa fonte. Tais compiladores escrevem uma versão parcialmente transformada do programa no disco e executam mais de uma passagem para efetuar as transformações pendentes. Este tipo de compilador é chamado de compilador *n* passagens. “No futuro, esperamos ver mais compiladores largos e menos compiladores estreitos. Quase todos os compiladores para os novos paradigmas de programação já são largos...” (GRUNE et al, 2001, p. 24).

A outra consideração arquitetônica se refere ao fluxo de controle entre os módulos do compilador. Para o compilador largo, o controle não se torna um problema tendo em vista que os módulos funcionam em seqüência. Em sua execução, cada módulo tem controle total da mesma já que recebe os dados de entrada por completo e os disponibiliza também de forma

[1] Neste contexto, campo de visão deve ser entendido como a quantidade de informações que um compilador possui em um determinado momento.

integral, após seu processamento, para o próximo módulo. Em um compilador estreito, é mais complicado já que os módulos recebem os dados de entrada fragmentados. Nesse caso, o fluxo de controle não é seqüencial, devendo ativar o módulo apropriado no momento certo (GRUNE et al, 2001, p. 25).

2.3.1 Módulos de um compilador

Segundo Louden (2004, p. 6-13), o processo de compilação está dividido nas seguintes fases: analisador léxico, analisador sintático, analisador semântico, gerador de código intermediário ou otimizador de código fonte, gerador de código alvo e otimizador de código alvo. Grune et al (2001, p. 20) divide os módulos de um compilador em *front-end* e *back-end*. Os módulos de *front-end* são: o sistema de varredura, o analisador sintático, o analisador semântico e o gerador de código intermediário. Os demais módulos responsáveis por geração e otimização de código alvo são denominados *back-end*.

A seguir são detalhados os módulos de *front-end* de um compilador. Os demais módulos não são discutidos já que não são implementados neste trabalho. A execução do compilador a partir da linguagem intermediária é de responsabilidade da plataforma .NET.

2.3.1.1 Sistema de varredura ou analisador léxico

Geralmente o código fonte do programa é fornecido como uma seqüência de caracteres. Esta é organizada nesta fase como unidades significativas denominadas marcas. As marcas ou *tokens* são como palavras em uma linguagem natural. “O sistema de varredura tem função similar a de um sistema para soletrar.” (LOUDEN, 2004, p. 8).

Para o código fonte de uma linguagem de programação, o sistema de varredura

identifica literais, identificadores, palavras reservadas e outros *tokens* relevantes para a análise sintática. Também faz parte da análise léxica, descartar os *tokens* que não são utilizados para a geração de código como quebra de linha e comentários de linha e bloco.

Expressões regulares fornecem a principal descrição formal para a análise léxica. Uma expressão regular é uma fórmula que descreve um conjunto de palavras (GRUNE et al, 2001, p. 55). Porém apenas isto não é suficiente para tal especificação. A especificação léxica deve permitir a compilação de cadeias de caracteres em classes de símbolos, permitindo assim utilizar estas classes de símbolos ao invés de apenas símbolos (WILHELM; MAURER, 1995, p. 248). Esta forma é chamada de especificação ou definição regular. O Quadro 2 apresenta a definição de identificadores para uma linguagem de programação qualquer utilizando uma especificação regular. Para tal, são utilizadas as classes de caracteres `letras` e `digitos`.

```
letras: [a..zA..Z]
digitos: [0..9]
identificador: (letra | _ (letra | digito)) (letra | digito)*
```

Quadro 2 – Exemplo de especificação regular

2.3.1.2 Analisador sintático

A análise sintática determina a estrutura de um programa com base em uma gramática. A especificação da estrutura sintática é feita utilizando o alfabeto definido pela análise léxica (LOUDEN, 2004, p. 96 - 97).

Na definição de gramáticas pode ser utilizada a *Backus-Naur Form* (BNF). O Quadro 3 apresenta uma BNF para definir uma operação aritmética, onde `identificador` representa o *token* definido no Quadro 2. Cada linha do quadro é uma regra gramatical.

```
expr -> mult + mult
mult -> oper * oper
oper -> identificador
```

Quadro 3 – Exemplo de BNF

Embora a BNF seja uma forma completa de representar uma linguagem, foi desenvolvida uma extensão para esta. A BNF estendida (EBNF) possui alguns recursos que facilitam a especificação de uma linguagem, tais como os operadores `*` e `+`, também utilizados na definição de expressões regulares (GRUNE et al, 2001, p. 33-34). Estes operadores indicam a repetição da expressão que os antecedem. O operador `*` indica qualquer quantidade de repetição, enquanto o operador `+` qualquer quantidade maior do que 0. O Quadro 4 contém um exemplo parcial de uma gramática que define um comando de atribuição de uma LP qualquer utilizando a EBNF. No exemplo são utilizados o *token* `identificador` e o símbolo especial `:=`.

```
var -> var1 := expr
var1 -> identificador (, identificador)*
```

Quadro 4 – Exemplo de EBNF

A função principal da análise sintática é a criação da *Abstract Syntax Tree* (AST) ou árvore sintática abstrata (GRUNE et al, 2001, p. 48). Esta é uma representação da linguagem de programação de forma hierárquica. As folhas desta árvore são os *tokens* definidos pela análise léxica e os nós intermediários são os símbolos da gramática (LOUDEN, 2004, p. 109 - 114).

Há dois modos de realizar a análise sintática: *top-down* através do método *Left-to-right Leftmost derivation* (LL) e *bottom-up* através dos métodos LR – *Left-to-right Rightmost derivation* (LR) e *Look Ahead Left-to-right Rightmost derivation* (LALR). A finalidade de ambos é a construção de uma árvore sintática a partir de uma entrada (GRUNE et al, 2001, p. 101-102). Os dois métodos diferem de acordo com a ordem em que os nós da árvore sintática são construídos. O método *top-down* monta a árvore sintática a partir do nó superior (em pré-ordem), enquanto o método *bottom-up* contrói os nós da árvore sintática em pós-ordem. Isto é, a parte superior de uma subárvore é construída após todos os seus nós inferiores terem sido construídos (GRUNE et al, 2001, p. 104).

A construção de um analisador sintático *top-down* exige a definição de uma gramática LL que é uma gramática que não contém recursão à esquerda nem não determinismo à esquerda. Este conflito não permite que algumas gramáticas complexas sejam definidas neste formalismo. Analisadores sintáticos *bottom-up* permitem a análise de gramáticas que resolvem este tipo de conflito. É o caso das gramáticas LR e LALR. Porém este tipo de analisador tem o inconveniente de ser mais complexo para se desenvolver e compreender (GRUNE et al, 2001, p. 110-158).

2.3.1.3 Analisador semântico

Também chamada de tratamento de contexto, a análise semântica é necessária para duas finalidades distintas: verificar condições de contexto impostas pela especificação da linguagem e coletar informações para processamento semântico. Um compilador ideal, totalmente limpo, executaria estas duas fases seqüencialmente, porém tal tratamento é desnecessário. Isto ocorre porque estas duas fases devem tratar o contexto da mesma forma. O resultado do analisador semântico é a árvore sintática anotada cuja estrutura é semelhante à AST porém seus nós possuem informações relevantes para as demais fases da compilação (GRUNE et al, 2001, p. 178-179).

2.3.1.4 Gerador de código intermediário

Nesta fase da compilação, a árvore sintática anotada é transformada na árvore de código intermediário. Esta contém informações como expressões (inclusive atribuições), chamadas de rotinas, cabeçalhos de procedimentos e instruções de retorno, e saltos condicionais e incondicionais (GRUNE et al, 2001, p. 254).

A árvore resultante deste módulo do compilador normalmente aumenta o tamanho da AST, porém diminui sua complexidade. Isto se deve ao fato de que as instruções e conceitos de alto nível definidos pela linguagem são substituídos por conceitos de baixo nível (GRUNE et al, 2001, p. 254-255).

2.3.2 Compiladores orientados a objetos

Grande parte da construção de compiladores é independente do paradigma do código fonte. Ainda assim um compilador Java difere de um compilador Prolog. Quanto às técnicas para as análises léxica e sintática, dificilmente são específicas para cada paradigma de programação (GRUNE et al, 2001, p. 398).

Para um analisador semântico de um compilador de uma linguagem orientada a objetos deve ser considerada a definição de classes. Louden (2004, p. 323) diz que a declaração de uma classe vai além da definição de um simples tipo de dado já que esta pode incluir métodos e atributos, além de permitir o uso de características como herança, que as linguagens procedurais não suportam. Segundo Wilhelm e Maurer (1995, p. 181), métodos são compilados da mesma forma que funções em linguagens imperativas. A diferença é que métodos podem acessar membros de seus próprios objetos diretamente.

A identificação de nomes também é efetuada através do escopo no qual o identificador pertence. Estes funcionam em forma de pilha onde os identificadores declarados são inseridos no elemento do escopo superior. Espaços de nomes, classes e métodos fazem parte do escopo de uma linguagem de programação orientada a objetos (GRUNE et al. 2001, p. 402-406). Algumas linguagens de programação permitem a importação de escopos de outros lugares. Estes passam a fazer parte do escopo da classe que está efetuando a importação.

2.3.3 Ferramentas geradoras de compiladores

Para facilitar o desenvolvimento de compiladores, existem diversas ferramentas que geram código para analisadores léxicos, sintáticos e até semânticos. Estas são denominadas *compiler-compilers* (compiladores de compiladores).

Os geradores de analisadores léxicos têm como entrada definições regulares que especificam os *tokens* da linguagem de programação. Os geradores de analisadores sintáticos geralmente utilizam BNF ou EBNF para especificar a estrutura sintática da linguagem. As regras semânticas são definidas geralmente por gramáticas de atributos. Existem ferramentas que através deste formalismo geram analisadores semânticos, embora as ferramentas *compiler-compiler* mais comuns geram os analisadores léxicos e sintáticos (LOUDEN, 2004, p. 229-240).

2.3.3.1 JavaCCCS

JavaCCCS é uma adaptação do JavaCC para gerar código C# ao invés de Java. Esta ferramenta é um compilador de compilador que a partir de um arquivo de uma gramática LL, gera código para análises léxica e sintática (SOURCEFORGE, 2005).

JavaCC é o gerador de compiladores mais popular para uso em aplicações Java (JAVA.NET, 2004). Este gera analisadores sintáticos *top-down* permitindo o uso de gramáticas mais comuns. Como vantagem, o JavaCC também possui uma grande comunidade de usuários, o que torna uma tarefa fácil encontrar exemplos de gramáticas para esta ferramenta.

Outra vantagem de utilizar o JavaCC, da mesma forma que o JavaCCCS, é permitir especificação EBNF, o que torna mais fácil de manter a especificação da linguagem.

2.4 TRABALHOS CORRELATOS

Em Kudzu (2005) é apresentada uma solução para agregar funcionalidades de acesso a dados para a plataforma .NET. Sua pesquisa consiste em fazer uso de *custom attributes*[2] para definir como os dados serão buscados em um banco de dados (BD) utilizando a linguagem SQL (*Structured Query Language*).

Para solucionar o mesmo problema, a Microsoft está desenvolvendo uma linguagem de programação chamada C Omega (Cw). Segundo Microsoft Research (2005), Cw é uma linguagem de programação que estende o C#. É fortemente tipada com o intuito de preencher o espaço entre o CTS e os dados relacionais (SQL) ou semi-estruturados (XML). Além disso, trata a programação concorrente com maior facilidade se comparada com o C#.

[2] Funcionalidade do .NET que permite que o desenvolvedor adicione informações nos metadados de uma classe ou membro de classe.

3 DESENVOLVIMENTO DO TRABALHO

Este capítulo descreve as especificações léxica, sintática e semântica da linguagem de programação proposta bem como o desenvolvimento de um compilador para ela.

3.1 ESPECIFICAÇÃO DA LINGUAGEM

A linguagem em questão tem como características ser orientada a objetos e projetada para a plataforma .NET, permitindo o uso de sua biblioteca de classes. Também são características desta linguagem:

a) permitir sobrecarga de métodos;

b) possuir herança simples;

c) ser *case-sensitive*;

d) ter sintaxe semelhante à linguagem C para os comandos de controle de fluxo e semelhante às linguagens Java e C# para as classes;

e) permitir o uso de rotinas escritas em outras linguagens .NET.

f) permitir que outras linguagens .NET utilizem as rotinas desenvolvidas na linguagem proposta;

g) ser produtiva para o desenvolvimento da camada de negócio[3] de uma aplicação comercial.

[3] Endente-se por camada de negócio toda a parte de um software que contém a lógica da aplicação. É onde os dados são manipulados para gerar um resultado esperado. Pode abranger desde validações simples de dados até cálculos complexos.

3.1.1 Especificação léxica da linguagem

A especificação léxica da linguagem é feita usando definições regulares. O Quadro 5 apresenta a definição[4] de letras e dígitos, utilizados para definir os *tokens* da linguagem.

```
#letter = [a-zA-Z]
#digit = [0-9]
```

Quadro 5 – Definições regulares

Alguns caracteres ou seqüências de caracteres devem ser ignorados. Este são os caracteres que representam nova linha ou *line feed*, retorno de cursor ou *carriage return*, tabulação horizontal e comentários de linha e bloco, definidos a seguir. Estas definições não são identificadas já que não são utilizadas para especificar outros *tokens*. Os caracteres *line feed*, *carriage return* e tabulação horizontal são representados por `\n`, `\r` e `\t`, respectivamente (Quadro 6).

```
(\n | \r | \t)
```

Quadro 6 – Caracteres ignorados

Os comentários de linha iniciam com duas barras e terminam no final da linha corrente. Comentários de bloco são delimitados por `/*` (no início) e `*/` (no final). A especificação[5] dos comentários é apresentada no Quadro 7.

```
// (~[\n,\r])* (\n | \r | \r\n)
"/*" (~["*/"])* "*/"
```

Quadro 7 – Comentários de linha e bloco

A linguagem proposta tem as seguintes literais ou constantes:

a) inteiro e decimal: representam constantes para os tipos numéricos. No Quadro 8, a primeira definição (`integer`) representa constantes numéricas sem casas decimais, enquanto a segunda (`numeric`) define literais numéricas com casas decimais.

[4] O caractere # no início da definição indica que esta é uma definição auxiliar utilizada apenas na especificação dos *tokens* da linguagem.

[5] A expressão regular `~[X]` indica que pode ser encontrado qualquer caractere com exceção da seqüência representada por `X`.

Também é permitido que constantes numéricas possuam expoentes para facilitar a definição de seus valores. O número após o caractere `e` indica um expoente na base 10 que é multiplicado pelo número antes deste caractere (por exemplo, `2e3` é igual a 2 x 10^3 ou 2000);

b) caractere e cadeia de caracteres: definidas[6] no Quadro 9, estas literais representam um caractere isolado (delimitado por apóstrofo) e uma seqüência deles (delimitada por aspas), respectivamente. Para representar os caracteres *line feed*, *carriage return*, tabulação horizontal, barra invertida, apóstrofo e aspas são utilizadas as seqüências de *escape* `\n`, `\r`, `\t`, `\\`, `\'` e `\"`;

c) data e hora e intervalo de tempo: uma literal `datetime` representa um momento no tempo (por exemplo, a constante `datetime '31/12/2005 23:59:59'` representa o dia 31 de dezembro de 2005 às 23:59:59), enquanto uma literal `timespan` representa uma quantidade de tempo com precisão de segundos (por exemplo, a constante `timespan '6:00:00'` representa a quantidade de 6 horas). Estas constantes estão definidas[7] no Quadro 10;

d) lógico: constantes lógicas são definidas pelas palavras reservadas (Quadro 12) `true` e `false`.

```
INTEGER = digit+
NUMERIC = digit+ . digit+ expoent?
        | . digit+ expoent?
        | digit+ expoent
#expoent = e ("+" | "-") digit+
```

Quadro 8 – Constantes numéricas

```
CHARACTER = ' ( ~['\,\n,\r] | \[n,t,r,\,'","] ) '
STRING    = " ( ~["\,\n,\r] | \[n,t,r,\,'","] )* "
```

Quadro 9 – Constantes dos tipos caractere e cadeia de caracteres

[6] O operador `?` indica que a expressão anterior é opcional.

[7] A seqüência `<SPACE>` significa que pode vir um caractere em branco.

```
DATETIME = "datetime" (<SPACE>)? ' date <SPACE> time '
#date = digit digit? / digit digit? / digit digit (digit digit)?
#time = digit digit : digit digit (: digit digit)?
TIMESPAN = "timespan" (<SPACE>)?
           ' digit digit : digit digit : digit digit '
```

Quadro 10 – Constantes dos tipos data e hora e intervalo de tempo

Os identificadores da linguagem podem conter apenas letras, dígitos ou o caractere sublinhado *underscore* (_), sendo que não podem iniciar com dígito. Caso o identificador inicie com o caractere *underscore*, este deve ser seguido por letra ou dígito (Quadro 11).

```
ID = (letter | _ (letter | digit))(letter | digit | _)*
```

Quadro 11 – Identificadores

Alguns *tokens*, chamados palavras reservadas, agregam um significado a mais para a linguagem e não podem ser definidos como identificadores. O Quadro 12 lista estes *tokens*.

abstract	do	long	static
asc	double	namespace	string
assert	else	new	switch
base	extends	null	sync
bool	false	order	this
break	finally	override	throw
by	float	private	time
byte	for	protected	timespan
case	foreach	public	true
catch	from	readonly	try
char	get	return	unsigned
class	if	sealed	using
continue	implements	select	virtual
date	in	set	void
datetime	int	short	where
default	internal	signed	while
desc	like	soft	

Quadro 12 – Palavras reservadas

Os operadores para expressões binárias e unárias que podem ser utilizados na linguagem estão relacionados no Quadro 13. Estes representam operações relacionais, aritméticas, lógicas e de atribuição.

OP.	DESCRIÇÃO	OP.	DESCRIÇÃO
==	Igual	&	E
!=	Diferente	|	Ou
<	Menor	^	Ou exclusivo
>	Maior	=	Atribuição
<=	Menor ou igual	+=	Soma e atribuição
>=	Maior ou igual	-=	Subtração e atribuição
+	Adição ou positivo	*=	Multiplicação e atribuição
-	Subtração ou negativo	/=	Divisão e atribuição
*	Multiplicação	%=	Resto e atribuição
/	Divisão	&=	E e atribuição
%	Resto	|=	Ou e atribuição
!	Negação	^=	Ou exclusivo e atribuição

Quadro 13 – Operadores

3.1.2 Especificação sintática da linguagem

A principal estrutura de uma linguagem orientada a objetos é a classe. Esta é definida utilizando a palavra reservada `class` seguida por um identificador. A especificação de classes e seus membros encontram-se no Quadro 14. Comandos são especificados no Quadro 15 e expressões no Quadro 16. A sintaxe da linguagem foi especificada utilizando a EBNF, conforme definição da ferramenta JavaCCCS.

```
unitDecl ::= "namespace" name ";" usingList ( classDecl )+ <EOF>
usingList ::= ( "using" name ( ".*" ";" | ";" ) )*
classDecl ::= classModifiers "class" <ID> "extends" name
              "{" membersDecl "}"
classModifiers ::= ("public" | "internal" | "abstract" | "sealed")*
membersDecl ::= ( memberDecl )*
memberDecl ::= memberModifiers returnType <ID> ( method | attribute )
method ::= "(" ( parameters )? ")" ( blockCmd | ";" )
attribute ::= ";"
memberModifiers ::= (   "public"
                      | "protected"
                      | "internal"
                      | "private"
                      | "abstract"
                      | "virtual"
                      | "override"
                      | "static"
                    )*
parameters ::= type <ID> ( "," type <ID> )*
```

Quadro 14 – Definição de classes

```
command ::=   ifCmd
            | switchCmd
            | whileCmd
            | doCmd
            | continueCmd
            | breakCmd
            | returnCmd
            | forCmd
            | actionOrVarCmd
            | blockCmd
ifCmd ::= "if" "(" expression ")" blockCmd ( "else" command )?
switchCmd ::= "switch" "(" expression ")"
              "{" ( "case" expression ":" blockCmd )*
                  "default" ":" blockCmd
              "}"
whileCmd ::= "while" "(" expression ")" blockCmd
doCmd ::= "do" blockCmd "while" expression ";"
continueCmd ::= "continue" ";"
breakCmd ::= "break" ";"
returnCmd ::= "return" ( expression )? ";"
forCmd ::= "for" "(" expression (<ID> "=" expression ";" | ";")
                     expression ";" expression
                  ")" blockCmd
actionOrVarCmd ::= expression (<ID> ("=" expression)? ";" | ";")
blockCmd ::= "{" ( command )* "}"
```

Quadro 15 – Definição de comandos

```
exprList ::= expression ( "," expression )*
expression ::= or ( assignOp expression )?
orExpr ::= xorExpr ( orOp orExpr )?
xorExpr ::= andExpr ( xorOp xorExpr )?
andExpr ::= equalExpr ( andOp andExpr )?
equalExpr ::= relationalExpr ( equalOp equalExpr )?
relationalExpr ::= addExpr ( relationalOp relationalExpr )?
addExpr ::= multExpr ( addOp addExpr )?
multExpr ::= unaryExpr ( multOp multExpr )?
unaryExpr ::= unaryOp unaryExpr | prefixExpr ( suffixExpr )*
prefixExpr ::= ( <ID> | "this" | "base" ) ( "(" ( exprList )? ")" )?
             | newExpr
             | literalExpr
             | primitive
             | ( "(" expression ")" ( expression )? )
suffixExpr ::= "." <ID> ( "(" ( exprList )? ")" | "[" exprList "]" )?
newExpr ::= "new" type "(" ( exprList )? ")"
literalExpr ::=   <STRING>
                | <CHARACTER>
                | <NUMERIC>
                | <INTEGER>
                | <DATETIME>
                | <TIMESTAMP>
                | "true"
                | "false"
                | "null"
name ::= <ID> ( "." <ID> )*
nameList ::= name ( "," name )*
type ::=  primitive | name
returnType ::= type | "void"
primitive ::=   "string"
              | "char"
              | "bool"
              | "datetime"
              | "timespan"
              | "float"
              | "double"
              | "short"
              | "int"
              | "long"
              | "byte"
assignOp ::= "=" | "+=" | "-=" | "/=" | "%=" |
                   "*=" | "&=" | "|=" | "^="
unaryOp ::= "+" | "-" | "!"
andOp ::= "&"
orOp ::= "|"
xorOp ::= "^"
equalOp ::=  "==" | "!="
relationalOp ::= "<" | ">" | "<=" | ">=" | "like"
addOp ::=  "+" | "-"
multOp ::=  "*" | "/" | "%"
```

Quadro 16 – Definição de expressões

3.1.3 Especificação semântica da linguagem

Faz parte da análise semântica a verificação da definição das classes e seus membros bem como do código escrito pelo desenvolvedor no contexto do método e da classe a qual pertence. A seguir são detalhadas as verificações semânticas da linguagem proposta. As mensagens de erro geradas por esta análise são apresentadas no Apêndice A.

3.1.3.1 Resolução de nomes

A resolução de nomes da linguagem proposta é feita utilizando tabelas de símbolos. Cada pacote, classe ou instância de classe[8] possui uma tabela de símbolos distinta. Isto se deve ao fato de cada uma delas possuir um contexto diferente.

Para resolver um nome é verificada na tabela de símbolos corrente a existência de alguma entrada com o identificador do nome que está sendo analisado. Por exemplo, o reconhecimento do nome `Cadastro` deve ser efetuado na tabela de símbolos do contexto onde se encontra. Caso este nome não se encontre no contexto de uma instância de classe ou da classe, o contexto delega a responsabilidade de encontrá-lo para a tabela de símbolos que o antecede. Esta é a tabela de símbolos do pacote que por sua vez pode identificar o nome `Cadastro` como sendo uma classe. O escopo de um nome é a tabela de símbolos de uma classe ou método que pertence. Caso o escopo seja a tabela de símbolos de um método, a tabela de símbolos que a antecede é a tabela de símbolos de uma classe.

Também é válido que um nome contenha um escopo, como por exemplo

[8] A tabela de símbolos de instância de classe contém os membros que não são estáticos, enquanto a tabela de símbolos da classe contém apenas os membros estáticos. Estas tabelas devem ser separadas, pois, dependendo do contexto em que o nome é resolvido, deve-se utilizar uma tabela ou outra. Por exemplo, caso seja acessada através de uma referência para uma instância da classe, usa-se a tabela de instância de classe.

`org.furb.Empregado`. A tabela de símbolos do contexto atual do qual o nome está sendo resolvido não tem conhecimento da classe `Empregado`. Porém, tem conhecimento do *namespace* `org` (através da tabela de símbolos global para toda a compilação) que por sua vez conhece o *namespace* `furb` (através da tabela de símbolos do *namespace* `org` que agora é o contexto para a resolução de nomes) que finalmente possui em sua tabela de símbolos a classe `Empregado`. Caso o identificador não seja encontrado na tabela de símbolos corrente, é gerado o erro semântico `SymbolNotDeclared` e no caso de haver mais de uma entrada para o mesmo identificador, `AmbiguityError`.

3.1.3.2 Tipos de dados

Existem dois grupos de tipos de dados. O primeiro são os tipos primitivos pré-definidos pela linguagem (Quadro 17). O segundo grupo são referências para instâncias de classes, sendo estas classes definidas pelo programador na linguagem proposta ou em outra linguagem da plataforma .NET. Estes tipos são verificados através da resolução de nomes.

NOME	**DESCRIÇÃO**
string	Cadeia de caracteres da tabela ASCII
char	Um único caractere da tabela ASCII
bool	Tipo lógico, podendo assumir os valores *true* para verdadeito e *false* para falso
datetime	Tipo que representa um momento no tempo
timespan	Tipo que representa um intervalo de tempo
float	Tipo numérico de ponto flutuante com precisão de 7 dígitos
double	Tipo numérico de ponto flutuante com precisão de 15 dígitos
Byte	Tipo inteiro de 8 bits com sinal
Short	Tipo inteiro de 16 bits com sinal
Int	Tipo inteiro de 32 bits com sinal
Long	Tipo inteiro de 64 bits com sinal

Quadro 17 – Tipos primitivos

3.1.3.3 Classes

Dentro de um arquivo fonte podem ser declaradas uma ou mais classes que

compartilham dos mesmos `namespace` e diretivas `using`.

O `namespace` diz respeito à localização lógica da classe e pode ser qualquer nome definido pelo não terminal `name`. A diretiva `using` é equivalente ao `import` do Java. É um atalho para o desenvolvedor não precisar escrever sempre o nome completo de uma classe. Caso um `using` termine com a seqüência `.*` significa que será importado um `namespace` sendo que o nome que antecede a estes caracteres representa o `namespace`, da mesma forma que a diretiva `using` da linguagem C#. Caso contrário, deve ser importada uma classe. Esta verificação é feita utilizando o resolvedor de nomes. Se não estiver de acordo, os erros `InvalidUsingNs ou InvalidUsingClass` são gerados. No exemplo do Quadro 18, a linha 3 indica que todas as classes do pacore `org.furb.pacote2` são reconhecidas dentro da definição da classe. A classe `org.furb.pacote3.Classe1` também pode ser utilizada dentro da classe `ClasseQualquer` através do nome `Classe1`. Isto se deve ao fato desta classe ter sido importada através da diretiva `using` (linha 4). Porém as demais classes do pacote `org.furb.pacote3` precisam ser acessadas através do seu nome completo (`org.furb.pacote3.xxx`).

```
001    namespace org.furb.pacote1;
002
003    using org.furb.pacote2.*;
004    using org.furb.pacote3.Classe1;
005
006    public class ClasseQualquer
007                        extends ClasseBase {
008        //Definição interna da classe
009    }
```

Quadro 18 – Exemplo de definição de classe

A classe deve ter um nome único para o *namespace* onde ela se encontra, caso contrário é gerado o erro `DuplicatedClassDefinition`. Também faz parte da análise semântica da linguagem proposta a verificação da existência da classe base contida no nome que sucede a palavra reservada `extends` (linha 7 no exemplo do Quadro 18). Classes também definem métodos e atributos que devem ser únicos para a classe. Caso sejam duplicados, geram os erros `DuplicatedMethodDefinition` e `DuplicatedAttributeDefinition`.

Adicionalmente ao nome e ao *namespace*, a classe também define modificadores. Estes definem as classes como sendo abstratas, finais ou concretas. Classes abstratas e finais são marcadas com os modificadores `abstract` e `sealed`, respectivamente. Caso a classe não defina o modificador `abstract`, por *default*, é concreta. Classes abstratas não podem ser instanciadas diretamente. É preciso criar uma classe concreta derivada de uma classe abstrata. Classes finais não podem possuir classes derivadas.

Também são definidos modificadores de visibilidade para as classes. Estes podem ser `public` ou `internal`. O primeiro modificador define a visibilidade da classe como pública permitindo que esta possa ser acessada de todos os *assemblies*. O segundo modificador restringe o acesso à classe apenas por outras classes que compartilhem do mesmo *assembly*. Caso nenhum modificador de visibilidade seja definido, a classe é pública. Se o desenvolvedor definir o mesmo modificador mais de uma vez para a classe, irá ocorrer o erro `DuplicatedModifier`.

Os membros da classe também podem conter modificadores de visibilidade. Além dos modificadores `public` ou `internal`, membros de classe podem ser marcados com os modificadores de visibilidade `private` ou `protected`. Os membros privados (`private`) podem ser acessados apenas por outros membros que pertençam a mesma classe, enquanto membros protegidos (`protected`) podem ser acessados internamente à classe (da mesma forma que o membro privado) e através dos membros de suas classes derivadas. Membros de classe também podem ser estáticos (marcados com o modificador `static`), o que permite que sejam acessados diretamente. Ou seja, não é necessário que o membro marcado com este modificador seja acessado através de uma instância da classe. Os modificadores de um membro devem ser únicos, caso contrário, é gerado o erro `DuplicatedModifier`.

Cada classe possui um contexto para que seus membros possam ser encontrados pela resolução de nomes (tabela de símbolos). Outra funcionalidade deste contexto é verificar se o

método ou atributo pode ser utilizado a partir de um outro contexto. Este verificação consiste em analisar as restrições de visibilidade.

3.1.3.4 Métodos

Os métodos adicionam comportamento a uma classe. Cada método possui um bloco de comandos onde o desenvolvedor escreve a lógica da aplicação. A definição de um método aceita parâmetros que são utilizados para controlar seu fluxo de dados. Um parâmetro contém um identificador para acessar seu valor dentro do método. Também é possível que um método tenha um tipo de retorno. Métodos, da mesma forma que classes, podem ser concretos e abstratos. Os métodos concretos devem definir um bloco de comandos. Métodos abstratos devem ser definidos apenas em classes abstratas, pois não possuem implementação (bloco de comandos). Para definir este tipo de método é utilizado o modificador `abstract`.

Para fazer o uso de polimorfismo, a linguagem proposta permite que sejam redefinidos métodos abstratos ou virtuais nas classes derivadas. Métodos virtuais são definidos utilizando o modificador `virtual` que indica que o método pode ser sobrescrito. Todo método abstrato também é virtual, porém um método abstrato não possui implementação obrigando o desenvolvedor a sobrescrevê-lo na classe derivada, se esta for uma classe concreta (não abstrata). Para redefinir o método é preciso apenas redeclará-lo substituindo o modificador `abstract` ou `virtual` pelo modificador `override`. No exemplo do Quadro 19 o método abstrato `void correr(int)` da classe `Animal` é sobrescrito na classe `Cachorro`.

```
001    namespace org.furb;
002
003    public abstract class Animal {
004        public abstract void correr(int km);
005    }
006
007    public class Cachorro extends Animal {
008        public override void correr(int km) {
009            //Lógica para fazer um cachorro correr
010        }
011    }
```

Quadro 19 – Exemplo de sobrescrita de métodos

Também é permitida na linguagem proposta a sobrecarga de parâmetros de métodos, uma vez que a assinatura de um método leva em consideração sua classe, identificador (nome) e seus parâmetros. Isto torna possível a definição de mais de um método com o mesmo nome, porém com parâmetros diferentes. No caso do desenvolvedor definir numa mesma classe mais de um método com mesmo identificador e parâmetros, ocorre o erro `DuplicatedMethodDefinition`.

Para que os parâmetros de um método sejam visíveis dentro dele, este contém um contexto. Da mesma forma que o contexto de uma classe é responsável por “encontrar” seus membros, o contexto de um método deve “encontrar” seus parâmetros. Outra funcionalidade do contexto de um método é verificar se seu código pode acessar membros de instância de sua classe. Isto é verificado de acordo com o modificador `static`. Se o método define este modificador ele é estático, caso contrário ele é de instância podendo ser acessado apenas através de uma referência para uma instância da classe que contém determinado método.

3.1.3.5 Atributos

Atributos podem ser entendidos como variáveis de uma classe. Estes devem possuir um tipo de dado válido e um nome único na classe, que é utilizado por outros membros para acessá-lo. Caso o desenvolvedor defina mais de um atributo com o mesmo nome, o erro `DuplicatedAttributeDefinition` será gerado. Da mesma forma que métodos, os atributos

podem ser estáticos ou de instância. O Quadro 20 contém exemplos de definição de atributos.

```
001    namespace org.furb;
002
003    public class Carro {
004        private double potencia;
005        private Pessoa dono;
006        private static Carro instancia;
007    }
```

Quadro 20 – Exemplo de definição de atributos

3.1.3.6 Comandos

Os comandos da linguagem proposta podem ser comandos de seleção, iteração, desvio incondicional, invocação de métodos ou declaração de variáveis. Embora a declaração de variáveis não satisfaça todas as características de um comando, esta é caracterizada como tal por pertencer diretamente a um bloco de comandos.

`If` e `switch` são comandos de seleção. O comando `if` possui uma condição para executar um bloco de comandos. Este comando também pode opcionalmente possuir um bloco que será executado quando a condição não for satisfeita. A condição deve obrigatoriamente ser do tipo lógico. Caso isto não seja verdade, é gerado o erro semântico `CannotConvertType`. O comando `switch` é utilizado para selecionar um valor que não seja lógico. Este define uma expressão *pivot* a qual é comparada com cada uma das expressões nas seções *case.* O tipo da expressão *pivot* deve aceitar o operador `==` com o tipo de cada uma das expressões *case.* Se esta condição não for satisfeita, o erro `InvalidBinaryOperation` é gerado. Cada expressão *case* define um bloco de comandos a ser executado se o valor da expressão *case* for igual ao valor da expressão *pivot*. É obrigatório que o comando `switch` defina um bloco `default` que é executado no caso do valor da expressão *pivot* não ser igual a nenhum dos valores das expressões *case*. O exemplo apresentado no Quadro 21 demonstra o uso dos comandos `if` e `switch`.

```
001    {
002        if (x) {
003            //Bloco executado caso x == true
004        } else {
005            //Bloco executado caso x == false
006        }
007
008        switch (y) {
009            case 1: {
010                //Bloco executado se y == 1
011            }
012            case 2: {
013                //Bloco executado se y == 2
014            }
015            default: {
016                //Bloco executado se y != 1 & y != 2
017            }
018        }
019    }
```

Quadro 21 – Exemplo de comandos `if` e `switch`

Existem três comandos de iteração. O primeiro é o `while` que repete um bloco de comandos enquanto uma condição definida por uma expressão do tipo lógico for verdadeira. O segundo comando é o `do-while` que executa um bloco de comandos e depois verifica uma expressão também do tipo lógico. Enquanto a expressão for verdadeira, o bloco é executado novamente. Por fim, o comando `for` utiliza três expressões: uma inicialização, uma expressão de comparação e um incremento. A inicialização é executada na primeira iteração do *loop*. Nas demais iterações a expressão de incremento é executada antes da verificação da condição de parada do *loop*. O *loop* acaba quando a expressão de comparação não for mais válida. O exemplo do Quadro 22 demonstra o uso destes comandos.

```
001    {
002        while (true) {
003            //Bloco do comando while
004        }
005        do {
006            //Bloco do comando do-while
007        } while (true);
008        for (int x = 1; x < 12; x += 1) {
009            //Bloco do comando for
010        }
011    }
```

Quadro 22 – Exemplo de comandos de iteração

Os comandos `continue`, `break` e `return` são classificados como comandos de desvio incondicional. O comando `continue` efetua um desvio para a execução da próxima iteração de um *loop* (se houver). O comando `break` é utilizado para sair de um *loop* ou encerrar as

comparações de um comando `switch`. Para encerrar a execução de um método é utilizado o comando `return`. Caso o método possua um valor de retorno, este comando é utilizado para retornar o valor. Caso o tipo da expressão de retorno não seja compatível com o retorno do método, o erro `CannotConvertType` é gerado. No exemplo do Quadro 23 é definido um método cuja função é verificar se um número é ou não primo, utilizando instruções de desvio incondicional.

```
001    private bool ehPrimo(int x) {
002        bool ret = true;
003        for (int i = 2; i < x; i += 1) {
004            if (x % i == 0) {
005                ret = false;
006                break;
007            }
008        }
009        return ret;
010    }
```

Quadro 23 – Exemplo de comandos de desvio incondicional

É permitido ao desenvolvedor declarar variáveis locais para uso em rotinas internas a um método (exemplo Quadro 23, linha 2). Para isto, o programador define seu tipo e um identificador único (nome) para o escopo. Caso seja definida mais de uma variável com mesmo nome para o mesmo escopo, é gerado o erro `DuplicatedVariableDeclaration`. O escopo de uma variável local é definido pelo bloco de comandos atual e seus blocos antecessores. Um bloco é uma lista de comandos definida entre os caracteres `{` e `}`. O Quadro 24 mostra a situação onde ocorre o erro `DuplicatedVariableDeclaration`.

```
001    {
002        if (true) {
003            int x = 0;
004        } else {
005            int x = 0; //Isto esta ok
006        }
007        int y = 0;
008        if (true) {
009            int y = 0; //Ocorre o erro DuplicatedVariableDeclaration
010        }
011    }
```

Quadro 24 – Exemplo de variável duplicada

A invocação de um método também pode ser considerada um comando, se o desenvolvedor não fizer uso de seu valor de retorno. O Quadro 25 exemplifica o uso deste

comando invocando o método `ehPrimo(int)` definido no Quadro 23.

```
001    {
002        ehPrimo(17);
003        //O método é executado, mas seu retorno é descartado
004    }
```

Quadro 25 – Exemplo de invocação de método

3.1.3.7 Expressões

Pode ser considerada uma expressão toda parte de código que define um valor. Portanto, toda expressão tem um tipo. As expressões existentes na linguagem proposta são expressões unárias, binárias, de atribuição, invocação de método, uso de atributo, instanciação de classe e constantes.

A expressão unária contém apenas um operando que é outra expressão e um operador que pode ser o caractere + (positivo), - (negativo) ou ! (negação). Conforme o operador utilizado e o tipo do operando, esta expressão possui um tipo de retorno. Este é definido pelo Quadro 26.

OPERADOR	**OPERANDO**	**TIPO RESULTANTE**
+	timespan	timespan
+	float	float
+	double	double
+	byte	byte
+	short	short
+	int	int
+	long	long
-	timespan	timespan
-	float	float
-	double	double
-	byte	byte
-	short	short
-	int	int
-	long	long
!	bool	bool

Quadro 26 – Expressões unárias

Caso o tipo do operando não suporte o operador da expressão unária, por exemplo a

expressão `!3` (operador de negação para o tipo inteiro), o erro `InvalidUnaryOperation` é gerado.

Expressões binárias são semelhantes às expressões unárias, porém trabalham com dois operandos. Se os operandos não forem válidos para determinado operador, é gerado o erro `InvalidBinaryOperation`. O Apêndice B apresenta as operações válidas para cada operador e operandos.

Para facilitar o uso de cadeias de caractere do tipo `string`, a linguagem proposta possui o operador `like`. A expressão binária retorna o valor lógico verdadeiro caso o operando da esquerda, do tipo `string`, se enquadre com o formado definido pelo operando da direita. O formato deve ser um valor também do tipo `string`, onde o caractere `?` indica que deve ser reconhecido um caractere qualquer e `%` indica que podem ser reconhecidos *n* caracteres. Os demais caracteres contidos no formato são obrigatórios. O Quadro 27 mostra alguns casos de uso para o operador `like`.

```
"Pink Floyd" like "Pink%"        //retorna verdadeiro
"Pink Floyd" like "Pink?"        //retorna falso
"89010-500"  like "?????-???"    //retorna verdadeiro
"89010500"   like "?????-???"    //retorna falso
```

Quadro 27 – Exemplos de uso do operador `like`

A atribuição de um valor para uma variável local ou atributo de classe é feita através de uma expressão. Esta expressão sempre retorna um valor do mesmo tipo da variável ou atributo que é o operando da esquerda. O principal operador de atribuição é o símbolo `=`, mas também é permitido que seja atribuído um valor calculado com base no valor anterior do operando da esquerda (atributo ou variável local). Para isto são utilizados os operadores `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=` e `^=`. O comportamento das operações feitas antes da atribuição é idêntico ao comportamento das expressões binárias `+`, `-`, `*`, `/`, `%`, `&`, `|` e `^`, respectivamente, com a ressalva de que a operação somente pode ser feita se o tipo do operador da esquerda foi igual ao tipo de retorno da expressão binária correspondente.

Quando um método retorna um valor, ele pode ser invocado através de uma expressão cujo valor e tipo de retorno são definidos pelo método. Os parâmetros utilizados para chamar os métodos também são expressões. Estas são resolvidas antes da invocação do método. Se o tipo do parâmetro não corresponder ao definido no método, este não é encontrado pela resolução de nomes, gerando o erro `MethodNotFound`.

Da mesma forma que os métodos, os atributos também podem são acessados por expressões. O tipo resultante da expressão é o próprio tipo do atributo. O mesmo ocorre para variáveis locais.

Para métodos e atributos podem ocorrer erros de visibilidade, quando o escopo de onde está sendo chamado o membro não tem acesso a ele. Os erros ocasionados por estas situações podem ser `PrivateMemberNotVisible`, `ProtectedMemberNotVisible`, `InternalMemberNotVisible` e `ProtectedInternalMemberNotVisible`. Também é verificado se o método ou atributo está sendo chamado a partir de uma instância de sua classe ou de forma estática (através do nome da classe). Caso o desenvolvedor esteja tentando invocar um membro não estático de um contexto estático, ocorre o erro `CallingNonStaticMemberFromStatic`. Caso o desenvolvedor esteja tentando chamar um membro estático a partir de um contexto não estático, é lançado o erro `CallingStaticMemberFromNonStatic`. O exemplo do Quadro 28 demonstra algumas destas situações.

```
001   public class A {
002       public int instance_X;
003       public static int static_X;
004       private int private_X;
005   }
006
007   public class B {
008      public void teste(A a) {
009         int val;
010         val = a.instance_A; //Forma correta
011         val = A.instance_A; //Erro acessando de forma estatica
012         val = A.static_X;   //Forma correta
013         val = a.static_X;   //Erro acessando por instancia
014         val = a.private_X;  //Erro de visibilidade
015       }
016   }
```

Quadro 28 – Exemplos de erros de invocação de membros

Para fazer o uso de orientação a objetos é primordial que se possa criar instâncias de classes. Esta funcionalidade é implementada na linguagem proposta através da expressão `new`. Esta expressão é constituída pelo nome da classe que se deseja instanciar e os parâmetros de seu construtor. A linguagem proposta não permite a definição de construtores. Todas as classes criadas nesta linguagem contém apenas um construtor padrão (público e sem parâmetros). Porém, é possível criar instâncias de classes definidas em outras linguagens da plataforma .NET e estas podem conter construtores com parâmetros. A passagem de parâmetros para construtores é igual à chamada de um método.

Valores constantes também são expressões. Literais como `1`, `2` e `3` são expressões do tipo `int` por exemplo. Os tipos que possuem literais são `string`, `char`, `bool`, `datetime`, `timespan`, `double` e `int`.

3.2 ESPECIFICAÇÃO DO COMPILADOR

Nesta seção é especificado o compilador para a linguagem proposta. Este tem a função de verificar erros no código escrito pelo desenvolvedor e gerar código MSIL. Por fim, o compilador executa o montador ILAsm da plataforma .NET para gerar o *assembly* referente ao código fonte.

3.2.1 Requisitos

Os requisitos do compilador são:

a) reportar erros léxicos, sintáticos e semânticos;
b) gerar código MSIL a partir de um conjunto de arquivos fonte;
c) gerar um *assembly* utilizando o montador ILAsm da plataforma .NET;
d) ser implementado na linguagem C# utilizando o ambiente de desenvolvimento Visual Studio .NET.

3.2.2 Geração de código

A geração de código MSIL é feita depois da verificação semântica do código fonte. Todas as estruturas da linguagem proposta geram este código a partir de uma árvore anotada (resultado da análise semântica).

O Quadro 29 mostra um exemplo de um comando escrito na linguagem proposta e sua equivalência em MSIL. A geração de código para os demais recursos da linguagem estão detalhados no Apêndice C.

EXEMPLO	**MSIL**

namespace org.furb.tcc;
 	
using System.String;
using System.Console;
 	
public class HelloWorld {
 	
  public static void main(String[]
args)
  {
  Console.WriteLine("Hello, World!");
  }
}
	.namespace org.furb.tcc
	{
	  .class public ansi auto HelloWorld
	  extends [mscorlib] System.Object
	  {
	  .method public void .ctor()
	  cil managed
	  {
	  ret
	  }
	  .method public static hidebysig
	  void main(string[] args) cil managed
	  {
	  .entrypoint
	  ldstr "Hello World!"
	  call void
	  [mscorlib] System.Console
	  ::WriteLine(string)
	  ret
	  }
	  }
	}

Quadro 29 – Exemplo de código gerado

3.2.3 Modelagem

Para modelar o compilador foi utilizada a *Unified Modeling Language* (UML). Para desenvolver os diagramas de caso de uso, classes, seqüência e implantação deste protótipo foi utilizada a ferramenta *Enterprise Architect* (EA).

3.2.3.1 Diagrama de caso de uso

A Figura 2 apresenta o diagrama de caso de uso do compilador desenvolvido junto com os requisitos que o mesmo atende.

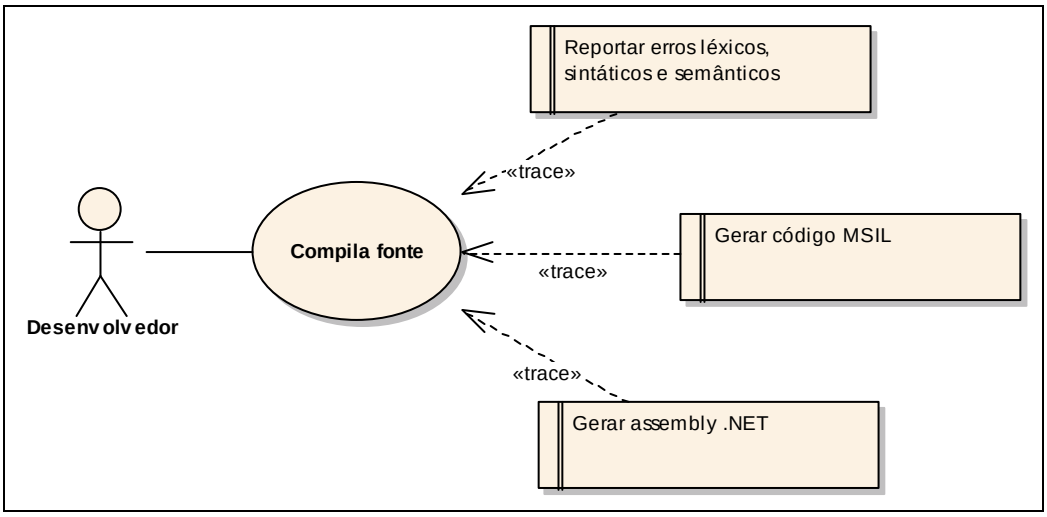


Figura 2 – Diagrama de casos de uso

O compilador possui apenas um caso de uso (`Compila fonte`) que é detalhado no Quadro 30.

Descrição	Verifica se os arquivos fonte possuem erros e gera um *assembly* .NET
Pré-condições	1. O desenvolvedor deve escrever os arquivos fonte no editor de sua preferência. 2. O desenvolvedor deve salvar os arquivos fonte com a extensão ".wak" em um diretório comum. 3. O desenvolvedor deve executar o compilador informando os diretórios de entrada e de saída, o nome do arquivo fonte, as referências e o tipo do *assembly*, através de parâmetros.
Fluxo principal	1. O compilador executa as análises léxica, sintática e semântica. 2. O compilador gera código MSIL. 3. O compilador executa o montador ILAsm para gerar um *assembly* .NET.
Fluxo alternativo	Não há.
Fluxo de exceção	1. Caso ocorram erros nos arquivos fonte, estes são informados ao desenvolvedor para que ele faça a correção.
Pós-condições	1. É gerado um assembly .NET ".DLL" ou ".EXE".
Requisitos atendidos	1. Reportar erros léxicos, sintáticos e semânticos. 2. Gerar código MSIL. 3. Gerar *assembly* .NET.

Quadro 30 – Caso de uso `Compila fonte`

3.2.3.2 Diagramas de classes

O protótipo foi dividido em 6 projetos enumerados abaixo:

a) `Wakizashi`: aplicação *console* responsável pela interação com o usuário;

b) `Waki`: biblioteca de classes que coordena o processo de compilação e geração de código;

c) `Waki.Common`: biblioteca de classes que possui funcionalidades comuns entre os módulos;

d) `Waki.Comp`: biblioteca de classes responsável pelas análises léxica, sintática e semântica;

e) `Waki.Gen`: biblioteca de classes responsável pela geração de código MSIL;

f) `System.Waki`: biblioteca de classes para execução de *assemblies* criados pelo compilador.

A seguir é apresentada uma visão geral das classes utilizadas para implementar cada um destes projetos e seus principais métodos. Apenas as principais classes estão representadas nos diagramas a seguir pois não foi feita a completa modelagem da solução já que esta seria muito extensa.

O projeto `Wakizashi` é uma aplicação do tipo *console* onde o usuário entra com os parâmetros para a compilação. Este é composto pelas classes `EntryPoint` e `ArgumentsHelper` apresentadas na Figura 3. A primeira é a entrada da aplicação *console* responsável por chamar o compilador, e a segunda trata os argumentos informados por linha de comando.

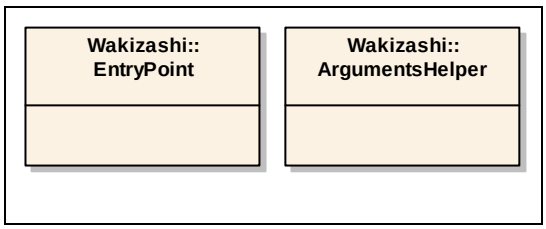


Figura 3 – Classes do projeto `Wakizahsi`

O projeto `Waki.Common` tem suas as classes relacionadas no diagrama da Figura 4, onde a classe `Project` representa um projeto criado a partir da linguagem proposta, e as demais classes representam erros de compilação. Entende-se por projeto: um conjunto de

arquivos fonte, diretório e nome de arquivo de saída, referências e *flag* indicando se para o projeto deve ser gerada uma biblioteca de classes ou um *assembly* executável. A classe `SemanticException` é a exceção lançada quando ocorre um erro semântico. Este possui um código de erro, representado pela enumeração `SemanticError`, uma origem (`IErrorOrigin`) e uma descrição (`ErrorDescriptionAttribute`).

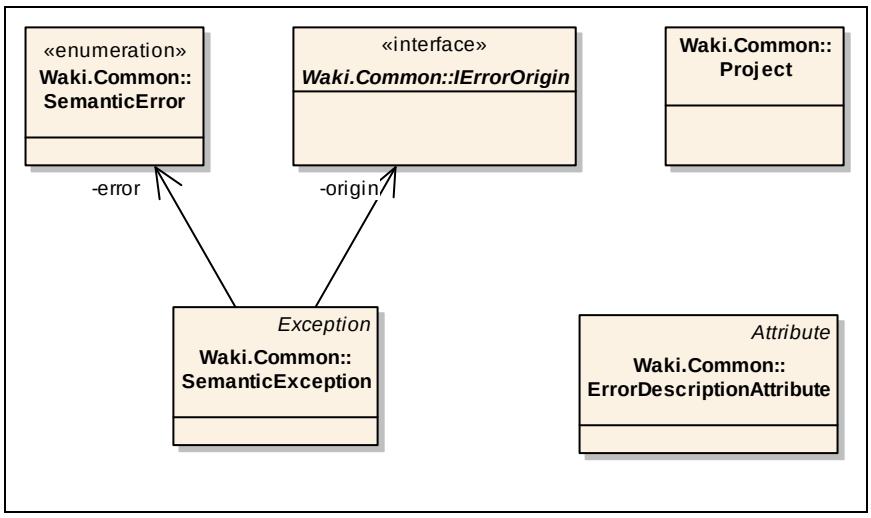


Figura 4 – Classes do projeto `Waki.Common`

A descrição dos erros e seu código são ligados a partir de *custom attributes* conforme o Quadro 31. A classe de exceção `SemanticException` busca a mensagem de erro através do item de enumeração `SemanticError` através de reflexão.

```
public enum SemanticError
{
	[ErrorDescription("Símbolo '{0}' não declarado.")]
	SymbolNotDeclared,

	[ErrorDescription("Modificador '{0}' duplicado.")]
	DuplicatedModifier,
	...
}
```

Quadro 31 – Exemplo de definição de erros na classe `SemanticError`

O projeto `Waki` coordena toda a compilação de um projeto. O diagrama da Figura 5 mostra as principais classes responsáveis pela compilação. A classe responsável por iniciar a compilação é a classe `Compiler`. Esta contém os métodos estáticos `Compile(Project)` e `Compile(Project, bool)`. A classe em questão também carrega as referências externas através de reflexão para que possam ser utilizadas na linguagem proposta. As demais classes

deste diagrama estão relacionadas no Quadro 32.

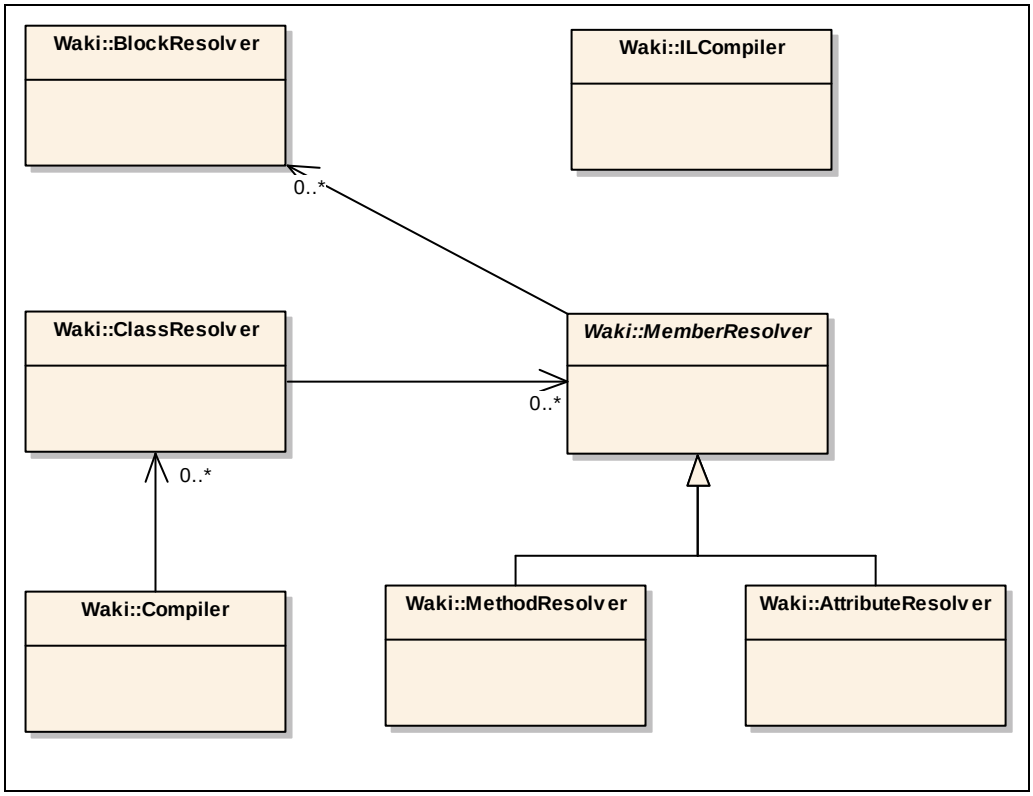


Figura 5 – Classes do projeto `Waki`

NOME	**FUNÇÃO**	**MÉTODOS**
`Compiler`	`Compilar projetos`	`1. bool Compile(Project project)` `2. bool Compile(Project project, bool debug)`
`ClassResolver`	`Compilar classes`	`1. void Resolve()` `2. IList GetUnresolvedMembers()`
`MemberResolver`	`Resolver membro de uma classe`	`1. void Resolve()` `2. IList GetUnresolvedBlocks()`
`MethodResolver`	`Compilar métodos`	`1. void Resolve()` `2. IList GetUnresolvedBlocks()`
`AttributeResolver`	`Compilar atributos`	`1. void Resolve()` `2. IList GetUnresolvedBlocks()`
`BlockResolver`	`Compilar comandos`	`1. void Resolve()`
`ILCompiler`	`Chamar montador ILAsm`	`1. static bool GenerateAssembly(string inpath, string outpath, bool dll, bool debug)`

Quadro 32 – Descrição das classes do projeto `Waki`

As classes principais do projeto `Waki.Comp` estão representadas no diagrama da Figura 6. A classe `WakiParser` é responsável pelas análises léxica e sintática, enquanto as demais são responsáveis pela análise semântica. Suas funcionalidades estão descritas no Quadro 33.

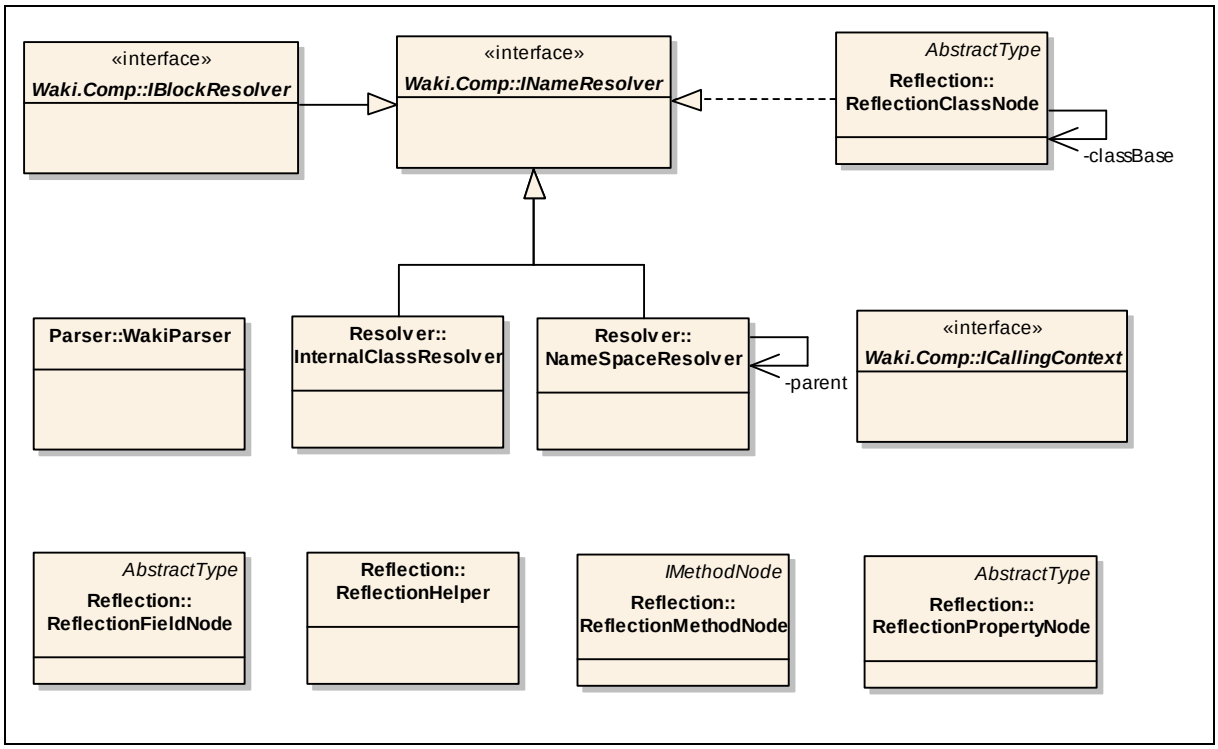


Figura 6 – Classes do projeto `Waki.Comp`

NOME	**FUNÇÃO**	**MÉTODOS**
WakiParser	Fazer as análises léxica e sintática de um arquivo fonte	1. CompilationUnit Parse(StreamReader input)
INameResolver	Interface base para uma tabela de símbolos	1. ITypedNode Resolve(Token name) 2. ITypedNode Resolve(Token name, ITypedNode[] args)
NameSpaceResolver	Resolver itens contidos dentro de um namespace (tabela de símbolos)	1. void AddClass(string fullname, IClassNode clazz)
IBlockResolver	Interface para resolver uma tabela de símbolos dentro do contexto de um bloco	1. void Add(string name, IType type)
ICallingContext	Contexto do item que está sendo resolvido	1. ITypedNode GetThis() 2. void Check(IContextualizedNode node)
InternalClassResolver	Resolver os métodos e atributos acessíveis dentro da instância de uma classe	-
ReflecionClassNode	Representar uma classe externa	-
ReflectionFieldNode	Representar um atributo de uma classe externa	-
ReflectionPropertyNode	Representar uma propriedade de uma classe externa	-
ReflecionMethodNode	Representar um método de uma classe externa	-
ReflectionHelper	Auxiliar o uso de reflexão	-

Quadro 33 – Descrição das classes do projeto `Waki`

O resultado do método `WakiParser.Parse(StreamReader)` é uma árvore sintática.

Parte desta árvore é representada pelo diagrama da Figura 7. Os demais nós desta árvore não são detalhados devido à sua extensão.

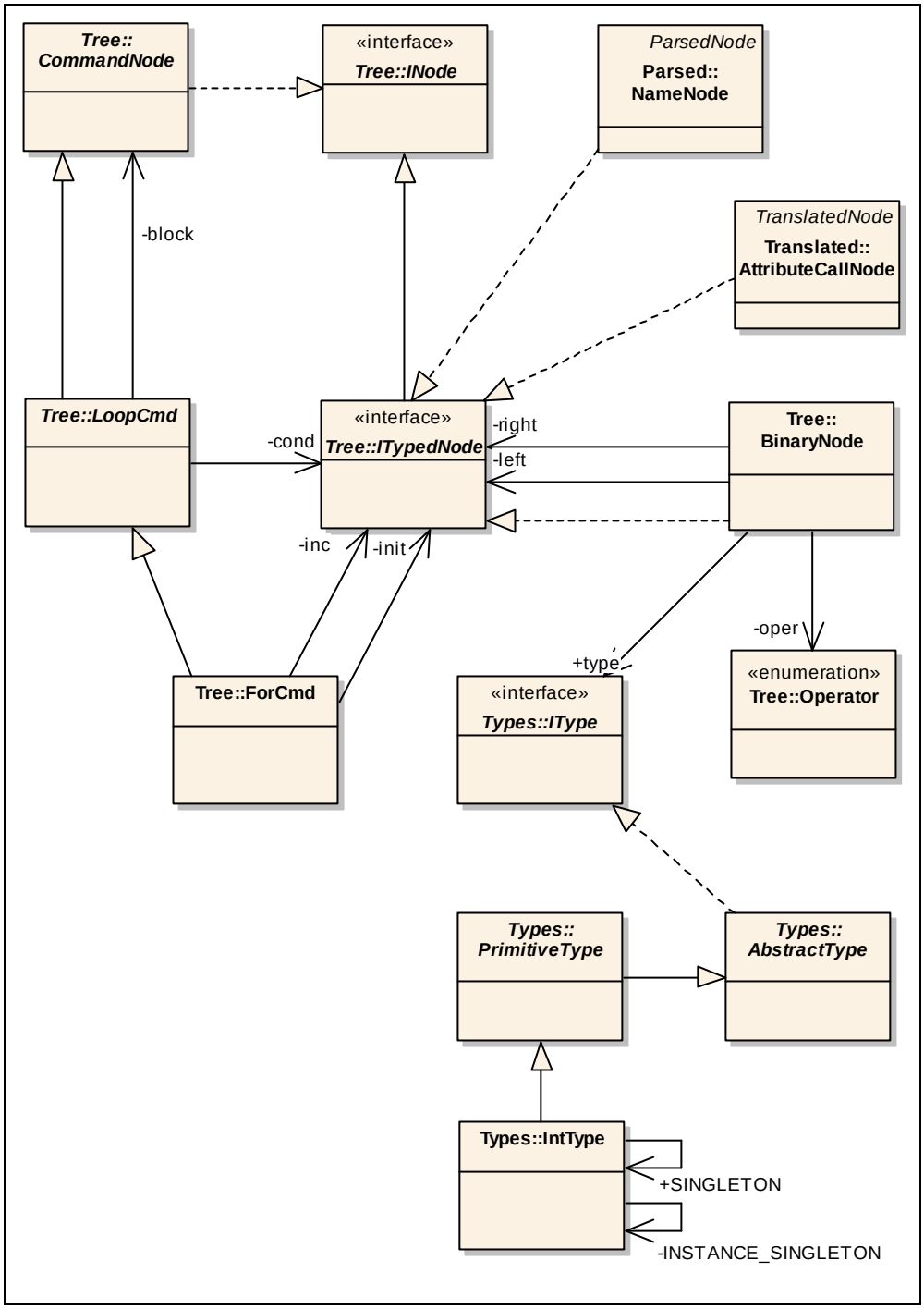


Figura 7 – Classes da árvore sintática

Todos os nós da árvore sintática implementam a interface `INode`. Os nós que representam uma expressão implementam a interface `ITypedNode`. A análise semântica opera nesta árvore, transformando-a em uma árvore anotada. Para que isto aconteça, deve ser

chamado o método `INode.Compile(INameContext, ICallingContext)` do nó principal (`CompilationUnit`). O primeiro argumento deste método representa a tabela de símbolos do compilador, e o segundo é o contexto da compilação. Este é responsável por verificar a visibilidade de um membro e a instância de onde está sendo chamado.

Os nós da árvore sintática que implementam a interface `ITypedNode` possuem a propriedade `Type` que retorna um objeto de uma classe que implementa a interface `IType`. Esta por sua vez possui o método `GetResolver()` que retorna um objeto da classe `INameResolver` que representa uma tabela de símbolos. A tabela de símbolos para um compilador de uma linguagem orientada a objetos é um grafo já que a resolução de nomes é feita respeitando o escopo de um nome. A Figura 8 representa a tabela de símbolos carregada durante a compilação do código fonte apresentado no Quadro 34.

```
namespace org.furb.empresa;

using org.furb.rh.Funcionario;

public class Empresa {
    public string razaoSocial;
    public string nomeFantasia;
    public string cnpj;
}

public class Departamento {
    public string nome;
    public Funcionario getResponsavel() {
        //Lógica para selecionar o responsável do
        //departamento
        return null;
    }
}

namespace org.furb.rh;

using org.furb.empresa.Departamento;

public class Pessoa {
    public string nome;
}

public class Funcionario extends Pessoa {
    public Departamento departamento;
}
```

Quadro 34 – Código exemplo para resolução de nomes

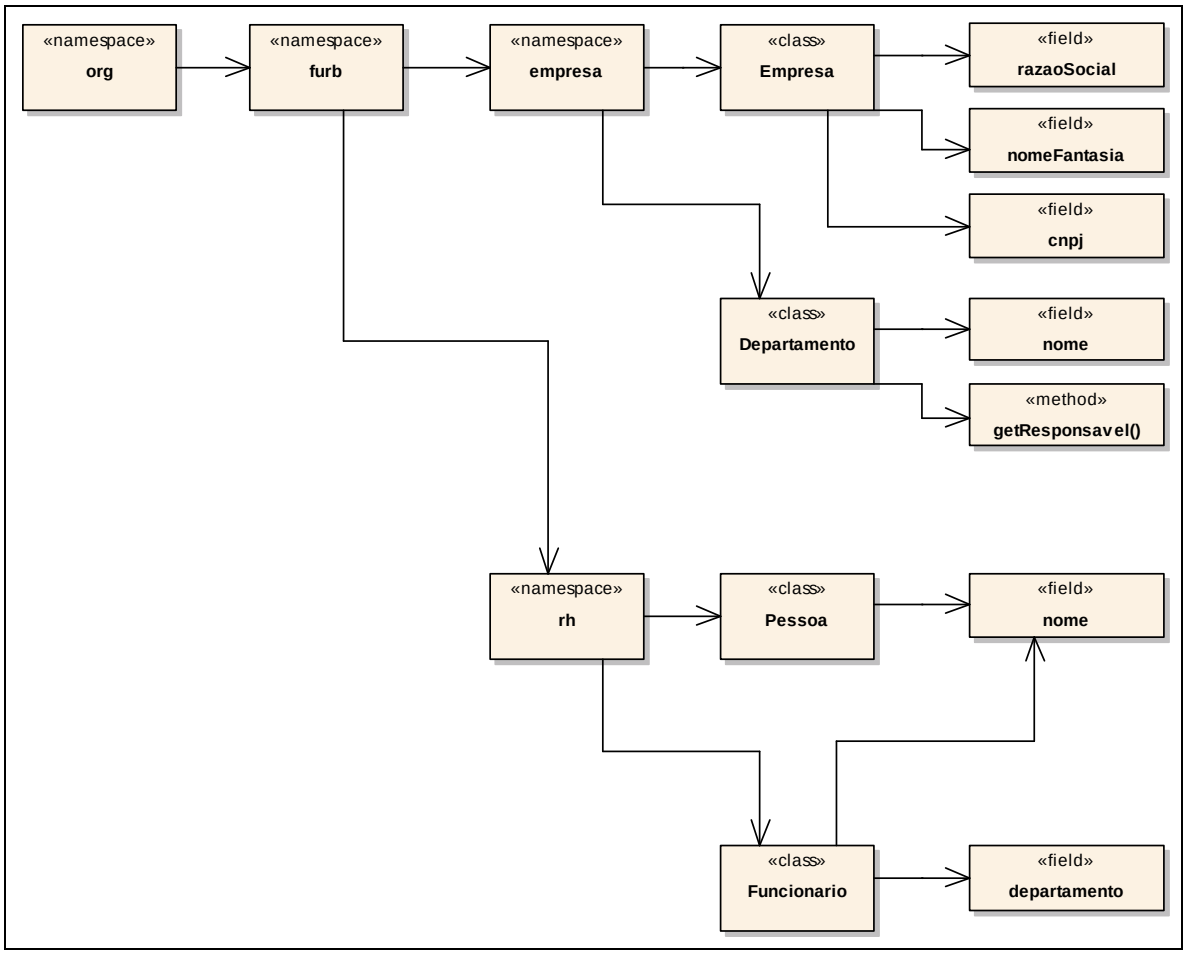


Figura 8 - Tabela de símbolos exemplo

A última etapa da compilação efetuada pelo compilador proposto é a geração de código intermediário MSIL. Para fazer isso é percorrida a árvore anotada através de visitadores implementados segundo o *design pattern visitor*. No diagrama da Figura 9 são apresentados os visitadores da árvore anotada e as classes auxiliares para gerar código. Destas destaca-se a classe `ILBuilder` que é o ponto de entrada para a geração de código MSIL.

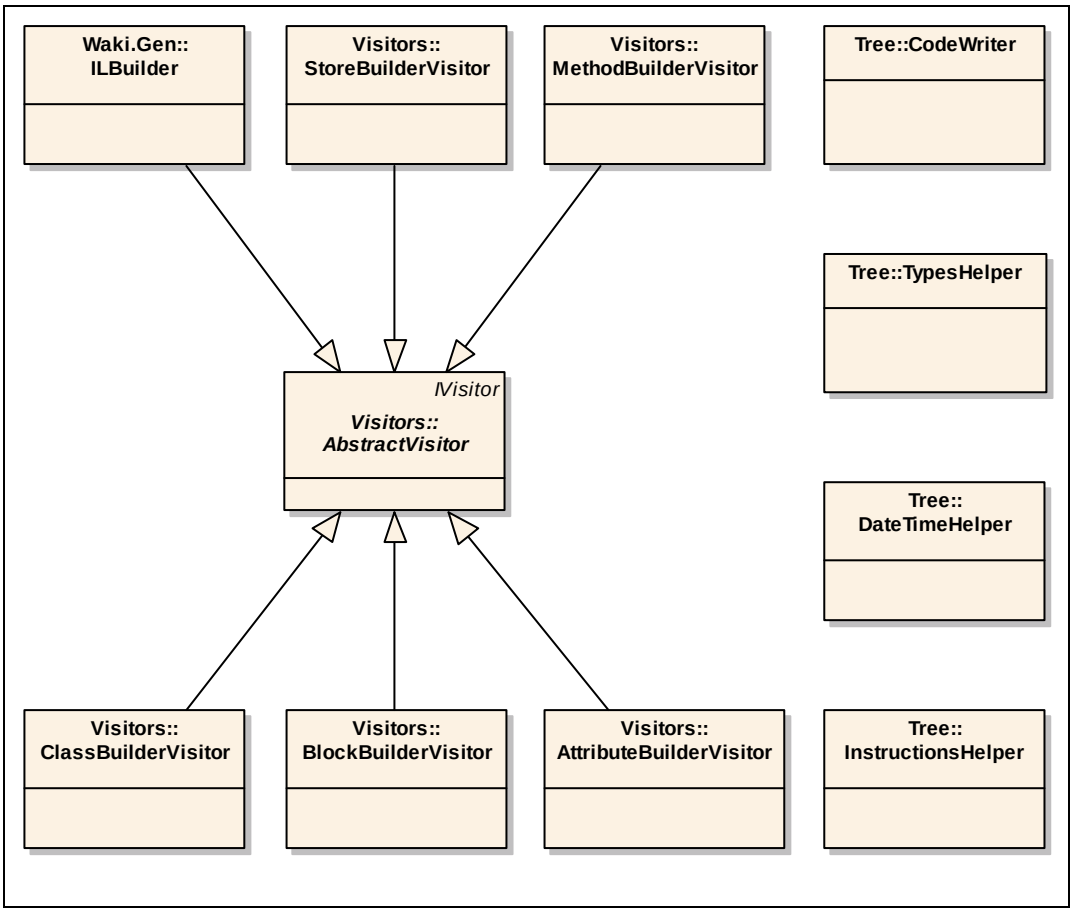


Figura 9 – Classes do projeto `Waki.Gen`

A medida que a árvore anotada (gerada a partir da análise semântica) é percorrida, é gerada uma outra árvore. Esta representa o código MSIL referente ao programa fonte de entrada. O nó principal desta árvore é representado pela instância da classe `ILAssembly`. A Figura 10 apresenta as classes utilizadas para formar a árvore de código MSIL.

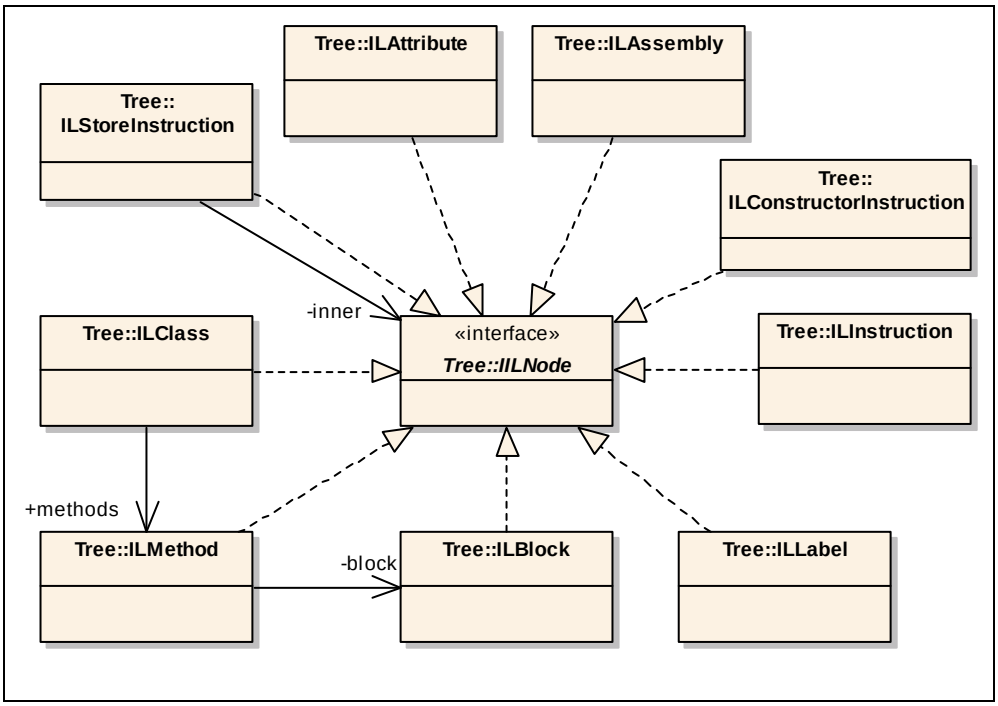


Figura 10 – Classes da árvore MSIL

Os quadros 35 e 36 detalham os métodos das classes utilizadas para a geração de código que estão representadas nos diagramas das figuras 9 e 10.

NOME	FUNÇÃO	MÉTODOS
ILBuilder	Coordenar a geração de código	1. ILBuilder(Project project, IList units) 2. Execute(StreamWriter out)
ClassBuilderVisitor	Criar uma classe MSIL a partir de uma classe escrita na linguagem proposta	1. ILClass GetILClass()
AttributeBuilderVisitor	Criar um atributo MSIL	1. ILAttribute GetILAttribute()
MethodBuilderVisitor	Criar um método MSIL	1. ILMethod GetILMethod()
BlockBuilderVisitor	Criar um bloco MSIL	1. ILBlock GetILBlock()
InstructionsHelper	Auxiliar a classe BlockBuilderVisitor a criar comandos MSIL a partir da linguagem proposta	1. string GetPushForType(IType t) 2. string GetPushTrue() 3. string GetPushFalse() 4. string GetPushChar() 5. IILNode[] GetOperationInstruction(IType leftType, IType rightType, Operator op) 6. IILNode[] GetOperationInstruction(IType operanType, Operator op) 7. object GetOpKey(IType leftType, IType rightType, Operator op) 8. object GetOpKey(IType operandType, Operator op)
DateTimeHelper	Fazer a conversão das literais dos tipos datetime e timespan	1. IntegerLiteral[] GetDateTimeArgs(string literal) 2. IntegerLiteral[] GetTimeStampArgs(string literal)
TypesHelper	Converter tipos da linguagem proposta para MSIL	1. string GetILType(IType type) 2. bool IsVoid(IType type)
AbstractVisitor	Classe base para todos os visitadores	-
StoreBuilderVisitor	Criar uma instrução para guardar um valor	1. ILStoreInstruction GetInstruction()
CodeWriter	Escrever código identado	1. void Ident() 2. void Unident() 3. void NewLine() 4. void Write(string s) 5. void Write(string format, object[] args) 6. void WriteLine(string s) 7. void WriteLine(string format, object[] args)

Quadro 35 – Descrição das classes para geração de código

NOME	FUNÇÃO	MÉTODOS
ILNode	Interface base para todos os nós da árvore de código MSIL	1. void AppendTo(CodeWriter writer)
ILAssembly	Representar o código MSIL para a criação de um *assembly*	1. void AddClass(ILClass ilClass)
ILClass	Representar o código MSIL para a criação de uma classe	1. void AddAttribute(ILAttribute att) 2. void AddMethod(ILMethod method)
ILMethod	Representar o código MSIL para a criação de um método	1. void SetBlock(ILBlock block)
ILAttribute	Representar o código MSIL para a criação de um atributo	-
ILBlock	Representar o código MSIL para a criação de comandos	Esta classe possui vários métodos para representar instruções MSIL, tais como ldloc, stloc, br, brfalse, etc.
ILInstruction	Representar uma instrução MSIL	-
ILStoreInstruction	Representar uma instrução de atribuição	-
ILConstructorInstruction	Representar uma invocação a um construtor	-
ILLabel	Representar um label para a linguagem MSIL	-

Quadro 36 – Descrição das classes da árvore MSIL

Com o intuito de deixar a linguagem mais flexível e facilitar a manutenção do compilador, é permitido que algumas instruções façam uso de uma biblioteca externa para a sua execução. É o caso da instrução `like` que para sua execução utiliza o método `StringHelper.Like(string, string)`. As classes deste projeto são descritas no diagrama da Figura 11.

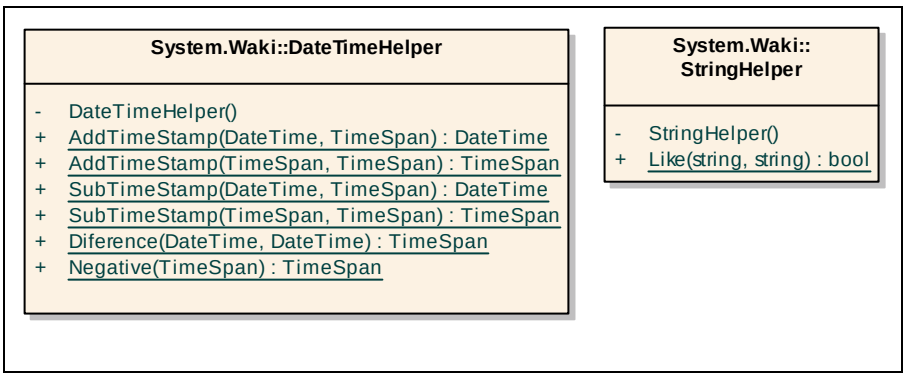


Figura 11 – Classes do projeto `System.Waki`

3.2.3.3 Diagrama de seqüência

O diagrama de seqüência da Figura 12 detalha o método `Compile` da classe `Compiler`. Este método coordena toda a compilação, chamando os módulos de análise léxica e sintática, análise semântica e geração de código MSIL.

Este método também é responsável por criar a tabela de símbolos que é carregada e utilizada na análise semântica. Para resolver problemas de referência circular, a análise semântica é feita em três fases. A primeira delas resolve as classes. Na fase seguinte onde são resolvidos os membros da classe, a tabela de símbolos já contém a definição de cada classe. Na última fase, são resolvidos os blocos de comando com a tabela de símbolos completa, incluindo os membros das classes.

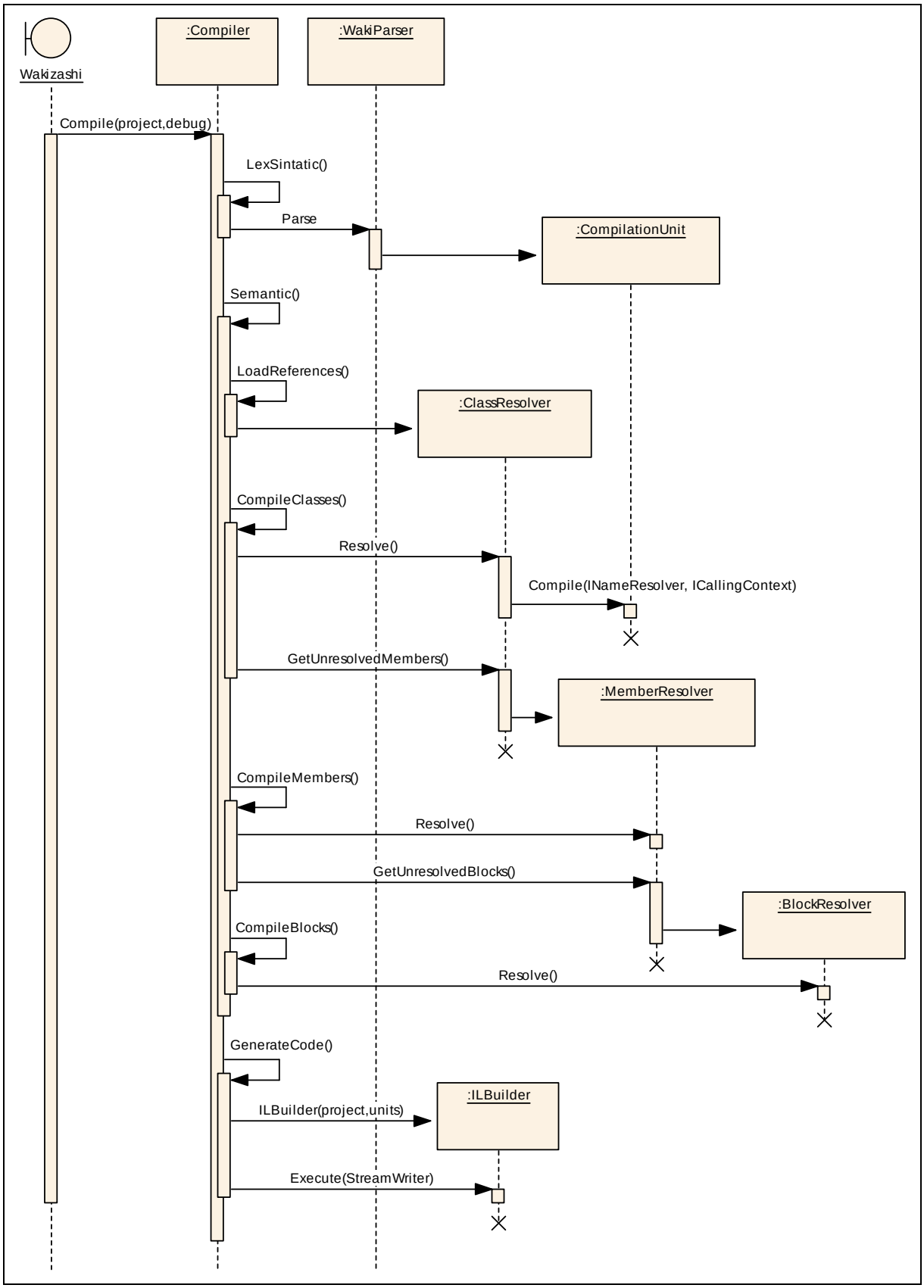


Figura 12 - Visão geral da compilação

3.2.3.4 Diagrama de implantação

O diagrama da Figura 13 mostra as dependências entre os *assemblies*. A interface `Wakizashi` é a aplicação *console* utilizada pelo desenvolvedor para compilar seu código e gerar um *assembly*. E a interface `Runtime` representa a execução de um *assembly* gerado pelo compilador da linguagem proposta.

O código gerado a partir da linguagem especificada utiliza o *assembly* `System.Waki` para sua execução. Este deve ser instalado no GAC da plataforma .NET. A dependência dos *assemblies* gerados é representada no diagrama abaixo.

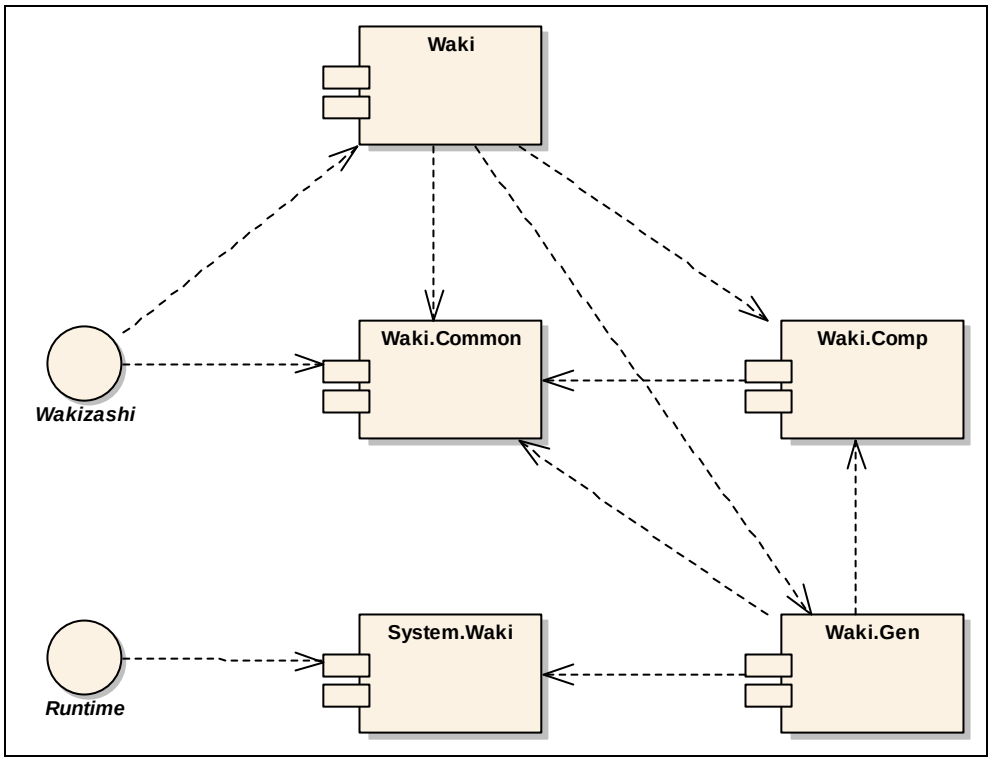


Figura 13 – Diagrama de implantação

3.3 IMPLEMENTAÇÃO

Nesta seção estão descritas as ferramentas, técnicas e padrões utilizados para a implementação do compilador.

3.3.1 Técnicas e ferramentas utilizadas

A implementação do compilador foi feita no ambiente de desenvolvimento Visual Studio .NET 2003 com a linguagem C#. O código para os analisadores léxico e sintático foram gerados utilizando a ferramenta JavaCCCS. Para varrer a árvore anotada com o intuito de gerar fontes, foi feito o uso do *design pattern* conhecido por *visitor*. Para compilar e gerar *assemblies* a partir do código MSIL gerado, foi utilizado o ILAsm da plataforma .NET.

3.3.1.1 JavaCCCS

Os analisadores léxico e sintático foram gerados a partir da ferramenta JavaCCCS. Da mesma forma que a sua versão para a plataforma Java (JavaCC), esta ferramenta utiliza a BNF estendida, porém gera código C#. A implementação da EBNF para o JavaCCCS permite que junto com a gramática da linguagem seja definido código C#. Este recurso na presente implementação é aproveitado para carregar a árvore sintática. O Quadro 37 mostra parte da definição da gramática utilizando o JAVACCS.

```
CommandNode whileCmd() : {
  WhileCmd ret = new WhileCmd();
} {
  "while" "(" ret.Cond = expression() ")"
  ret.Block = blockCmd()
  {return ret;}
}
```

Quadro 37 – Exemplo da gramática para o JavaCCCS

3.3.1.2 *Design patterns*

Os motivos pelos quais devem ser estudados e utilizados *design patterns*, segundo Shalloway e Trott (2001, p. 63-65), são dois: reutilizar soluções e estabelecer terminologia

comum. O primeiro motivo diz respeito a aprender com a experiência dos outros. Não há necessidade de reinventar soluções para problemas recorrentes. O segundo motivo é melhorar a comunicação dentro de equipes de desenvolvimento. *Design patterns* disponibilizam um ponto de referência comum durante as fases de análise e projeto de uma solução.

Existem diversos *design patterns* para resolver diferentes situações do dia a dia do desenvolvimento de software. Esta seção se restringe a detalhar os padrões utilizados para o desenvolvimento do trabalho. Estes são três: *singleton*, *interpreter* e *visitor*.

O *pattern singleton* é usado para garantir apenas uma instância de determinada classe para a aplicação (SHALLOWAY; TROTT, 2001, p. 255). Outro padrão utilizado é o *interpreter*. Este tem a intenção de tratar objetos relacionados entre si da mesma maneira (COOPER, 1998, p. 145). O *interpreteter* é responsável pelo tratamento de contexto da compilação. Para isto, todos os nós da árvore sintática implementam a interface `INode` que possui o método `Compile(INameResonver, ICallingContext)` que efetua a análise semântica. Da mesma maneira, este *design pattern* também é utilizado para a árvore de expressões utilizando a interface `ITypedNode`.

O último padrão de desenvolvimento utilizado é o *visitor*. Este pode ser utilizado quando se possui uma estrutura hierárquica razoavelmente estável para a qual devem ser adicionadas novas funcionalidades (COOPER, 1998, p. 210). O padrão *visitor* é utilizado na geração de código intermediário. Para percorrer a árvore anotada e criar a árvore de operações MSIL foram implementados visitadores.

3.3.2 Operacionalidade da implementação

A ferramenta desenvolvida é uma aplicação do tipo *console* feita para a plataforma .NET. Nela o desenvolvedor deve informar o diretório onde se encontram os arquivos fonte

que devem ter a extensão .WAK, suas referências externas (*assemblies* .NET) e o diretório e nome do arquivo de saída. Por padrão, o compilador gera um *assembly* executável .EXE, mas também é possível gerar uma biblioteca de classes, arquivo .DLL. Para isto, basta definir um parâmetro para o compilador (Quadro 38). Os parâmetros são apresentados quando o usuário executa a ferramenta com algum parâmetro inválido.

PARÂMETRO	EXEMPLO	DESCRIÇÃO
srcDir	.\project\src\	Diretório onde estão localizados os arquivos fonte .WAK
outDir	.\project\bin\	Diretório onde deve ser gerado o *assembly*
name	Teste	Nome do arquivo de saída (sem extensão)
refs	Biblioteca.dll	Referências utilizadas (*assemblies*)
dll	-	*Flag* indicando se deve ser gerada uma biblioteca

Quadro 38 – Parâmetros do compilador

A sintaxe para a definição dos parâmetros respeita o seguinte formato: `Wakizashi.exe <srcDir> /out:<outDir> /name:<name> [/refs:<refs>] [/dll]`, onde os caracteres `[` e `]` indicam que o parâmetro é opcional. A Figura 14 mostra a execução do compilador.

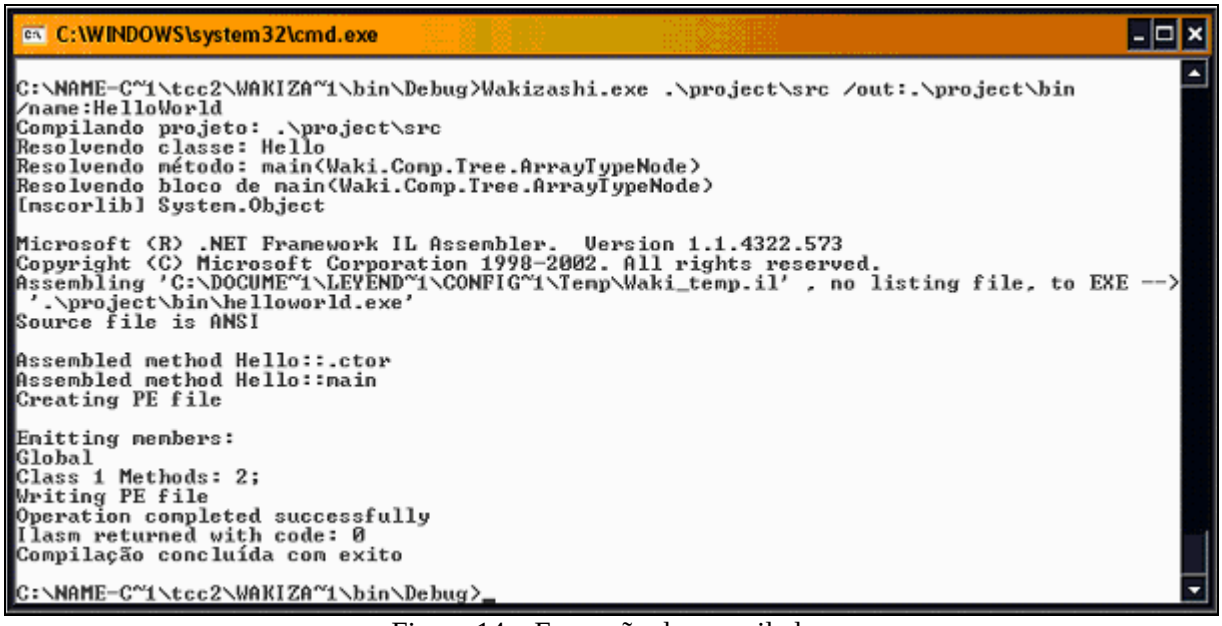


Figura 14 – Execução do compilador

3.3.2.1 Estudo de caso

Para validar o uso da linguagem desenvolvida, foi criado o protótipo de um sistema

para o controle de um WebCafe. Este sistema foi divido em três camadas: camada de interface, camada de negócio e validação e camada de dados.

As camandas de interface e dados foram desenvolvidas utilizando a linguagem C#, enquanto a camada de negócio e validação foi desenvolvida fazendo uso da linguagem proposta e utilizando o compilador para gerar uma biblioteca de classes utilizada pela camada de interface. As classes envolvidas na solução são apresentadas no diagrama da Figura 15.

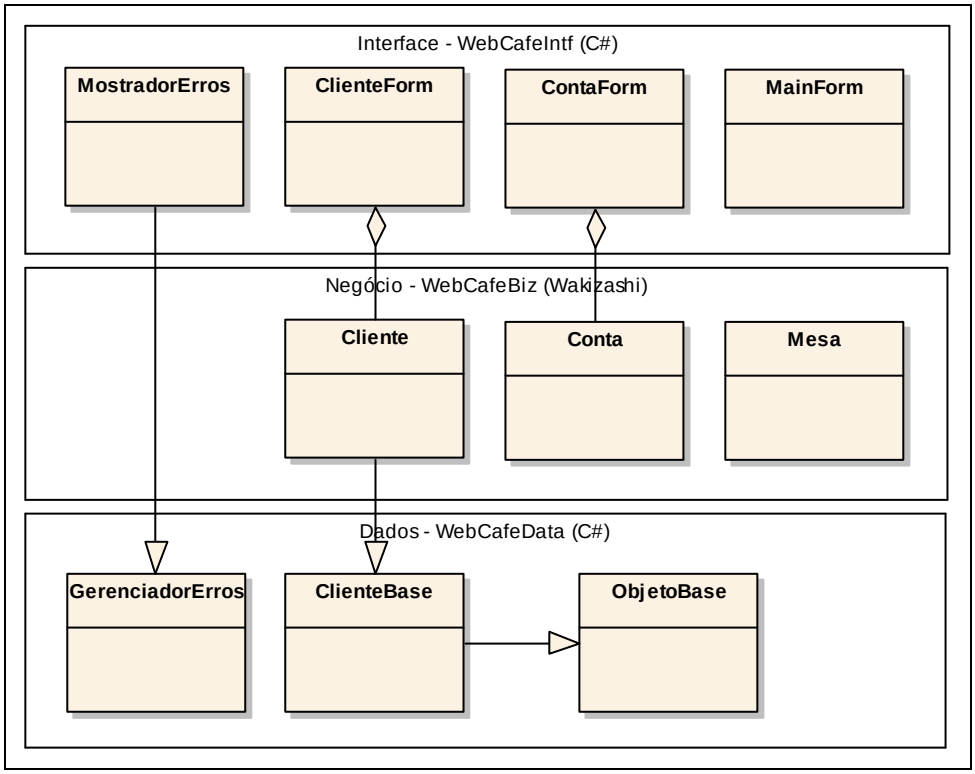


Figura 15 – Classes WebCafe

3.4 RESULTADOS E DISCUSSÃO

A linguagem definida possui os recursos básicos de orientação a objetos, tais como herança simples, visibilidade de classes, métodos, atributos e polimorfismo. Herança múltipla e definição de interfaces não são suportados na linguagem proposta nem outros recursos da plataforma .NET como o tratamento de exceções, definição de *custom attributes* e propriedades. A linguagem C# também permite a definição de *structs* e *enums* definidos pela

CTS que a linguagem proposta não implementa. Outro recurso básico que a linguagem não implementa é o suporte a vetores. O Quadro 39 apresenta um comparativo entre a linguagem de programação C# e a linguagem especificada neste trabalho. Este quadro foi elaborado com base na bibliografia existente a respeito da linguagem C# e no desenvolvimento do estudo de caso com a linguagem proposta.

C#	**Wakizashi**
Rápida curva de aprendizagem	Produtividade p/ camada de negócio
Recuperação de erros semânticos	Um erro por compilação
Expressões primitivas	*datetme*, *timespan*, *like*
Interfaces, *enums* e *structs*	Apenas classes
Suporta totalmente CTS	Suporta parcialmente CTS
3 anos no mercado	-

Quadro 39 – Comparativo entre as LPs C# e Wakizashi

A implementação de expressões com os tipos `datetime` e `timespan`, assim como definição de literais se mostra muito produtiva para a definição de rotinas de validação de dados e regras de negócio. Outra facilidade para a consistência de dados que a linguagem especifica é o operador `like`. No Quadro 40 são apresentadas as características desejáveis em uma linguagem de programação e resultados obtidos com a linguagem proposta. Este quadro foi desenvolvido com base nos recursos que a linguagem proposta provê ao desenvolvedor.

Característica	**Resultado**
Legibilidade	Bom
Redigibilidade	Bom
Confiabilidade	Regular
Eficiência	Independe da linguagem
Aprendizado	Regular
Ortogonalidade	Bom
Reusabilidade	Bom
Modificabilidade	Bom
Portabilidade	Independe da linguagem

Quadro 40 – Características desejadas atendidas pela LP proposta

O uso da ferramenta JavaCCCS mostrou-se bastante produtivo para o desenvolvimento do compilador, assim como a criação da árvore de operações MSIL facilitou e flexibilizou a geração de código. O compilador é limitado no que diz respeito a tratamento de erros devido a

não ser capaz de produzir mais de uma mensagem de erro por compilação, ou seja, a execução do compilador é abortada na ocorrência de um erro, mostrando apenas um erro por compilação.

4 CONCLUSÕES

Os objetivos principais deste trabalho foram atendidos com sucesso. No desenvolvimento deste foi especificada uma linguagem de programação orientada a objetos que inclui funcionalidades não existentes em outras linguagens de programação da plataforma .NET como as operações com datas e o operador `like`. Também foi implementado com êxito um compilador para a linguagem especificada.

Outra característica desejada na linguagem é ser produtiva para o desenvolvimento da camada de negócios de uma aplicação. Este requisito foi atendido parcialmente já que a linguagem possui algumas limitações essenciais para o desenvolvimento de software tais como a definição de vetores e interfaces. Porém, algumas limitações, como a herança múltipla, também não são implementadas pelas linguagens de programação existentes no mercado, nem são especificadas pelo CTS.

O desenvolvimento deste trabalho mostrou que é possível o desenvolvimento de uma LP orientada a objetos para a plataforma .NET que possa interagir com outras linguagens da mesma plataforma. Este fato torna viável a extensão desta LP para que conquiste seu objetivo final de tornar o desenvolvimento da camada de negócios de uma aplicação comercial menos complexo e mais produtivo.

4.1 EXTENSÕES

Para que a linguagem de programação descrita neste trabalho se torne produtiva para o desenvolvimento de aplicações, deve ser implementada por completo a especificação CTS. Além disso, são propostas algumas melhorias, tais como:

a) herança múltipla: para definir problemas complexos pode surgir a necessidade do desenvolvedor utilizar herança múltipla. Hoje, o CTS e o MSIL não suportam múltipla herança. Porém pode ser estudada uma maneira de resolver esta questão utilizando interfaces e duplicando o código implementado na classe base;
b) controle de concorrência: freqüentemente o desenvolvedor de aplicações se vê diante de problemas de concorrência. A LP proposta neste trabalho pode ser estendida para que incorpore instruções que facilitem o controle de concorrência;
c) tratamento de exceção: para controle de exceção deve ser possível criar estruturas na linguagem para lançar e capturar exceções. Estas são geralmente tratadas pelas palavras reservadas *try*, *catch* e *finally*;
d) sobrecarga de operadores: linguagens de programação como C# e C++ permitem ao desenvolvedor criar tipos de dados bem como operadores para eles. Tendo em vista que isto permite uma flexibilidade melhor para a linguagem, também poderia ser implementado este recurso na linguagem proposta;
e) suporte a instruções SQL: o trabalho correlato apresentado por Kudzu (2005) tem como objetivo utilizar instruções SQL no C# utilizando *custom attributes* para estender a linguagem. A LP proposta pode incorporar instruções SQL de forma nativa (na definição da linguagem);
f) suporte a programação orientada a aspecto: a linguagem proposta como extensão deste trabalho pode incluir funcionalidades a fim de suportar programação orientada a aspecto.

Também são propostas algumas melhorias no compilador para melhorar a interação como usuário, entre elas:

a) recuperação de erros: para melhorar a produtividade do uso de um compilador, é necessário que o processo de compilação não pare no primeiro erro, mostrando

mais de uma mensagem, se houver, para o usuário. Para isto, é preciso fazer com que o compilador se recupere dos erros de compilação para continuar o processo;

b) ambiente de desenvolvimento: é sugerido que para futuras implementações deste trabalho seja melhorada a interface com o usuário. Isto pode ser feito implementando um ambiente de desenvolvimento integrado (IDE) ou através da integração do compilador com outra IDE já existente como o Visual Studio .NET;

c) depuração: para facilitar a tarefa de detecção de erro de lógica, é necessário que o usuário do compilador possa executar seu programa passo a passo, verificando cada estado de sua aplicação.

REFERÊNCIAS BIBLIOGRÁFICAS

BORBA, Lucas L. **Compilador C# e framework .NET**. [S.l.], 2004. Disponível em: <http://atlas.ucpel.tche.br/~barbosa/consico/consico3/artigos/a2.pdf>. Acesso em: 07 ago. 2005.

COOPER, Jamer W. **The design patterns Java companion**. Wokingham: Addison-Wesley, 1998.

GRUNE, Dick et al. **Projeto moderno de compiladores**: implementação e aplicações. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001.

HUNT, John. **Smalltalk and object orientation**: an intruduction. New York: Springer-Verlag Telos, 1997.

JAVA.NET. **JavaCC**. [S.l.], 2004. Disponível em: < https://javacc.dev.java.net/>. Acesso em: 15 nov. 2005.

KUDZU, Chaz Z. H. **Extending C# to support SQL syntax at compile time**. [S.l.], 2005. Disponível em: <http://www.codeproject.com/dotnet/CSharpSQL.asp>. Acesso em: 07 ago. 2005.

LIDIN, Serge. **Inside Microsoft .NET IL assembler**. Redmond: Microsoft Press, 2002.

LISBÔA, Maria L. B. **Modelos de linguagens de programação**. [Porto Alegre], 2004. Disponível em: <http://www.inf.ufrgs.br/aulas/mlp/material.html>. Acesso em: 09 ago. 2005.

LOUDEN, Kenneth C. **Compiladores**: princípios e práticas. São Paulo: Pioneira Thomson Learning, 2004.

MAHMOODI, Rahman. **3-tier architecture in C#**. [S.l.], 2005. Disponível em: <http://www.codeproject.com/useritems/Three_tier_Architecture.asp>. Acesso em: 07 ago. 2005.

MICROSOFT CORPORATION. **.NET solutions for developers**. [S.l.], 2005. Disponível em: <http://www.microsoft.com/net/developers.mspx>. Acesso em: 07 ago. 2005.

MICROSOFT RESEARCH. **Comega**. [S.l.], 2005. Disponível em: <http://research.microsoft.com/Comega/>. Acesso em: 07 ago. 2005.

RICHTER, Jeffrey. **Applied Microsoft .NET framework programming**. Redmond: Microsoft Press, 2002.

SHALLOWAY, Alan; TROTT, James R. **Design patterns explained**: a new perspective on object oriented design. Wokingham: Addison-Wesley, 2001.

SOURCE FORGE. **Project info**: JavaCCCS. [S.l.], 2005. Disponível em: <http://sourceforge.net/projects/javacccs/>. Acesso em: 09 ago. 2005.

STOECKER, Matthew A. **Developing windows-based applications**: with Visual Basic .NET and Visual C# .NET. 2. ed. Redmond: Microsoft Press, 2003.

THAI, Thuan; LAM, Hoang Q. **.NET framework essentials**. Sebastopol: O'Reilly, 2001.

VAREJÃO, Flávio. **Linguagens de programação**: Java, C e C++ e outras - conceitos e técnicas. Rio de Janeiro: Elsevier Editora Ltda, 2004.

WILHELM, Reinhard; MAURER, Dieter. **Compiler design**. Wokingham: Addison-Wesley, 1995.

WILSON, Leslie B.; CLARK, Robert G. **Comparative programming languages**. 2. ed. Workignham: Addison-Wesley, 1993.

APÊNDICE A – Mensagens de erros

A relação de mensagens de erro é apresentada no Quadro 41. As sequências `{0}` e `{1}` representam argumentos para as mensagens.

CODIGO	NOME	MENSAGEM DO ERRO
001	SymbolNotDeclared	Símbolo '{0}' não declarado
002	DuplicatedModifier	Modificador '{0}' duplicado
003	InvalidUsingNs	Declaração de 'using' inválida, era esperado um namespace
004	InvalidUsingClass	Declaração de 'using' inválida, era esperada uma classe
005	DuplicatedClassDefinition	Classe '{0}' já definida neste namespace
006	CannotConvertType	Não é possível converter do tipo '{0}' para o tipo '{1}'
007	InvalidBinaryOperation	Expressão binária inválida. Não é possível utilizar o operador '{2}' com os tipos '{0}' e '{1}'
008	InvalidUnaryOperation	Expressão unária inválida. Não é possível utilizar o operador '{0}' com o tipo '{1}'
009	DuplicatedVariableDeclaration	Variável local '{0}' já declarada neste escopo
010	MethodNotFound	Método '{0}' não declarado
011	AmbiguityError	Erro de ambiguidade com o identificador '{0}'
012	PrivateMemberNotVisible	Membro '{0}' marcado como 'private' não é visível a partir deste escopo
013	ProtectedMemberNotVisible	Membro '{0}' marcado como 'protected' não é visível a partir deste escopo
0014	InternalMemberNotVisible	Membro '{0}' marcado como 'internal' não é visível a partir deste escopo
015	ProtectedInternalMemberNotVisible	Membro '{0}' marcado como 'protected' e 'internal' não é visível a partir deste escopo
016	CallingNonStaticMemberFromStatic	Não é possível acessar membro de instância '{0}' de um contexto estático
017	CallingStaticMemberFromNonStatic	Não é possível acessar membro estático '{0}' como se fosse membro de instância. Use o nome da classe para acessá-lo
018	DuplicatedMethodDefinition	Método '{0}' duplicado
019	DuplicatedAttributeDefinition	Atributo '{0}' duplicado

Quadro 41 – Erros tratados pela análise semântica

APÊNDICE B – Expressões binárias

Os quadros a seguir apresentam os operandos para as expressões binárias de acordo com seus operadores.

OPERANDO1	OPERANDO2	TIPO RESULTANTE
string	string	bool
char	char	bool
bool	bool	bool
datetime	datetime	bool
timespan	timespan	bool
float	float	bool
double	double	bool
byte	byte	bool
short	short	bool
int	int	bool
long	long	bool

Quadro 42 – Operadores “==” e “!=”

OPERANDO1	OPERANDO2	TIPO RESULTANTE
string	string	bool
char	char	bool
datetime	datetime	bool
timespan	timespan	bool
float	float	bool
double	double	bool
byte	byte	bool
short	short	bool
int	int	bool
long	long	bool

Quadro 43 – Operadores “>”, “<”, “<=” e “>=”

OPERANDO1	OPERANDO2	TIPO RESULTANTE
string	string	string
string	char	string
string	bool	string
string	datetime	string
string	timespan	string
string	float	string
string	double	string
string	byte	string
string	short	string
string	int	string
string	long	string
datetime	timespan	datetime
timespan	timespan	timespan
float	float	float
double	double	double
byte	byte	byte
short	short	short
int	int	int
long	long	long

Quadro 44 – Operador “+”

OPERANDO1	OPERANDO2	TIPO RESULTANTE
datetime	timespan	datetime
datetime	datetime	timespan
timespan	timespan	timespan
float	float	float
double	double	double
byte	byte	byte
short	short	short
int	int	int
long	long	long

Quadro 45 – Operador “-”

OPERANDO1	OPERANDO2	TIPO RESULTANTE
float	float	float
double	double	double
byte	byte	byte
short	short	short
int	int	int
long	long	long

Quadro 46 – Operador “*”

OPERANDO1	OPERANDO2	TIPO RESULTANTE
float	float	double
double	double	double
byte	Byte	double
short	short	double
int	int	double
long	long	double

Quadro 47 – Operador “/”

OPERANDO1	OPERANDO2	TIPO RESULTANTE
byte	byte	byte
short	short	short
int	int	int
long	long	double

Quadro 48 – Operador “%”

OPERANDO1	OPERANDO2	TIPO RESULTANTE
bool	bool	bool
byte	byte	byte
short	short	short
int	int	int
long	long	double

Quadro 49 – Operadores “&”, “|” e “^”

APÊNDICE C – Geração de código MSIL

Os quadros a seguir definem o código MSIL gerado para cada elemento da linguagem proposta.

SINTAXE	MSIL

type ::= (primitive | name)
primitive ::= ("string"
| "char"
| "bool"
| "datetime"
| "timespan"
| "float"
| "double"
| "byte"
| "short"
| "int"
| "long")
	//string
	string
	//char
	char
	//bool
	bool
	//datetime
	valuetype [mscorlib]System.DateTime
	//timespan
	valuetype [mscorlib]System.TimeSpan
	//float
	float32
	//double
	float64
	//byte
	int8
	//short
	int16
	//int
	int32
	//long
	int64

Quadro 50 – Código MSIL para definição de tipos

SINTAXE	MSIL

unitDecl ::= "namespace" name ";"
usingList
(classDecl)+
usingList ::= ("using" name
(".*" ";" | ";"))*
classDecl ::= classModifiers
("class") <ID>
(("extends" name
"{" membersDecl "}"
classModifiers ::= (("public"
| "internal"
| "abstract"
| "sealed")
)*
	.namespace <name>
	{
	.class <modifiers> ansi auto <id>
	extends <class>
	{
	//construtor padrão
	.method public void .ctor()
	cil managed
	{
	ret
	}
	
	//membros
	
	}
	}

Quadro 51 – Código MSIL para definição de classes

SINTAXE	MSIL

method ::= memberModifiers
returnType
<ID>
"(" parameters ")"
block | ";"
 	
field ::= memberModifiers
<type>
<ID> ";"
 	
memberModifiers ::= (("public"
| "protected"
| "internal"
| "private"
| "abstract"
| "virtual"
| "override"
| "static")
)*
parameters ::= type <ID>
("," type <ID>)*
returnType ::= type | "void"
	.method <memberModifiers> hidebysig
	<returnType> <ID> <parameters>
	cil managed
	
	"{"
	//Se é o método de entrada "main",
	//este recebe a diretiva:
	//.entrypoint
	<block>
	"}"
	
	.field <memberModifier> <type> <ID>

Quadro 52 – Código MSIL para definição de membros

SINTAXE	MSIL

ifCmd ::= "if" "(" expression ")"
blockCmd
("else" command)?
	//resolve expressão de condição
	brfalse.s lbl_0001
	//código para condição verdadeira
	br.s lbl_0002
	lbl_0001:
	//código para condição falsa
	lbl_0002:
	//fim do comando if

Quadro 53 – Código MSIL para comando *if*

SINTAXE	MSIL

switchCmd ::= "switch"
"(" expression ")"
"{" ("case" expression
":" blockCmd)*
"default"
":" blockCmd
"}"
	//resolve expressão de comparação
	dup
	//para cada case
	//resolve expressão do case
	clt
	brfalse.s lbl_xxxx //xxxx: proximo label
	comando caso condição seja satisfeita
	//fim para cada case
	//comandos default
	pop

Quadro 54 – Código MSIL para comando *switch*

SINTAXE	MSIL

whileCmd ::= "while" "(" expression ")"
blockCmd
	//resolve expressão de condição
	brfalse lbl_fimWhile
	//blockCmd
	lbl_fimWhile:
	//proximos comandos

Quadro 55 – Código MSIL para comando *while*

SINTAXE	MSIL

doCmd ::= "do" blockCmd
"while" expression ";"
	lbl_inicioDo:
	//blockCmd
	//resolve expressão de condição
	brtrue lbl_inicioDo
	//proximos comandos

Quadro 56 – Código MSIL para comando *do*

SINTAXE	MSIL

forCmd ::= "for" "("
(expression
(<ID> "=" expression ";"
| ";")
)
expression ";" expression
")"
blockCmd
	//resolve iniciazação
	label_inicioFor:
	//resolve condição
	brfalse label_fimFor
	//blockCmd
	//resolve expressão incremento
	br label_inicioFor
	label_fimFor
	//comandos

Quadro 57 – Código MSIL para comando *for*

SINTAXE	MSIL

continueCmd ::= "continue" ";"
breakCmd ::= "break" ";"
returnCmd ::= "return"
(expression)? ";"
	//continue
	br <label do inicio do loop corrente>
	
	//break
	br <label do fim do loop corrente>
	
	//return
	//resolve expressão caso exista
	ret

Quadro 58 – Código MSIL para comandos *continue*, *break* e *return*

SINTAXE	MSIL

varCmd ::= expression <ID>
("=" expression)? ";"
	//no início do bloco
	.locals init(<tipos>)
	
	//caso tenha inicialização
	//resolve expressão de inicialização
	stloc <indice do tipo em "tipos">

Quadro 59 – Código MSIL para comando *var*

SINTAXE	MSIL
methodCall ::= expression ";"	//resolve expressão pop

Quadro 60 – Código MSIL para chamada de método

SINTAXE	MSIL

expression ::= or (assignOp expression)?
orExpr ::= xorExpr (orOp orExpr)?
xorExpr ::= andExpr (xorOp xorExpr)?
andExpr ::= equalExpr (andOp andExpr)?
equalExpr ::= relationalExpr
(equalOp equalExpr)?
relationalExpr ::= addExpr
(relationalOp relationalExpr)?
addExpr ::= multExpr (addOp addExpr)?
multExpr ::= unaryExpr
(multOp multExpr)?
	//resolve expressão da esquerda
	//resolve expressão da direita
	//chama instrução do operador
	
	//ex: 1 < 2
	ldc.i 1
	ldc.i 2
	clt

Quadro 61 – Código MSIL para expressões binárias

SINTAXE	MSIL

unaryExpr ::= ((unaryOp unaryExpr)
| (prefixExpr (suffixExpr)*))
 	
prefixExpr ::= newExpr
| literalExpr
| primitive
 	
newExpr ::= "new" type "("(exprList)?")"
 	
literalExpr ::= (<STRING>
| <CHARACTER>
| <NUMERIC>
| <INTEGER>
| <DATETIME>
| <TIMESTAMP>
| "true"
| "false"
| "null")
	//resolve expressão do operando
	//chama instrução do operador
	
	//new
	newobj instance <construtor>
	
	//literais
	//cada tipo tem um comando diferente
	//para empilhar sua literal
	//ex: string
	ldstr "Exemplo de literal"

Quadro 62 – Código MSIL para expressões unárias

SINTAXE	MSIL

suffixExpr ::= ("." <ID>)
| ("("(exprList)?")"
	//Método
	//de instancia
	callvirt instance <definição do método>
	//estático
	call <definição do método>
	//Obs.: Devem ser empilhados os
	//parâmetros do método antes da
	//instrução call
	
	//Atributo
	ldfld <definição do atributo>

Quadro 63 – Código MSIL para chamada de métodos e atributos