UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

IMPLEMENTAÇÃO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO COM GERAÇÃO DE CÓDIGO MICROSOFT .NET INTERMEDIATE LANGUAGE

GIANCARLO TOMAZELLI

BLUMENAU
2004

2004/1-12

GIANCARLO TOMAZELLI

IMPLEMENTAÇÃO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO COM GERAÇÃO DE CÓDIGO MICROSOFT .NET INTERMEDIATE LANGUAGE

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 Ciência da Computação — Bacharelado.

Prof. José Roque Voltolini da Silva - Orientador

BLUMENAU
2004

2004/1-12

IMPLEMENTAÇÃO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO COM GERAÇÃO DE CÓDIGO MICROSOFT .NET INTERMEDIATE LANGUAGE

Por

GIANCARLO TOMAZELLI

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

Presidente:	Prof. José Roque Voltolini da Silva – Orientador, FURB
Membro:	Prof. Alexander Roberto Valdameri, FURB
Membro:	Prof. Jomi Fred Hübner, FURB

Blumenau, Junho de 2004

“Nas grandes batalhas da vida, o primeiro passo para a vitória é o desejo de vencer!”

Mahatma Gandhi

AGRADECIMENTOS

Em primeiro lugar, agradeço os meus pais, Rui e Sueli. Vocês me forneceram uma sólida educação e me apóiam em todos os momentos, dando muito amor e carinho.

Agradeço também os meus irmãos Giu, Luize e minha namorada Vera, que me acompanham todos os dias, dando muitas alegrias e motivos para continuar a viver.

Meus especiais agradecimentos para Joyce que, mesmo longe, apoiou-me na definição e no esclarecimento de dúvidas do trabalho. Agradecimentos especiais também para o meu patrão, Werner, que me permitiu o acesso à tecnologia.

Obrigado também ao meu orientador Roque que, apesar de todos os afazeres do departamento, aceitou o desafio de orientar-me neste trabalho contribuindo significamente para o desenvolvimento do mesmo.

Para finalizar, agradeço em geral a todos os meus colegas e professores, por todos os momentos, conhecimentos e experiências repassados.

RESUMO

Este trabalho descreve a criação de uma linguagem de programação para ser executada na máquina virtual Common Language Runtime (CLR) da plataforma Microsoft .NET. Uma linguagem de programação simplificada e em português é definida, herdando algumas características estruturais da linguagem C. A especificação da linguagem é mostrada na notação Backus-Naur Form (BNF). O ambiente de programação é especificado usando orientação a objetos. O ambiente possui um editor de textos, opções para compilar, montar e executar programas. O compilador é implementado usando a ferramenta *ProGrammar*. O resultado do processo de compilação é um arquivo texto ".IL", o qual é lido pelo montador ILAsm da Microsoft gerando um arquivo ".EXE" que pode ser executado pelo CLR.

Palavras chaves: Compilador; .NET Intermediate Language; Ambiente de Programação.

ABSTRACT

This work describes the creation of a programming language to be executed on the Common Language Runtime (CLR) virtual machine of Microsoft .NET platform. A simplified programming language in portuguese is defined, inheriting some structural characteristics of the C language. The language specification is written in Backus-Naur Form (BNF) notation. The programming environment is specified using object oriented design. This environment has a text editor, options to compile, assemble and execute programs. The compiler is implemented using the ProGrammar tool. The compilation process results in a ".IL" text file, which is read by the Microsoft ILAsm assembler generating a ".EXE" file, which can be executed by the CLR.

Key words: Compiler; .NET Intermediate Language; Programming Environment.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

BNF – *Backus Naur Form*

CIL – *Common Intermediate Language*

CLI – *Common Language Infrastructure*

CLR – *Common Language Runtime*

CLS – *Common Language Specification*

CTS – *Common Type System*

FCL – *Framework Class Library*

GDL – *Grammar Definition Language*

IL – *Intermediate Language*

JVM – *Java Virtual Machine*

JIT – *Just-in-Time*

MFC – Microsoft *Foundation Classes*

MSIL – Microsoft *Intermediate Language*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Desde os primórdios da era da computação, os operadores ou usuários necessitam informar os dados para os mesmos serem processados. Entretanto, uma máquina e dados para processamento nada fazem sem a existência de um programa. Um programa é um conjunto de instruções que indicam à máquina o que fazer com os dados informados (dados de entrada).

Inicialmente, o conceito de programa da maneira conhecida atualmente não existia. Os computadores eram limitados e conseqüentemente, as aplicações também. Programar um computador consistia em escrever as operações para manipulação dos dados usando notação binária, comunicando-se diretamente com a máquina em uma seqüência de zeros e uns (PRICE; TOSCANI, 2001, p. 1).

Com o passar do tempo e a evolução da tecnologia, os computadores foram ficando cada vez mais complexos, bem como programá-los. Era muito difícil a memorização das operações que uma determinada máquina (arquitetura) poderia realizar e assim, fazia-se necessária uma maneira mais intuitiva de escrever programas. É neste ponto que entram os compiladores, tema principal deste trabalho de conclusão de curso.

Em resumo, um compilador nada mais é do que um tradutor que, a partir de uma linguagem de origem produz uma linguagem de destino, preservando seu significado original. Estas linguagens de origem são denominadas linguagens de programação, sendo que a compilação de um programa escrito numa linguagem de origem pode gerar um programa em outra linguagem de programação. Como exemplo, cita-se a linguagem C++, cuja compilação de um programa escrito nesta linguagem gera um programa em linguagem *assembler*. Atualmente, existem inúmeras linguagens de programação, bem como diversas arquiteturas que executam os programas escritos nas mesmas. Portanto, para cada combinação linguagem X arquitetura existem um ou mais compiladores para traduzir os programas escritos.

O objetivo deste trabalho é o desenvolvimento de um compilador para uma linguagem de programação simplificada e em português, gerando código .NET *Intermediate Language* (IL) para ser executado na plataforma Microsoft .NET. A plataforma Microsoft .NET representa uma importante mudança na plataforma de desenvolvimento da Microsoft em quase dez anos. Esta plataforma tem como base uma nova infra-estrutura de *software* (.NET

Framework) para execução de aplicativos, um novo ambiente de desenvolvimento e linguagens de programação para suportar essa infra-estrutura.

1.1 MOTIVAÇÃO

O principal motivo para o desenvolvimento deste trabalho constitui o estudo de ferramentas para auxiliar o desenvolvimento de compiladores, o desenvolvimento de uma nova linguagem de programação e a pesquisa da arquitetura Microsoft .NET.

As técnicas envolvidas na construção de compiladores são amplamente utilizadas no campo da ciência da computação. Entre essas técnicas, podem-se citar os algoritmos usados e as estruturas de dados que os compiladores possuem. Exemplos de algoritmos usados em compiladores são os relacionados com a teoria dos grafos. Entre as estruturas de dados, destacam-se as tabelas pré-calculadas, mecanismos de pilha e alocação dinâmica de memória.

Os conhecimentos e as ferramentas empregadas durante o processo de construção do compilador podem ser utilizadas para desenvolver um tradutor para uma nova linguagem de programação bem como na resolução de inúmeros problemas na área da ciência da computação, dentre os quais pode-se citar: a leitura de dados estruturados, introdução de formatos e problemas de conversão de arquivos.

1.2 OBJETIVOS

O objetivo deste trabalho é construir um compilador para uma linguagem de programação simplificada e em português e construir um ambiente de desenvolvimento para a nova linguagem.

Os objetivos específicos do trabalho são:

a) executar os programas gerados pelo compilador na plataforma Microsoft .NET;
b) demonstrar a estrutura de uma aplicação .NET.

1.3 ORGANIZAÇÃO DO TEXTO

O trabalho está organizado em quatro capítulos. O primeiro capítulo apresentou a introdução e os objetivos do trabalho.

No segundo capítulo, inicialmente é fornecida uma breve explanação sobre compiladores, relatando as principais fases do processo de compilação. A ferramenta usada na implementação do compilador é descrita. É apresentada a plataforma Microsoft .NET, que constitui a base de execução dos aplicativos programados na linguagem implementada neste trabalho.

O terceiro capítulo apresenta a especificação da sintaxe e semântica da nova linguagem de programação, além da implementação do compilador e do ambiente de programação para a linguagem proposta.

No quarto capítulo são apresentadas as conclusões do trabalho, suas limitações e possíveis extensões para o mesmo.

2 FUNDAMENTAÇÃO TEÓRICA

Inicialmente neste capítulo são abordados de forma resumida os conceitos de compiladores. A seguir, é descrita a ferramenta *ProGrammar*, a qual é usada para auxiliar a construção do compilador para a linguagem proposta. Também, a estrutura da plataforma Microsoft .NET é descrita, juntamente com o MSIL.

2.1 COMPILADORES

As pessoas de um modo geral utilizam linguagens para comunicarem-se umas com as outras. Isto somente é possível com a existência de um idioma, isto é, um conjunto de regras e símbolos que visam disciplinar a comunicação entre um grupo de indivíduos.

Segundo Price e Toscani (2001, p. 1), na programação de computadores utilizam-se linguagens de programação como meio de comunicação com o computador escolhido. Estas linguagens são usadas para descrever a resolução de um determinado problema. A linguagem de programação faz a ligação entre o pensamento humano (na maioria das vezes de natureza não estruturada) com a precisão requerida do processamento pelo computador.

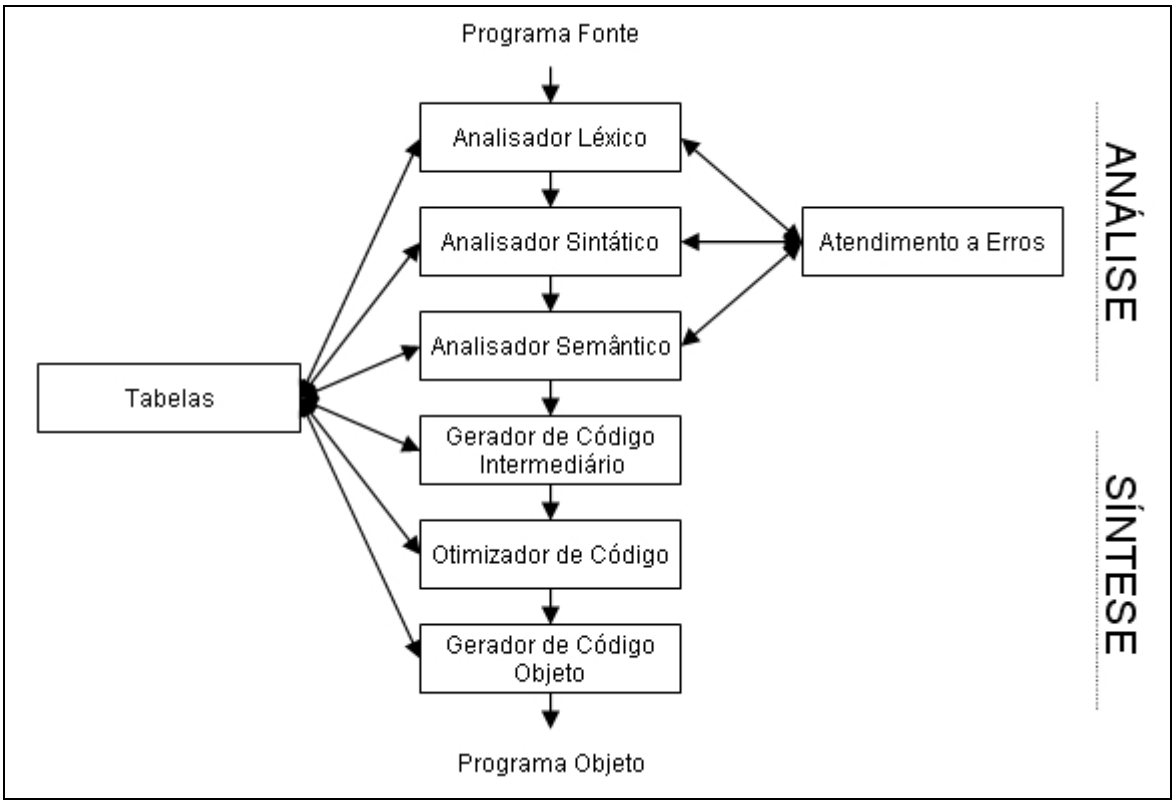
Atualmente, as linguagens de programação mais utilizadas são de alto nível, entendidas pelos seres humanos e consideradas mais próximas às linguagens naturais. Entretanto, essas linguagens são totalmente irreconhecíveis aos computadores que somente entendem sua própria linguagem: a linguagem de baixo nível ou linguagem de máquina, tipicamente composta por uma seqüência de zeros e uns. Portanto, os programas escritos em linguagens de alto nível precisam ser traduzidos para serem entendidos pelos computadores. Esta tradução é realizada por sistemas especializados denominados compiladores (PRICE; TOSCANI, 2001, p. 1).

Grune et al. (2001, p. 1) define compilador como “um programa que aceita como entrada um texto de programa em uma certa linguagem e produz como saída um texto de programa em outra linguagem, enquanto preserva o significado deste texto”. Desta maneira, é possível mapear programas escritos em linguagens de alto nível (programa fonte) para programas equivalentes em linguagens de baixo nível (simbólica ou de máquina) viabilizando a sua execução (programa objeto) em uma determinada arquitetura (*hardware*).

2.1.1 FASES DE COMPILAÇÃO

Os compiladores são programas de alta complexidade. Entretanto, devido às inúmeras implementações e ao desenvolvimento de teorias relacionadas aos mesmos, existe uma estrutura básica a ser seguida no projeto de um compilador.

Os compiladores, de um modo geral, compõem-se de funções padronizadas. A saída de uma função constitui a entrada de outra, compreendendo a análise do programa fonte e a posterior síntese para a derivação do código objeto (PRICE; TOSCANI, 2001, p. 7), como pode ser visto na Figura 1.



Fonte: Price e Toscani (2001, p. 8)

Figura 1 - Estrutura de um compilador

Desta maneira, o processo de compilação consiste basicamente das seguintes fases: análise léxica, análise sintática, análise semântica e geração de código.

2.1.1.1 ANÁLISE LÉXICA

A análise léxica é responsável pela identificação das seqüências de caracteres que formam os *tokens*. Os *tokens* são as classes de símbolos da linguagem, como as palavras reservadas, delimitadores, operadores, identificadores, etc. A identificação dos *tokens* ocorre

através da leitura caractere a caractere do arquivo fonte, verificando se os mesmos são símbolos válidos e estão de acordo com o padrão de formação especificado, desconsiderando os comentários e caracteres em branco. Padrões de formação são definidos através de expressões regulares que, segundo Grune et al. (2001, p. 54), "é uma fórmula que descreve um conjunto de *strings* possivelmente infinito". Como exemplo, a expressão regular ab* gera um conjunto infinito de *tokens* ({a, ab, abb, abbb, ...}). Caso a seqüência identificada não contenha símbolos válidos ou não esteja de acordo com as especificações feitas nas expressões regulares, uma mensagem de erro é gerada.

Além de identificar os *tokens*, o analisador léxico inicia a construção da tabela de símbolos. Aho, Sethi e Ullman (1995, p. 5) definem a tabela de símbolos como uma estrutura de dados que contém um registro para cada identificador. Os campos do registro contêm os atributos do identificador, como o tipo, o escopo, número de parâmetros (caso o identificador seja o nome de um procedimento), etc. A tabela de símbolos permite localizar rapidamente um identificador, juntamente com os campos contendo seus atributos. O resultado da análise léxica consiste de uma cadeia de *tokens* a ser processada pelo analisador sintático.

2.1.1.2 ANÁLISE SINTÁTICA

A função da análise sintática é validar a estrutura gramatical do programa, verificando se o mesmo está em conformidade com as regras gramaticais da linguagem. As regras gramaticais que definem as construções da linguagem geralmente são descritas através de produções (regras que produzem, geram) e seus elementos incluem símbolos terminais (aqueles que fazem parte do código fonte) e símbolos não-terminais (aqueles que geram outras regras) (PRICE; TOSCANI, 2001, p. 10). Geralmente, as produções são apresentadas utilizando-se a notação BNF, conforme exemplificadas no Quadro 1.

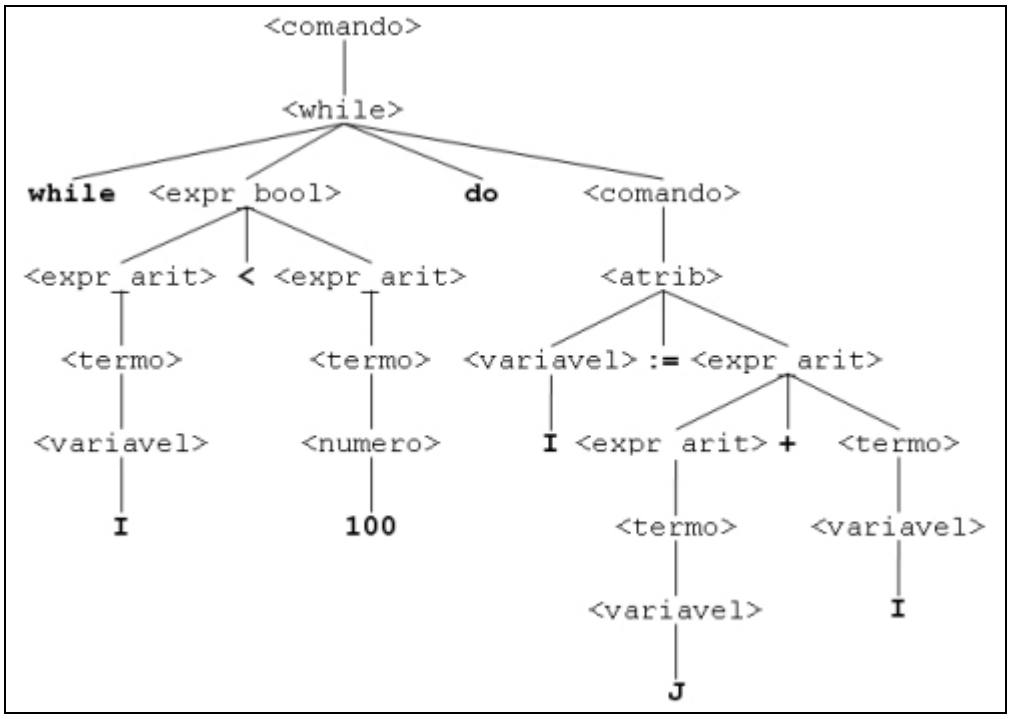
```
<comando>   -> <while> | <atrib>
<while>     -> while <expr_bool> do <comando>
<atrib>     -> <variavel> := <expr_arit>
<expr_bool> -> <expr_arit> < <expr_arit>
<expr_arit> -> <expr_arit> + <termo> | <termo>
<termo>     -> <numero> | <variável>
<variável>  -> I | J
<numero>    -> 100
```

Fonte: Price e Toscani (2001, p. 10)

Quadro 1 - Exemplo de produções BNF

Através da varredura ou *parsing* da cadeia de *tokens* resultante do processo anterior (análise léxica) e a aplicação das regras gramaticais da linguagem, são identificadas as seqüências de símbolos que formam as estruturas sintáticas, como as expressões e os comandos. Estas estruturas são geralmente representadas por uma árvore de derivação ou árvore gramatical.

Aplicando-se a BNF descrita no Quadro 1 para o fragmento de código `while I < 100 do I := I + J` é gerada a árvore de derivação exibida na Figura 2.



Fonte: Price e Toscani (2001, p. 11)

Figura 2 - Árvore de derivação

Segundo Price e Toscani (2001, p. 29), a árvore de derivação pode ser construída explicitamente (representada através de uma estrutura de dados) ou implicitamente nas chamadas das rotinas que aplicam as regras de produção da gramática durante o reconhecimento. Os analisadores sintáticos devem ser projetados a fim de procederem à análise de todo o programa fonte, mesmo que existam erros no mesmo. A análise sintática possui duas estratégias básicas:

a) estratégia *top-down* ou descendente;
b) estratégia *bottom-up* ou redutiva.

Os analisadores sintáticos baseados na estratégia *top-down* constroem a árvore de derivação a partir do símbolo inicial da gramática e crescem até atingirem as suas folhas (*tokens* do texto fonte) e em cada passo, o lado esquerdo da produção é substituído por um lado direito (expansão). Na estratégia *bottom-up*, a análise é realizada no sentido inverso, a partir das suas folhas até o símbolo inicial da gramática, sendo que em cada passo um lado direito de produção é substituído por um símbolo não-terminal (redução).

2.1.1.3 ANÁLISE SEMÂNTICA

A principal função do analisador semântico é determinar se as estruturas sintáticas montadas na fase anterior (análise sintática) possuem sentido. A análise ocorre através de ações semânticas, onde estas ações incluem, por exemplo, a verificação de tipos, verificação da declaração de variáveis, compatibilidade entre operando e operadores, etc.

As ações semânticas são ações associadas às regras de produção da gramática da linguagem. Quando uma ação é processada, o código correspondente é executado. Esta execução pode gerar ou interpretar código, armazenar informações na tabela de símbolos, emitir mensagens de erro, etc.

2.1.1.4 GERAÇÃO DE CÓDIGO

Nesta fase tem-se início a montagem do código objeto. Utilizando a representação produzida pelos processos anteriores (análises sintática e semântica), uma seqüência de código é gerada.

A geração de código pode ser precedida pela geração de um código intermediário. Segundo Price e Toscani (2001, p. 115) a geração de código intermediário apresenta algumas vantagens, como:

a) possibilita a otimização do código intermediário, obtendo-se um código objeto final mais eficiente;
b) simplifica a implementação do compilador, resolvendo gradativamente os problemas da passagem de código fonte (alto nível) para código objeto (baixo nível);
c) possibilita a tradução do código intermediário para diversas máquinas.

A desvantagem na geração de código intermediário é a requisição de um passo adicional ao processo. Traduzir diretamente o código fonte para o código objeto representa uma compilação mais rápida.

Esta seqüência de código intermediário pode sofrer mudanças durante a sua geração. Como o próprio nome sugere, este código pode ser otimizado, obtendo-se um código objeto final mais eficiente.

Esta fase também resolve os problemas da passagem de código fonte para o código objeto. Geralmente, as linguagens alvos de um compilador são linguagens de baixo nível, onde uma linha do código fonte pode ser transformada em inúmeras instruções na linguagem de destino. Esta passagem pode ser efetuada de maneira gradual, gerando um código objeto mais limpo e livre de erros.

2.1.2 MÁQUINAS VIRTUAIS COMO ALVO DE COMPILAÇÃO

Gough (2002, p. 2) afirma que uma tendência na tecnologia de *software* a fim de se obter portabilidade é compilar programas para uma forma intermediária, baseada na definição de uma máquina virtual.

Usualmente, uma máquina virtual é um conjunto de instruções que operam em uma máquina de pilhas abstrata. As instruções da máquina retiram e colocam operandos que residem em uma pilha de avaliação. O conjunto de instruções consiste basicamente em 3 tipos de operações (GOUGH, 2002, p. 3):

a) instruções que colocam operandos na pilha de avaliação;
b) instruções para trabalhar com os operandos que estão no topo da pilha;
c) instruções que retiram operandos da pilha e armazenam os mesmos em memória.

Os primeiros estudos referentes à geração de código objeto para máquinas virtuais datam da década de 70. Recentemente, o assunto voltou à tona com o advento da linguagem de programação *Java*, cujos programas são compilados para *bytecodes* compreendidos pela *Java Virtual Machine* (JVM).

Em julho de 2000 a Microsoft lançou a plataforma .NET. Semelhante a JVM, a plataforma Microsoft .NET também é baseada em uma máquina virtual, o CLR. O CLR é

responsável pela execução dos programas compilados nas diversas linguagens disponíveis para esta plataforma.

A JVM e o CLR possuem as seguintes características em comum:

a) são baseadas em pilhas (*stack-based*);
b) oferecem maior segurança em operações de acesso a memória;
c) possuem mecanismos para gerenciamento de memória (*garbage collector*);
d) instruções primitivas que dão suporte à orientação a objetos (criação de objetos, acesso a membros, etc).

2.2 FERRAMENTAS PARA AUXILIAR A CONSTRUÇÃO DE COMPILADORES

Como em qualquer outra área da computação, existem ferramentas para auxiliar os desenvolvedores de compiladores. Algumas destas ferramentas são conhecidas como compilador de compiladores (*compiler-compilers*) e geradores de compiladores (*compiler generators*), além de outras.

Segundo Price e Toscani (2001, p. 14), estas ferramentas classificam-se em três grupos:

a) geradores de analisadores léxicos: o objetivo destas ferramentas é gerar reconhecedores de símbolos léxicos (palavras reservadas, identificadores, operadores, etc.) utilizando expressões regulares. Um exemplo clássico de gerador de analisador léxico é o Lex (GERMAN NATIONAL RESEARCH CENTER FOR INFORMATION TECHNOLOGY, 2003);
b) geradores de analisadores sintáticos: constituem as principais ferramentas na construção de um compilador e sua função é produzir reconhecedores sintáticos a partir de gramáticas livres de contexto (GLCs). Antigamente, implementar um analisador sintático consistia em uma árdua tarefa e exigia tempo, além do que mudanças na sintaxe da linguagem obrigavam muitas vezes a re-escrever o analisador. Um exemplo bastante conhecido de geradores desse tipo é o Yacc (GERMAN NATIONAL RESEARCH CENTER FOR INFORMATION TECHNOLOGY, 2003);
c) geradores de geradores de código: estas ferramentas compõem regras que definem a tradução das operações da linguagem intermediária para a linguagem objeto final. As regras devem possuir detalhes suficientes para permitir a manipulação correta

com diferentes métodos de acesso aos dados. Geralmente as instruções intermediárias são mapeadas em instruções de máquina.

2.2.1 FERRAMENTA *PROGRAMMAR*

A ferramenta *ProGrammar* é um ambiente de desenvolvimento para a construção de analisadores sintáticos (NORKEN TECHNOLOGIES, 2004). Os analisadores sintáticos são amplamente usados na construção de compiladores, mas também podem ser relativamente úteis na implementação de programas para leitura de dados estruturados, validação de formatos, conversores de arquivos, etc.

A ferramenta é composta basicamente de:

a) *Interactive Development Environment* (IDE): ambiente visual para implementar, testar e depurar analisadores sintáticos;
b) *Grammar Definition Language* (GDL): uma linguagem de alto nível utilizada para definir gramáticas;
c) *Parse Engine*: componente que faz a análise léxica e sintática de um texto conforme a gramática especificada. Este componente está disponível nos seguintes formatos: biblioteca dinâmica (DLL), biblioteca estática (LIB) e controle ActiveX;
d) *Application ProGrammar Interface*: uma interface que provém métodos para acesso ao *parse engine*.

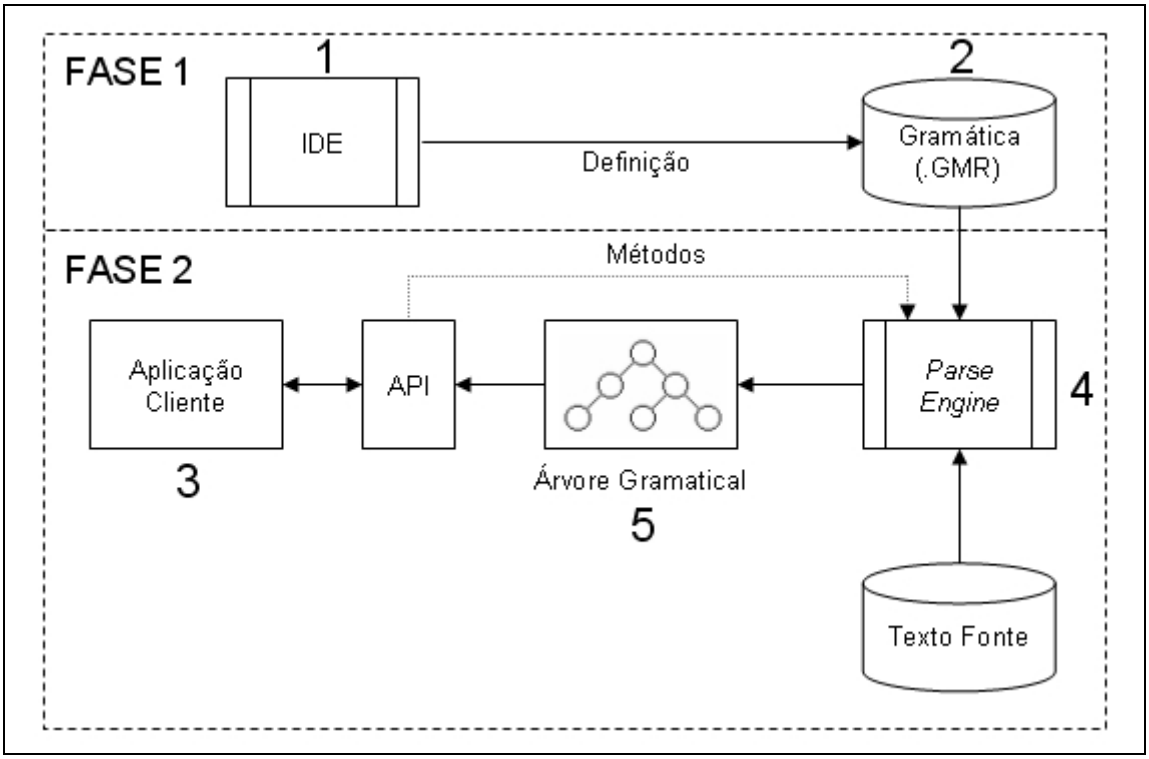
A GDL serve para descrever a estrutura de uma gramática. Uma gramática é um conjunto de regras que dizem ao analisador como proceder a análise de um determinado texto fonte. A GDL possui a seguintes características:

a) orientada a objetos: uma gramática pode ser herdada de uma ou mais gramáticas. As gramáticas herdam todos as regras e símbolos de seus antecessores, facilitando o processo de criação de gramáticas reutilizáveis;
b) alto-nível: a GDL simplifica o processo de desenvolvimento de gramáticas, através das seguintes facilidades: *tokens* sensíveis ao caso, descarte de comentários e espaços em branco, *tokens* pré-definidos;
c) simples: é uma linguagem projetada para ser consistente e simples, não fazendo uma distinção entre as tarefas de análise léxica e sintática;
d) independente: a GDL não faz nenhuma suposição sobre quais linguagens de programação são usadas para escrever os programas que utilizarão o analisador

sintático. Os analisadores são desenvolvidos independentemente das aplicações que os usam e, portanto, a mesma gramática pode ser empregada em qualquer aplicação, seja ela programada em C++, Visual Basic ou Delphi.

Diferente de outros compiladores de compiladores, o *ProGrammar* não gera código fonte. O desenvolvedor pode construir o analisador independente da linguagem de programação utilizada. A aplicação cliente acessa a árvore gramatical gerada através da *Application ProGrammar Interface* e executa as ações semânticas.

Dessa maneira, o *ProGrammar* divide a tarefa da construção do analisador sintático em duas fases distintas (Figura 3). Na primeira fase (Figura 3, fase 1), a gramática é definida através do IDE (passo 1). Em seguida, o IDE gera um arquivo binário da gramática contendo instruções para a análise do texto fonte (passo 2). A segunda fase (Figura 3, fase 2) envolve a chamada para o *parse engine* (passo 4) a partir da aplicação cliente através da *Application ProGrammar Interface* (passo 3), realizando a análise sintática do texto fonte e processando a árvore gramatical resultante (passo 5).



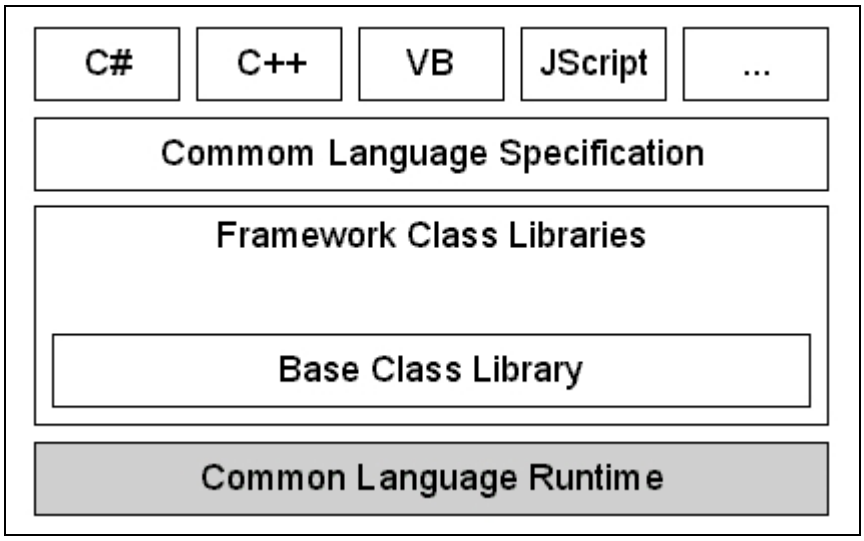
Fonte: Norken Technologies (2004).

Figura 3 - Passos para utilizar a ferramenta *ProGrammar*

2.3 PLATAFORMA MICROSOFT .NET

A plataforma Microsoft .NET consiste de uma plataforma para integração de aplicativos e serviços na *internet*. Ela é semelhante ao *Java 2 Enterprise Edition*, da Sun Microsystems (SUN MICROSYSTEMS, 2004). Devido à natureza distribuída dos aplicativos da *internet*, os mesmos necessitam de comunicação uns com os outros, operando em diversas plataformas e sistemas operacionais. Projetar um sistema para uma única plataforma já consiste em uma tarefa difícil, projetar para inúmeras configurações é mais complexo ainda. Para facilitar esse processo, a plataforma Microsoft .NET é composta das seguintes partes (Figura 4):

a) *Common Language Runtime* (CLR);
b) .NET *Framework Class Library* (FCL).



Fonte: Cherry e Demichillie (2002, p. 3)
Figura 4 - A plataforma Microsoft .NET

A primeira parte consiste do mecanismo para execução de aplicativos projetados para esta plataforma, de maneira semelhante a *Java Virtual Machine* (JVM) (SUN MICROSYSTEMS, 2004). A segunda parte é um conjunto integrado de bibliotecas, que oferecem aos desenvolvedores os componentes de *software* necessários para escrever programas na plataforma Microsoft .NET (CHERRY; DEMICHILLIE, 2002, p. 5). É análogo ao *Java Foundation Classes* (SUN MICROSYSTEMS, 2004).

Todos os aplicativos desenvolvidos são compilados para uma espécie de metacódigo, chamado *Microsoft Intermediate Language* (MSIL). Portanto, inúmeras ferramentas de desenvolvimento, como o C#Builder da Borland (BORLAND SOFTWARE

CORPORATION, 2004), possuem mecanismos para gerar aplicativos compatíveis com a plataforma Microsoft .NET. Com o código compilado para o MSIL, o aplicativo está pronto para ser executado pelo CLR.

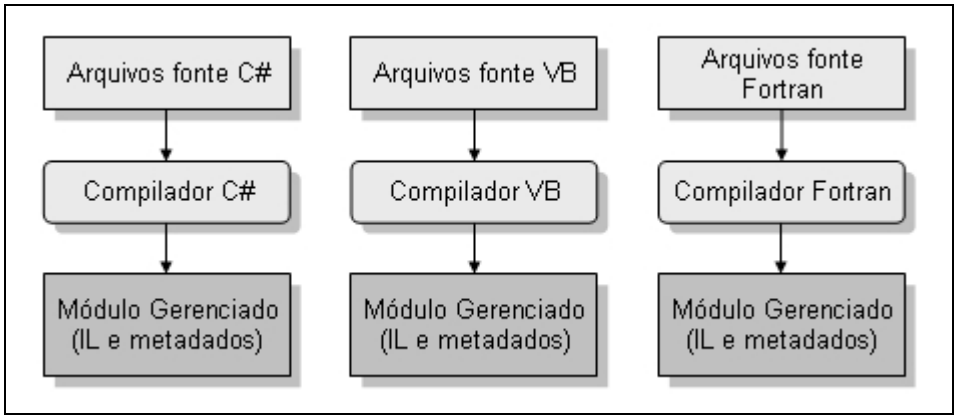
2.3.1 *COMMON LANGUAGE RUNTIME* (CLR)

Cherry e Demichillie (2002, p. 5) definem o CLR como um conjunto de rotinas que carregam aplicativos criados para a plataforma Microsoft .NET. Durante a carga de um aplicativo, o CLR verifica se todas as referências (classes, métodos) utilizadas estão resolvidas, verifica as permissões de segurança e o executa, fazendo a limpeza de memória durante e após o término do processo.

Os objetivos principais do CLR são:

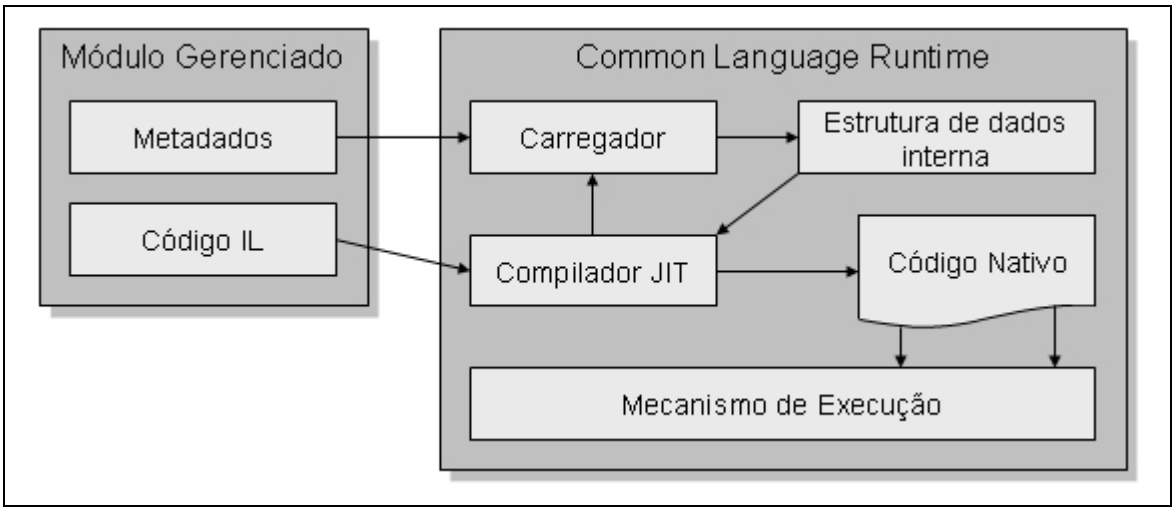
a) melhorar a confiabilidade e segurança dos aplicativos;
b) reduzir o volume de código de baixo nível nos fontes dos aplicativos.

O CLR permite ao desenvolvedor utilizar várias linguagens de programação, desde que os compiladores para estas linguagens emitam código MSIL. Independente da linguagem e compilador utilizados, o resultado da compilação é um módulo gerenciado (*managed module*) exibido na Figura 5. Um módulo gerenciado é um arquivo executável (composto pelos elementos descritos na Tabela 1), que requer o CLR para ser executado (Figura 6) (RICHTER, 2002, p. 4).



Fonte: Richter (2002, p. 5)

Figura 5 – Exemplos de compiladores que geram módulos gerenciados



Fonte: Serge (2002, p. 7)

Figura 6 - Execução de um aplicativo .NET

Tabela 1 - Partes de um módulo gerenciado

Parte	**Descrição**
PE *header*	É o cabeçalho padrão dos arquivos executáveis da plataforma Windows. O cabeçalho indica o tipo do aplicativo: gráfico, console, ou biblioteca dinâmica.
CLR *header*	Contém informações que são interpretadas pelo CLR (e alguns utilitários). Este cabeçalho inclui a versão do CLR requerida, o método de entrada do módulo (*entry point*) e o tamanho do metadados do módulo, além de outras informações.
Metadados	Todo módulo gerenciado possui dois tipos principais de tabelas de metadados: tabelas que descrevem as classes e membros definidos no código fonte e tabelas que descrevem as classes e membros referenciados pelo código.
Código *Intermediate Language* (IL)	É o código produzido por um compilador compatível com o CLR. O CLR compila o código IL em instruções nativas da CPU no momento da execução.

Fonte: Richter (2002, p. 5)

Entretanto, o CLR não trabalha diretamente com o módulo gerenciado. O CLR trabalha com *assemblies*. Um *assembly* é um conjunto de um ou mais módulos gerenciados. A principal parte que compõem o *assembly* é o código escrito em MSIL, que é traduzido pelo CLR para código nativo executável. É este controle da tradução que permite ao CLR

gerenciar a execução do aplicativo e tratar exceções como, por exemplo, o acesso a locais inválidos de memória.

Além do código MSIL, um *assembly* contém metacódigo que descreve os tipos e componentes exigidos pelo código MSIL para ser executado corretamente. Finalmente, o *assembly* inclui um documento que lista todos os arquivos e componentes utilizados. Este documento é conhecido como manifesto. O manifesto indica onde o CLR deverá procurar outros *assemblies* contendo os componentes necessários para a execução.

Para executar um aplicativo, o CLR utiliza o manifesto do *assembly* para encontrar a versão correta dos *assemblies* de que o aplicativo necessita (CHERRY; DEMICHILLIE, 2002, p. 5). O CLR examina todo o código MSIL do aplicativo, verificando se o código executa apenas as operações apropriadas em certos tipos de dados. Ainda, determina de onde o código foi baixado, quais componentes ele precisa para executar, qual usuário está tentando executá-lo e verifica também se o *assembly* possui uma assinatura digital válida. Estas verificações constituem a base para a segurança do .NET *Framework*. Na seqüência, o CLR faz a tradução do código MSIL em código nativo para ser executado pelo processador. Com um recurso chamado compilação *Just-in-Time* (JIT), o CLR pode "adiar" a tradução do código MSIL até que este seja realmente necessário, evitando a tradução de códigos utilizados esporadicamente. Finalmente, o CLR faz o monitoramento do código traduzido durante a execução e realiza o processo de gerenciamento de memória e coleta de lixo.

2.3.2 .NET *FRAMEWORK CLASS LIBRARY* (FCL)

A plataforma Microsoft .NET possui um conjunto de *assemblies* chamado *Framework Class Library* (RICHTER, 2002, p. 21). Este conjunto de *assemblies* forma uma biblioteca que contém milhares de classes, onde cada classe implementa alguma funcionalidade para os desenvolvedores, permitindo a construção dos seguintes tipos de aplicativos:

a) XML *Web Services*: provém métodos que podem ser acessados através da *internet*;
b) *Web Forms*: usado para construir aplicações baseadas em HTML. Geralmente, as aplicações baseadas em HTML fazem consultas à banco de dados e chamadas para *Web Services*, combinando, filtrando informações e apresentando as mesmas em um navegador Web;
c) *Windows Forms*: usado para construir aplicações com uma interface gráfica. Disponibiliza ao desenvolvedor a maioria dos controles da interface Windows

como menus, caixas de edição, caixas de seleção, *radio buttons*, etc. É semelhante aos *Web Forms*, permitindo também consultas à banco de dados e chamadas para *Web Services*;

d) *Windows Console Applications*: usado para aplicações com uma interface simples. Geralmente é utilizada para construir utilitários como compiladores e conversores de arquivos;

e) *Windows Services*: permite a criação de aplicações do tipo serviço, controladas pelo *Windows Service Control Manager* (SCM);

f) *Component Library*: permite ao desenvolvedor projetar classes que podem ser incorporadas facilmente em qualquer tipo de aplicações mencionadas anteriormente.

Devido ao fato da FCL conter milhares de classes, as mesmas são agrupadas segundo a sua funcionalidade em *namespaces* (RICHTER, 2002, p. 22). Por exemplo, o *namespace* `System` contém a classe base `Object`, na qual todos os outros objetos são derivados. O *namespace* `System` também contém classes para tipos de dados inteiros, caracteres, *strings*, tratadores de exceções, rotinas para entrada/saída em aplicações console além de várias classes para conversão de tipos de dados, formatação de dados, geração de números randômicos e funções matemáticas. A Tabela 2 apresenta alguns *namespaces* da FCL.

Tabela 2- Alguns *namespaces* da FCL

Namespace	**Descrição**
`System`	Todas as classes bases usadas por qualquer aplicativo.
`System.Collections`	Classes para manipulação de coleções de objetos tais como pilhas, filas e tabelas *hash*.
`System.IO`	Classes para Entrada/Saída em arquivos, navegação em diretórios.
`System.Threading`	Classes para operações assíncronas e sincronização para acesso à recursos.

Fonte: Richter (2002, p. 23)

2.3.3 *COMMON TYPE SYSTEM* (CTS)

A plataforma Microsoft .NET é composta por classes, que provém funcionalidades para aplicações e componentes. São estas classes que permitem a comunicação entre as diversas linguagens de programação da plataforma Microsoft .NET. Como todo o CLR é baseado em classes, uma especificação formal foi criada pela Microsoft a fim de descrever a

definição e o comportamento destas classes. Essa especificação formal é o *Common Type System* (CTS) (RICHTER, 2002, p. 24).

Segundo a especificação CTS, uma classe pode conter zero ou mais membros. Os membros podem ser:

a) *field*: é um atributo da classe, identificado por um tipo e nome;
b) *method*: é uma função que realiza uma operação no objeto, geralmente mudando seus atributos. Os métodos possuem um nome, uma assinatura e modificadores. A assinatura especifica a convenção de chamada, o número de parâmetros (e sua seqüência), tipos dos parâmetros e tipo de retorno do método;
c) *property*: é semelhante a um *field* e um *method*. As propriedades permitem ao desenvolvedor validar os parâmetros de entrada e os atributos da classe antes de acessá-lo. Também permitem uma sintaxe mais simples e finalmente, permitem criar atributos somente de leitura ou somente de escrita.
d) *event*: permite um mecanismo de notificação entre dois ou mais objetos. Por exemplo, um objeto botão pode oferecer um evento que notifica outros objetos quando o mesmo é pressionado.

O CTS também especifica as regras para visibilidade e acesso aos membros de uma classe. Por exemplo, uma classe marcada como *public* é visível para qualquer *assembly*. As seguintes opções são válidas para especificar a visibilidade de um método ou atributo:

a) *private*: o método pode ser chamado somente em classes do mesmo tipo;
b) *family*: o método pode ser chamado por classes derivadas, independentemente se elas estão declaradas no mesmo *assembly*. Também é referenciado como *protected*;
c) *family* e *assembly*: o método pode ser chamado por classes derivadas, mas somente se elas forem declaradas no mesmo *assembly*;
d) *assembly*: o método pode ser chamado por qualquer classe no mesmo *assembly*. Também é referenciado como *internal*;
e) *family* ou *assembly*: o método pode ser chamado por classes derivadas em qualquer *assembly*. Pode também ser chamado por quaisquer classes no mesmo *assembly*.
f) *public*: o método pode ser chamado por qualquer classe em qualquer *assembly*.

Adicionalmente, o CTS define regras para herança, funções virtuais, tempo de vida dos objetos e demais regras para contemplar a maioria da semântica encontrada nas linguagens de programação modernas.

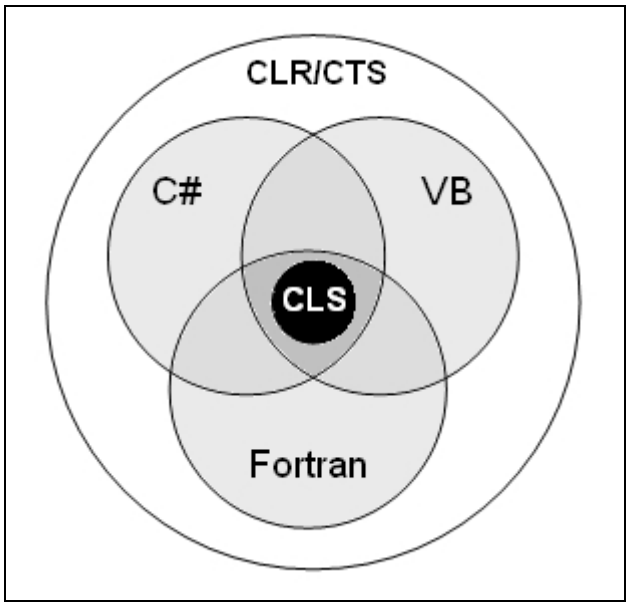
2.3.4 *COMMON LANGUAGE SPECIFICATION* (CLS)

O CLR integra todas as linguagens de programação disponíveis para a plataforma Microsoft .NET, permitindo que as classes escritas em uma linguagem possam ser utilizadas em outras linguagens completamente diferentes (RICHTER, 2002, p. 27). Esta integração somente é possível devido ao fato do CLR ser um conjunto padrão de classes, metadados (tipos de informação autodescritíveis) e um ambiente de execução comum.

Richter (2002, p. 27) afirma que as linguagens de programação diferem muito umas das outras. Por exemplo, algumas linguagens não diferenciam maiúsculas de minúsculas, não oferecem tipos de dados com sinal, não suportam métodos sobrecarregados ou métodos que possuem um número variado de parâmetros. Portanto, para utilizar características de uma determinada linguagem em outra, os desenvolvedores precisam ter a garantia de que estas características estejam disponíveis em todas as outras linguagens.

Para auxiliar este processo, a Microsoft definiu o *Common Language Specification* (CLS). O CLS detalha o conjunto mínimo de características que um compilador deve suportar para gerar código a ser executado pelo CLR.

A Figura 7 exibe a estrutura do CLR/CTS. Algumas linguagens de programação expõem um grande subconjunto do CLR/CTS. Um desenvolvedor que optar por escrever o seu código em MSIL está apto a utilizar todo o conjunto de classes que o CLR/CTS oferece. As demais linguagens (C#, Visual Basic e Fortran) expõem um subconjunto de classes para o programador. O CLS define o conjunto mínimo de classes que todas as linguagens devem suportar.



Fonte: Richter (2002, p. 27)

Figura 7 - Exemplos de linguagens que oferecem um sub-conjunto do CLR/CTS

2.3.5 MICROSOFT *INTERMEDIATE LANGUAGE* (MSIL)

Segundo Richter (2002, p. 11), o MSIL é uma linguagem de máquina de alto nível, independente de CPU. O MSIL é capaz de interpretar classes de objetos e contém instruções para criar e inicializar objetos, fazer chamadas de funções virtuais de objetos e manipulação direta de elementos de matrizes. O MSIL possui também instruções para tratamento de exceções, fazendo desta linguagem "uma linguagem de máquina orientada a objetos" (RICHTER, 2002, p. 12).

O MSIL é baseado em uma máquina de pilhas (*stack-based*). Isto significa que todas as instruções inserem valores em uma pilha de execução e retiram os resultados desta pilha. Os desenvolvedores de compiladores não precisam se preocupar com o gerenciamento de registradores, pois o MSIL não possui instruções para manipulação de registradores (RICHTER, 2002, p. 19).

As instruções do MSIL operam diretamente com os operandos que estão na pilha, desconsiderando os tipos dos operandos. Por exemplo, não existe uma instrução `add` específica para adicionar um tipo inteiro de 32 *bits* e outra instrução para um tipo inteiro de 64 *bits*. Quando a operação `add` é executada, a própria instrução determina os tipos dos operandos na pilha e executa a instrução apropriada.

Richter (2002, p. 19) afirma que uma das maiores vantagens do MSIL é a segurança das aplicações. Durante a tradução do código MSIL para instruções nativas da CPU, o CLR realiza uma "verificação" do código MSIL. Esta verificação certifica que uma determinada operação pode ser executada sem problemas, como por exemplo, impedir a leitura de uma área de memória não inicializada, verificação do número e dos tipos dos parâmetros nas chamadas de métodos e assim por diante.

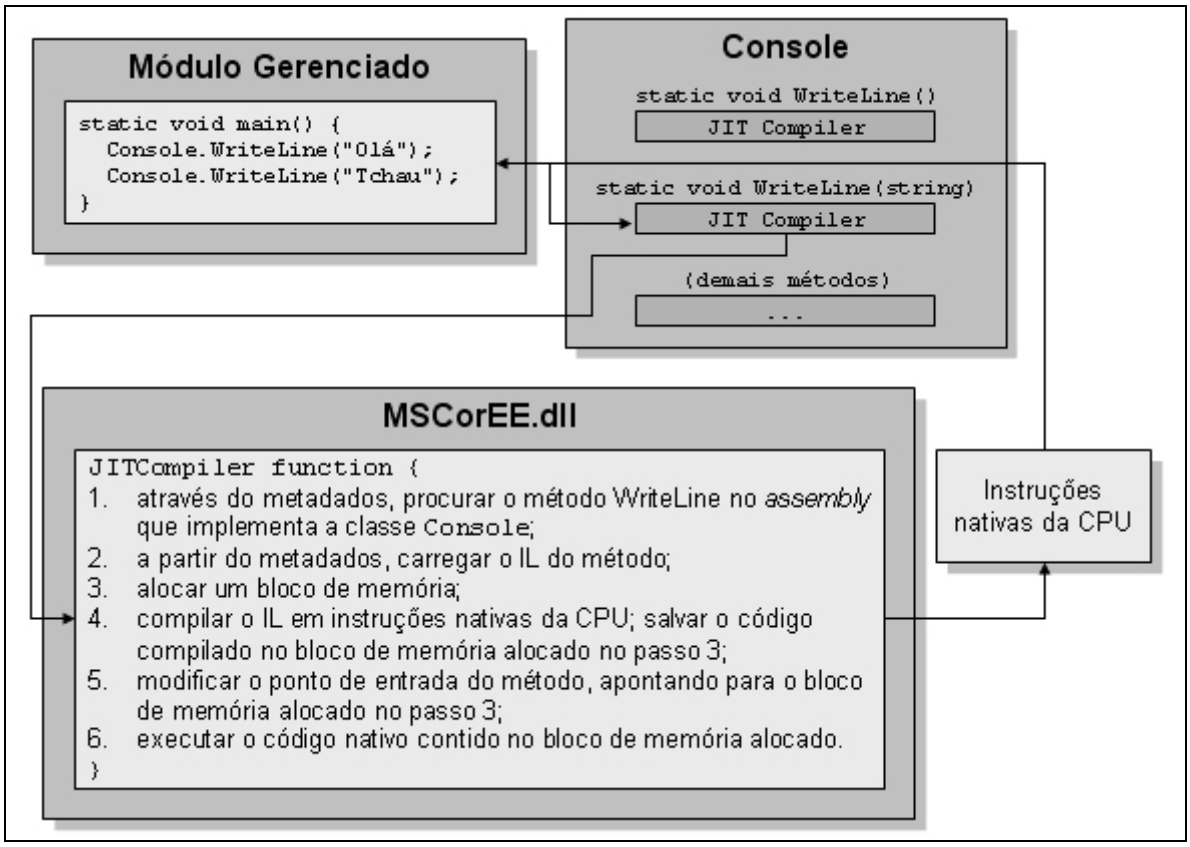
Existem cerca de 220 instruções no MSIL. Dois terços destas instruções são instruções básicas e o restante são instruções para manipulação de objetos (GOUGH, 2002, p. 28). O conjunto de instruções do MSIL é descrito no anexo A.

2.3.6 *JUST-IN-TIME COMPILER* (JIT)

Segundo Richter (2002, p. 15), as CPUs atuais não podem executar as instruções do MSIL diretamente. Para ser executado o MSIL precisa ser traduzido para instruções nativas da CPU. A tradução das instruções é realizada pelo compilador *Just-In-Time* (JIT). A plataforma Microsoft .NET oferece três tipos de compiladores JIT.

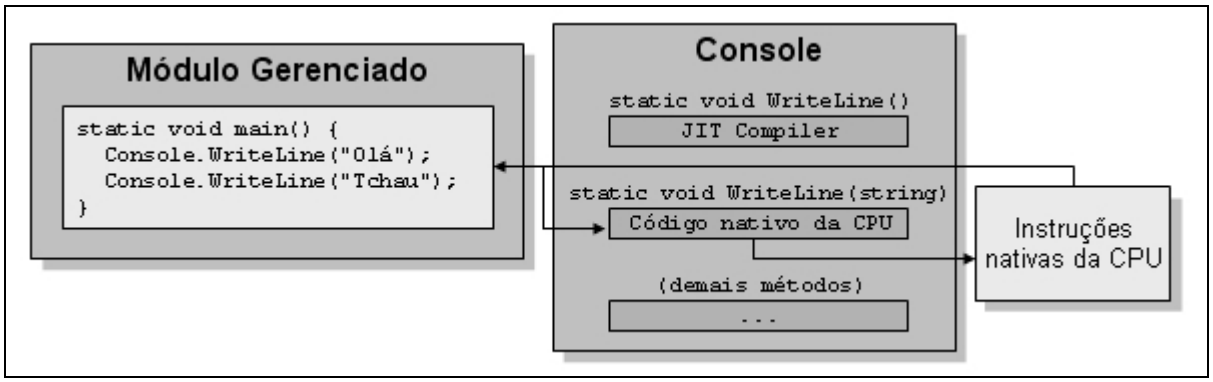
O compilador JIT padrão (*DefaultJIT*) faz a tradução do código MSIL para instruções nativas da CPU método por método, conforme seja necessário (Figura 8). O código traduzido é armazenado para ser acessado pelas chamadas subseqüentes, caso existam (Figura 9). Dessa maneira, o desempenho dos aplicativos que utilizam o *DefaultJIT* é incrementada conforme o uso. Este método é principalmente utilizado em ambientes com grandes quantidades de memória.

O segundo tipo de compilador JIT é o *EconoJIT*. Semelhante ao *DefaultJIT*, a tradução ocorre conforme seja necessário. Entretanto, a tradução é realizada de maneira mais rápida, mas produzindo um código de máquina não otimizado. Além disso, o código traduzido não é armazenado para ser usado pelas chamadas seguintes. O método *EconoJIT* é utilizado em ambientes com quantidade de memória reduzidas.



Fonte: Richter (2002, p. 15)

Figura 8 - Chamando um método pela primeira vez



Fonte: Richter (2002, p. 17)

Figura 9 - Chamando um método pela segunda vez

O último tipo de compilador JIT é chamado *PreJIT*. Este método também é semelhante ao *DefaultJIT*, mas a conversão do MSIL para código nativo ocorre na instalação dos *assemblies*. Durante a instalação, grandes partes de código MSIL são compiladas e armazenadas para posterior execução. Quando o *assembly* é carregado, o CLR verifica se existe uma versão pré-compilada do *assembly* e, caso exista, o CLR carrega este *assembly* tornando desnecessária a compilação em tempo de execução (RICHTER, 2002, p. 19).

Antes da execução do módulo gerenciado, o CLR detecta todas as classes que são referenciadas pelo código, alocando todas as estruturas de dados necessárias. Estas estruturas de dados possuem uma entrada para cada método definido pelas classes utilizadas contendo o endereço onde as implementações destes métodos podem ser encontradas.

3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo é descrita a implementação do compilador e do ambiente de programação para a linguagem proposta. A seção 3.1 apresenta as especificações léxicas, sintáticas e semânticas da nova linguagem de programação. Os *tokens* e a sintaxe da nova linguagem são definidos utilizando a notação BNF. A semântica da linguagem proposta é apresentada através de programas exemplos. Na seção 3.2 é descrita a especificação e a implementação do compilador. Através do uso da ferramenta *ProGrammar,* são introduzidas as ações semânticas. As ações semânticas montam um arquivo texto ".IL" que contém o código MSIL a ser processado pelo montador ILAsm. Na seção 3.3 é apresentado o ambiente de programação para a nova linguagem de programação. A operacionalidade do ambiente também é descrita nesta seção.

3.1 DEFINIÇÃO DA LINGUAGEM PROPOSTA

A nova linguagem desenvolvida constitui uma linguagem de programação seqüencial imperativa que gera aplicações do tipo *console* a ser executada na plataforma Microsoft .NET. A linguagem proposta possui as seguintes características:

a) comandos em língua portuguesa;
b) estrutura semelhante a linguagem C;
c) *case-sensitive*.

Os comandos da linguagem proposta incluem a declaração de variáveis locais, um comando de seleção, um comando de repetição, um comando para entrada e saída de dados, definição e uso de módulos.

3.1.1 ESPECIFICAÇÃO LÉXICA DA LINGUAGEM

Os *tokens* da linguagem são definidos através da notação BNF. Os termos em negrito constituem os símbolos da linguagem. Os termos envolvidos por { e } podem ser repetidos *n* vezes e os termos envolvidos por [e] são opcionais. O símbolo + após um termo envolvido por { e } indica que deve existir no mínimo uma ocorrência do termo. Um conjunto de caracteres é especificado com a seqüência .. como, por exemplo, a..z especifica o conjunto de todas as letras entre a e z. Para especificar um caractere da tabela ASCII, é utilizado a barra invertida (\), seguido do código decimal do caractere (por exemplo, \65 especifica o caractere A).

Existem duas maneiras para comentar o código fonte na linguagem proposta: comentário em bloco e comentário em linha. O comentário em bloco é iniciado com a seqüência `/*` e é finalizado com a seqüência `*/`. O comentário em linha é especificado com `//`. Nos comentários podem existir quaisquer tipos de caracteres.

Os espaços e tabulações entre os *tokens* são opcionais, com exceção das palavras reservadas, que devem ser envolvidas por espaços ou por nova linha.

Os identificadores da linguagem devem iniciar com uma letra, seguido por letras, dígitos ou o caractere sublinhado (_) (Quadro 2). Letras maiúsculas são diferentes de minúsculas. As palavras reservadas descritas no Quadro 3 não podem ser utilizadas como identificadores. Os identificadores não possuem um comprimento máximo para seu tamanho.

```
identifier ::= {L}+{L|D|_}
L ::= a..zA..Z
D ::= 0..9
```

Quadro 2 - Definição de um identificador

programa	logico	byte	caractere	inteiro	real	cadeia	vazio
retorne	escreva	leia	se	senao	enquanto		
verdadeiro	falso						

Quadro 3 - Palavras reservadas

A linguagem proposta reconhece as constantes numéricas inteiras e reais (Quadro 4). Os limites para as constantes inteiras e reais são apresentados na seção 3.1.3.1.

```
integer ::= {D}+
float   ::= {D}+[.{D}+]|.{D}+
```

Quadro 4 - Definição de uma constante numérica

As constantes cadeias (Quadro 5) são envolvidas por ", como por exemplo a constante "olá mundo", e podem conter qualquer caractere (exceto a combinação nova linha + retorno de cursor). Semelhante a linguagem C, algumas seqüências de caracteres (seqüências de escape) possuem um significado especial, conforme é mostrado na Tabela 3.

```
string ::= □{C}□
```

Quadro 5 - Definição de uma constante cadeia

Tabela 3 - Seqüências de escape

Seqüência	Descrição
`\n`	Nova linha
`\t`	Tabulação horizontal
`\v`	Tabulação vertical
`\b`	*Backspace*
`\r`	Retorno de cursor
`\f`	Avanço de formulário
`\a`	Alerta (*bell*)
`\\`	Barra invertida
`\□`	Apóstrofo
`\□`	Aspas

Fonte: Ellis e Stroustrup (1993, p. 13)

Semelhante as constantes literais, as constantes caracteres (Quadro 6) são envolvidas por `'` e devem conter apenas um caractere, como por exemplo a constante `'a'`.

```
character ::= □C□
C ::= \32..\255
```

Quadro 6 - Definição de uma constante caractere

A linguagem proposta também define duas constantes lógicas que podem conter os seguintes valores: `verdadeiro` e `falso`.

Os operadores relacionais, aritméticos e lógicos que podem ser usados na linguagem são apresentados na Tabela 4.

Tabela 4 - Operadores relacionais, aritméticos e lógicos

Operador	Descrição
`==`	Igual a
`!=`	Não igual a
`<`	Menor que
`>`	Maior que
`<=`	Menor que ou igual a
`>=`	Maior que ou igual a
`+`	Adição
`-`	Subtração
`*`	Multiplicação
`/`	Divisão
`%`	Resto da divisão
`&&`	E lógico
`||`	Ou lógico
`!`	Negação lógica

3.1.2 ESPECIFICAÇÃO SINTÁTICA DA LINGUAGEM

Um programa na linguagem proposta consiste em uma seqüência de declarações de módulos dentro de um programa principal. A recursão é permitida e os parâmetros dos módulos são passados por valor. Os módulos podem ou não retornar um valor. Os comandos do programa são inseridos nos módulos.

Todo programa deve possuir um módulo denominado `principal()`. É este módulo quem define o ponto de entrada (*entry point*) para a execução do programa.

O Quadro 7 define a sintaxe da linguagem proposta através da notação BNF.

```
<compilation_unit> ::= <program_declaration>

<program_declaration> ::=
  programa <identifier>
  { [<method_declarations>] }

<method_declarations> ::=
  { <method_declaration> }

<method_declaration> ::=
  <type_specifier>
  <identifier>
  ( [ <parameter_list> ] )
  <statement_block>

<type_specifier> ::=
    logico | byte | caractere | inteiro | real | cadeia
  | vazio

<parameter_list> ::=
  <parameter> [, <parameter_list>]

<parameter> ::=
  <type_specifier>
  <identifier>

<statement_block> ::=
  { [{ <statement> }] }

<statement> ::=
    <statement_block>
  | <variable_declaration>
  | <if_statement>
  | <while_statement>
  | <return_statement>
  | <write_statement>
  | <read_statement>
  | <assign_statement>
  | <call_statement>

<variable_declaration> ::=
  <type_specifier>
  <identifier>
  [= <variable_initializer>] ;

<variable_initializer> ::=
  <expression> ;

<if_statement> ::=
  se
  ( <expression> )
  <statement>
  [ senao <statement> ]

<while_statement> ::=
  enquanto
  ( <expression> )
  <statement>

<return_statement> ::=
  retorne [ <expression> ] ;

<write_statement> ::=
  escreva ( <argument_list> ) ;

<argument_list> ::=
  <expression> [, <argument list>]
```

```
<read_statement> ::=
  leia ( <identifier_list> ) ;

<identifier_list> ::=
  <identifier> [, <identifier_list>]

<assign_statement> ::=
  <identifier> <assign_op> <expression> ;

<assign_op> ::= = | *= | /= | %= | += | -=

<call_statement> ::=
  <identifier> ( [argument_list] ) ;


<expression> ::= <or_term>

<or_term>    ::= <and_term> [<or_level_op> <or_term>]
<and_term>   ::= <cmp_term> [<and_level_op> <and_term>]
<cmp_term>   ::= <add_term> [<cmp_level_op> <cmp_term>]
<add_term>   ::= <mult_term> [<add_level_op> <add_term>]
<mult_term>  ::= <factor> [<mult_level_op> <mult_term>]

<factor> ::=
    <reference_term>
  | <literal_expression>
  | <casting_expression>
  | ( <expression> )
  | <negation_op> <factor>
  | <unary_op> <factor>

<or_level_op>   ::= ||
<and_level_op>  ::= &&
<cmp_level_op>  ::= == | != | < | > | >= | <=
<add_level_op>  ::= + | -
<mult_level_op> ::= / | * | %
<unary_op>      ::= - | +
<negation_op>   ::= !

<reference_term> ::=
  <identifier> [ ( [<argument_list>] ) ]

<casting_expression> ::=
  ( <type_specifier> ) <expression>

<literal_expression> ::=
    <numeric_literal>
  | <boolean_literal>
  | <string>
  | <character>

<numeric_literal> ::=
    <integer>
  | <float>

<boolean_literal> ::= verdadeiro | falso

<integer>    ::= {D}+
<float>      ::= {D}+[.{D}+]|.{D}+
<identifier> ::= {L}+{L|D|_}
<string>     ::= "{C}"
<character>  ::= 'C'

D ::= 0..9
L ::= a..zA..Z
C ::= \32..\255
```

Quadro 7 - Sintaxe da linguagem proposta (BNF)

3.1.3 ESPECIFICAÇÃO SEMÂNTICA DA LINGUAGEM

A linguagem proposta possui sete tipos de dados: `vazio` (usado somente para identificar módulos que não retornam um valor), `byte`, `logico`, `caractere`, `inteiro`, `real` e `cadeia`.

Os comandos oferecidos pela linguagem são a declaração de variáveis, comando de saída, comando de entrada, comando de atribuição, comando de seleção, comando de repetição, comando de retorno e chamadas de módulos. São disponibilizados também operadores unários, aritméticos (multiplicativos e aditivos), lógicos e relacionais a serem utilizados em expressões.

3.1.3.1 TIPOS DE DADOS

Os tipos de dados são utilizados para definir o tipo do retorno dos módulos e o conteúdo das variáveis dentro do programa.

A Tabela 5 define os tipos de dados suportados pela linguagem proposta, além de algumas características dos mesmos.

Tabela 5 - Tipos de dados da linguagem

Tipo	**Descrição**	**Exemplos**

`vazio`	Usado em módulos que não retornam um valor.	`vazio principal()`
		`{`
		`  ...`
		`}`

`byte`	Inteiro de 8 *bits*.	`byte a = 10;`

`logico`	Valor lógico: verdadeiro ou falso.	`logico b = verdadeiro;`
		`enquanto (achou == falso)`
		`{`
		`  ...`
		`}`

`caractere`	Caractere de 16 *bits*.	`caractere c = □a□;`
`inteiro`	Inteiro de 32 *bits*.	`inteiro i = 15000;`
`real`	Ponto flutuante de 64 *bits*.	`real r = 15.5;`
`cadeia`	Texto.	`cadeia t = □olá mundo□;`

3.1.3.2 DECLARAÇÃO DE MÓDULOS

Os módulos são as estruturas da linguagem onde os comandos estão inseridos. Um programa obrigatoriamente deve conter o módulo `principal()`. O Quadro 8 exibe a forma e o código MSIL da declaração de um módulo.

Sintaxe	Código MSIL

`type_specifier identifier([parameter_list])`
`  statement_block`
	`.method public static type_specifier_ilasm`
	`identifier(parameter_list)`
	`  statement_block`

Quadro 8 - Declaração de módulos

O bloco de comandos é uma seqüência de comandos a serem executados. Os comandos são separados por ponto-e-vírgula (`;`). Opcionalmente, uma lista de parâmetros poderá ser definida, separados por vírgula no caso de existir mais de um. O formato e o código MSIL dos parâmetros são apresentados no Quadro 9.

Sintaxe	Código MSIL
`type_specifier identifier,` `type_specifier identifier, ...`	`type_specifier_ilasm A_n,` `type_specifier_ilasm A_n+1, ...`

Quadro 9 - Definição de um parâmetro em um módulo

O Quadro 10 apresenta um exemplo da definição de dois módulos e o código MSIL gerado.

Exemplo	Código MSIL

`programa teste`
`{`
`  vazio principal()`
`  {`
`  // comandos`
`  }`
 	
`  real calculaMedia(inteiro a, inteiro b)`
`  {`
`  // comandos`
`  }`
`}`
	`.assembly extern mscorlib { }`
	`.assembly teste { }`
	`.module teste.exe`
	
	`.class public teste`
	`{`
	`  .method public static void principal()`
	`  {`
	`  .entrypoint`
	`  ret`
	`  }`
	`  .method public static float64`
	`  calculaMedia(int32 A_0,int32 A_1)`
	`  {`
	`  ret`
	`  }`
	`}`

Quadro 10 - Exemplo da definição de módulos

3.1.3.3 DECLARAÇÃO DE VARIÁVEIS

As variáveis da linguagem proposta são fortemente tipadas e podem ser declaradas em qualquer posição dentro de um módulo. Variáveis com o mesmo nome podem ser declaradas mais de uma vez, desde que estejam em blocos de comandos (escopos) distintos.

A forma e o código MSIL da declaração de uma variável são apresentados no Quadro 11.

Sintaxe	Código MSIL
`type_specifier identifier;`	`.locals (type_specifier_ilasm V_n)`

Quadro 11 - Declaração de uma variável

Os tipos válidos para uma variável são: `byte`, `logico`, `caractere`, `inteiro`, `real` e `cadeia`. Uma variável não pode ser do tipo `vazio`. O intervalo dos tipos das variáveis são apresentados na Tabela 6.

Tabela 6 - Intervalo dos tipos de dados

Tipo	**Tamanho**	**Intervalo**
`byte`	8 *bits*.	0 até 255.
`logico`	32 *bits*.	-2.147.483.648 até 2.147.483.647. O valor 0 especifica `falso`, qualquer outro valor especifica `verdadeiro`.
`caractere`	16 *bits*.	0 até 65535 (caractere *unicode*).
`inteiro`	32 *bits*.	-2.147.483.648 até 2.147.483.647.
`real`	64 *bits*.	-1.79769313486232E+307 até 1.79769313486232E+307.
`cadeia`	32 *bits* (ponteiro para objeto).	-

Opcionalmente, uma variável pode ser inicializada na sua declaração através do comando de atribuição. O comando de atribuição é definido na seção 3.1.3.6.

O Quadro 12 exibe um exemplo da declaração de variáveis e o código MSIL gerado.

Exemplo	**Código MSIL**

`programa teste`
`{`
`  vazio principal()`
`  {`
`  inteiro idade;`
`  real peso = 80.0; // inicializacao`
`  }`
`}`
	`.assembly extern mscorlib { }`
	`.assembly teste { }`
	`.module teste.exe`
	
	`.class public teste`
	`{`
	`  .method public static void principal()`
	`  {`
	`  .entrypoint`
	`  .locals (int32 V_0)`
	`  .locals (float64 V_1)`
	`  ldc.r8 80.0`
	`  stloc  V_1`
	`  ret`
	`  }`
	`}`

Quadro 12 - Exemplo de declaração de variáveis

3.1.3.4 COMANDO DE SAÍDA

O comando de saída `escreva` é usado para exibir dados (variáveis e/ou expressões) na tela. A sintaxe e o código MSIL do comando de saída são dados no Quadro 13.

Sintaxe	**Código MSIL**

`escreva``(argument_list);`	`// para cada argumento da lista argument_list é realizada uma`
	`// chamada para o metodo [mscorlib]System.Console.Write(...)`
	`call void [mscorlib]System.Console.Write(type_specifier_ilasm)`
	`call void [mscorlib]System.Console.Write(type_specifier_ilasm)`
	`...`

Quadro 13- Comando de saída

Os parâmetros devem ser separados por vírgula, caso exista mais de um. O Quadro 14 mostra um exemplo do comando de saída e o respectivo código MSIL.

Exemplo	Código MSIL

programa teste
{
  vazio principal()
  {
  inteiro numero = 10;
  escreva(numero, "\n");
  }
}
	.assembly extern mscorlib { }
	.assembly teste { }
	.module teste.exe
	
	.class public teste
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  .locals (int32 V_0)
	  ldc.i4 10
	  stloc V_0
	  ldloc V_0
	  call void [mscorlib]System.Console::Write(int32)
	  ldstr "\n"
	  call void [mscorlib]System.Console::Write(string)
	  ret
	  }
	}

Quadro 14 - Exemplo do comando de saída

3.1.3.5 COMANDO DE ENTRADA

Através do comando de entrada `leia`, o programador pode requisitar dados para o programa a partir do teclado. O Quadro 15 fornece a sintaxe e o código MSIL para o comando de entrada.

Sintaxe	Código MSIL

leia(identifier_list**);**	// para cada argumento da lista identifier_list é realizada uma
	// chamada para [mscorlib]System.type_specifier_cts::Parse(string)
	call type_specifier_ilasm
	[mscorlib]System.type_specifier_cts::Parse(string)
	call type_specifier_ilasm
	[mscorlib]System.type_specifier_cts::Parse(string)
	...

Quadro 15 - Comando de entrada

Semelhante ao comando de saída `escreva`, podem-se definir múltiplas variáveis a serem lidas, separando-as por vírgula, conforme exemplo dado no Quadro 16.

Exemplo	Código MSIL

programa teste
{
  vazio principal()
  {
  inteiro idade;
  real peso;
 	
  leia(idade);
  leia(peso);
  }
}
	.assembly extern mscorlib { }
	.assembly teste { }
	.module teste.exe
	
	.class public teste
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  .locals (int32 V_0)
	  .locals (float64 V_1)
	  call string [mscorlib]System.Console::ReadLine()
	  call int32 [mscorlib]System.Int32::Parse(string)
	  stloc V_0
	  call string [mscorlib]System.Console::ReadLine()
	  call float64 [mscorlib]System.Double::Parse(string)
	  stloc V_1
	  ret
	  }
	}

Quadro 16 - Exemplo do comando de entrada

Durante a execução do programa, os dados informados devem estar em conformidade com o tipo de dado a ser lido. Por exemplo, se a variável de destino for um número inteiro, o dado a ser informado deve ser um número inteiro. Caso seja informado um tipo de dado inválido, uma exceção é gerada e a execução do aplicativo é abortada. Para o tipo `logico`, os possíveis valores a serem lidos são *`true`* e *`false`*. Para o tipo `real`, considerar a vírgula como separador de decimal.

3.1.3.6 COMANDO DE ATRIBUIÇÃO

O comando de atribuição é responsável pela inicialização e atualização das variáveis declaradas no programa. Sua sintaxe e código MSIL são dados no Quadro 17.

Sintaxe	Código MSIL
identifier assign_op expression;	stloc V_n

Quadro 17 - Sintaxe do comando de atribuição

A linguagem proposta define os seguintes operadores de atribuição: =, +=, -=, *=, /= e %=. Uma expressão geralmente resulta em um valor e pode causar efeitos colaterais, como a perda de dados. As expressões e os operadores de atribuição são descritos na seção 3.1.3.11. O Quadro 18 fornece alguns exemplos do comando de atribuição.

Exemplo	Código MSIL

programa teste
{
  vazio principal()
  {
  inteiro idade;
  real peso = 83.5;
 	
  peso += 10;
  idade = 15;
  }
}
	.assembly extern mscorlib { }
	.assembly teste { }
	.module teste.exe
	
	.class public teste
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  .locals (int32 V_0)
	  .locals (float64 V_1)
	  ldc.r8 83.5
	  stloc V_1
	  ldloc V_1
	  ldc.i4 10
	  conv.r8
	  add
	  stloc V_1
	  ldc.i4 15
	  stloc V_0
	  ret
	  }
	}

Quadro 18 - Exemplo do comando de atribuição

3.1.3.7 COMANDO DE SELEÇÃO

O comando de seleção `se` realiza um teste lógico em determinada expressão e executa ou não o bloco de comandos seguinte. O Quadro 19 exibe as possíveis sintaxes e o código MSIL do comando de seleção.

Sintaxe	Código MSIL

se (expression)
  statement
	brfalse lb000000
	  ...
	lb000000:

se (expression)
  statement
senao
  statement
	brfalse lb000000
	  ...
	  br lb000001
	lb000000:
	  ...
	lb000001:

Quadro 19 - Comando de seleção

O uso do comando de seleção é ilustrado no Quadro 20.

Exemplo	Código MSIL

programa teste
{
  vazio principal()
  {
  inteiro idade;
  leia(idade);
 	
  se (idade < 0)
  escreva("não nascido");
  senao
  escreva("nasceu");
  }
}
	.assembly extern mscorlib { }
	.assembly teste { }
	.module teste.exe
	
	.class public teste
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  .locals (int32 V_0)
	  call string [mscorlib]System.Console::ReadLine()
	  call int32 [mscorlib]System.Int32::Parse(string)
	  stloc V_0
	  ldloc V_0
	  ldc.i4 0
	  clt
	  brfalse lb000000
	  ldstr "não nascido"
	  call void [mscorlib]System.Console::Write(string)
	  br lb000001
	  lb000000:
	  ldstr "nasceu"
	  call void [mscorlib]System.Console::Write(string)
	  lb000001:
	  ret
	  }
	}

Quadro 20 - Exemplo do comando de seleção

3.1.3.8 COMANDO DE REPETIÇÃO

Semelhante ao comando de seleção, o comando de repetição `enquanto` também realiza um teste lógico, repetindo a execução do bloco de comandos até que a condição seja satisfeita. A sintaxe e o código MSIL do comando de repetição são dados no Quadro 21.

Sintaxe	Código MSIL

enquanto (expression)
  statement
	lb000000:
	  ...
	brfalse lb000001
	  ...
	br lb000000
	lb000001:

Quadro 21 - Comando de repetição

Conforme ilustrado no Quadro 22, é necessário incluir um comando que atualize a condição a ser satisfeita, caso contrário, o programa ficará executando eternamente o comando de repetição.

Exemplo	Código MSIL

programa teste
{
  vazio principal()
  {
  inteiro n = 0;
 	
  enquanto (n <= 10)
  {
  escreva(n * 2, "\n");
  n += 1;
  }
  }
}
	.assembly extern mscorlib { }
	.assembly teste { }
	.module teste.exe
	
	.class public teste
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  .locals (int32 V_0)
	  ldc.i4 0
	  stloc V_0
	  lb000000:
	  ldloc V_0
	  ldc.i4 10
	  cgt
	  ldc.i4 0
	  ceq
	  brfalse lb000001
	  {
	  ldloc V_0
	  ldc.i4 2
	  mul
	  call void [mscorlib]System.Console::Write(int32)
	  ldstr "\n"
	  call void [mscorlib]System.Console::Write(string)
	  ldloc V_0
	  ldc.i4 1
	  add
	  stloc V_0
	  }
	  br lb000000
	  lb000001:
	  ret
	  }
	}

Quadro 22 - Exemplo do comando de repetição

3.1.3.9 USO DE MÓDULOS

Os módulos são referenciados no texto do programa como as variáveis, exceto pelo fato de possuírem uma lista de parâmetros delimitados por (e) e separados por vírgula (,) caso existirem mais de um. O Quadro 23 exibe a sintaxe e o código MSIL para o uso de módulos.

Sintaxe	Código MSIL
identifier([argument_list]);	call type_specifier_ilasm program_name::identifier(type_specifier_ilasm, type_specifier_ilasm, ...)

Quadro 23 - Uso de módulos

Um módulo pode chamar ele mesmo, possibilitando dessa forma a recursão (Quadro 24). Na chamada de um módulo, o número e o tipo dos parâmetros devem ser respeitados conforme a declaração do módulo.

Exemplo	Código MSIL

programa fatorial
{
  vazio principal()
  {
  escreva(fatorial(10), "\n");
  }
 	
  real fatorial(inteiro n)
  {
  se (n == 0)
  {
  retorne 1;
  }
 	
  retorne (n * fatorial(n-1));
  }
}
	.assembly extern mscorlib { }
	.assembly fatorial { }
	.module fatorial.exe
	
	.class public fatorial
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  ldc.i4 10
	  call float64 fatorial::fatorial(int32)
	  call void [mscorlib]System.Console::Write(float64)
	  ldstr "\n"
	  call void [mscorlib]System.Console::Write(string)
	  ret
	  }
	  .method public static float64 fatorial(int32 A_0)
	  {
	  ldarg A_0
	  ldc.i4 0
	  ceq
	  brfalse lb000000
	  {
	  ldc.i4 1
	  conv.r8
	  ret
	  }
	  lb000000:
	  ldarg A_0
	  conv.r8
	  ldarg A_0
	  ldc.i4 1
	  sub
	  call float64 fatorial::fatorial(int32)
	  mul
	  ret
	  }
	}

Quadro 24 - Exemplo de uso de módulos

3.1.3.10 COMANDO DE RETORNO

O comando de retorno `retorne` determina o fim da execução de um módulo. Os módulos que retornam um valor diferente de `vazio` obrigatoriamente devem possuir um comando de retorno. A sintaxe e o código MSIL do comando de retorno são fornecidos no Quadro 25. Um exemplo do comando de retorno é dado no Quadro 26.

Sintaxe	Código MSIL
retorne expression;	ret

Quadro 25 - Comando de retorno

Exemplo	Código MSIL

programa teste
{
  vazio principal()
  {
  escreva(calculaMedia(8, 10), "\n");
  }
 	
  real calculaMedia(inteiro a, inteiro b)
  {
  real media = (a + b) / 2;
 	
  retorne media;
  }
}
	.assembly extern mscorlib { }
	.assembly teste { }
	.module teste.exe
	
	.class public teste
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  ldc.i4 8
	  ldc.i4 10
	  call float64
	teste::calculaMedia(int32,int32)
	  call void
	[mscorlib]System.Console::Write(float64)
	  ldstr "\n"
	  call void
	[mscorlib]System.Console::Write(string)
	  ret
	  }
	  .method public static float64
	calculaMedia(int32 A_0,int32 A_1)
	  {
	  .locals (float64 V_0)
	  ldarg A_0
	  ldarg A_1
	  add
	  ldc.i4 2
	  div
	  conv.r8
	  stloc V_0
	  ldloc V_0
	  ret
	  }
	}

Quadro 26 - Exemplo do comando de retorno

3.1.3.11 OPERANDOS E OPERADORES EM EXPRESSÕES

Esta seção define a sintaxe, a ordem de avaliação e o significado dos operandos e operadores nas expressões. Segundo Ellis e Stroustrup (1993, p. 57), uma expressão é uma seqüência de operadores e operandos que especifica uma computação. A linguagem proposta define seis tipos de operadores:

a) operadores unários;
b) operadores multiplicativos;
c) operadores aditivos;
d) operadores relacionais;
e) operadores lógicos;
f) operadores de atribuição.

As expressões com operadores unários e os operadores de atribuição são agrupados da direita para esquerda. Os demais operadores (multiplicativos, aditivos, relacionais e lógicos) são agrupados da esquerda para a direita.

Os tipos de operandos válidos para os operadores unários, multiplicativos, aditivos, relacionais e lógicos são: `logico`, `byte`, `caractere`, `inteiro` e `real`. Não é possível utilizar os tipos `vazio` e `cadeia` nestes operandos. O tipo `cadeia` somente pode ser usado com o operador de atribuição (`=`).

O resultado do operador de negação lógica (`!`) é 1 se o valor do operando é 0 e 0 caso o operando tenha um valor diferente de zero. O tipo do resultado é um inteiro de 32 *bits*.

O operador `*` indica multiplicação. O operador `/` indica divisão e o operador `%` produz o resto da divisão da primeira expressão pela segunda.

O resultado do operador + é a soma dos operandos e o resultado do operador − é a diferença entre os operandos.

Os operadores menor que (<), maior que (>), menor que ou igual a (<=) e maior que ou igual a (>=) produzem o valor 0 caso a relação especificada for falsa e 1 caso for verdadeira. O tipo do resultado é um inteiro de 32 *bits*. Deve-se atentar para o fato de `a<b<c` significar `(a<b)<c` e não `(a<b)` e `b<c)`. Os operadores igual a (==) e não igual a (!=) são análogos aos operadores relacionais <, <=, > e >=, exceto por sua precedência mais baixa.

O operador e lógico (`&&`) retorna 1 se ambos os seus operandos são diferentes de zero, e 0 em caso contrário. O operador ou lógico (`||`) retorna 1 se qualquer um dos seus operandos for diferente de zero, e 0 em caso contrário. O tipo do resultado para os operadores `&&` e `||` é um inteiro de 32 *bits*.

Os operadores de atribuição igual (=), adição igual (+=), subtração igual (−=), multiplicação igual (*=), divisão igual (/=) e resto da divisão igual (%=) necessitam de uma variável como seu operando esquerdo. O tipo do operando direito é convertido (quando possível) para o tipo do operando esquerdo.

O Quadro 27 exibe um programa com os operadores e operandos apresentados nesta seção.

Exemplo	Código MSIL

programa operandosoperadores
{
  vazio principal()
  {
  inteiro a;
 	
  a = 5;
 	
  escreva(a != 6);
  escreva(a < 6);
  escreva(a > 6);
  escreva(a <= 6);
  escreva(a >= 6);
  }
}
	.assembly extern mscorlib { }
	.assembly operandosoperadores { }
	.module operandosoperadores.exe
	
	.class public operandosoperadores
	{
	  .method public static void principal()
	  {
	  .entrypoint
	  .locals (int32 V_0)
	  ldc.i4 5
	  stloc V_0
	  ldloc V_0
	  ldc.i4 6
	  ceq
	  ldc.i4 0
	  ceq
	  call void
	[mscorlib]System.Console::Write(int32)
	  ldloc V_0
	  ldc.i4 6
	  clt
	  call void
	[mscorlib]System.Console::Write(int32)
	  ldloc V_0
	  ldc.i4 6
	  cgt
	  call void
	[mscorlib]System.Console::Write(int32)
	  ldloc V_0
	  ldc.i4 6
	  cgt
	  ldc.i4 0
	  ceq
	  call void
	[mscorlib]System.Console::Write(int32)
	  ldloc V_0
	  ldc.i4 6
	  clt
	  ldc.i4 0
	  ceq
	  call void
	[mscorlib]System.Console::Write(int32)
	  ret
	  }
	}

Quadro 27 - Exemplo de operadores e operandos

3.1.3.12 CONVERSÕES DE TIPOS

Geralmente os operandos das expressões podem ser de tipos diferentes. Nestes casos, conversões de tipos ocorrem automaticamente sem a intervenção do programador. Os operandos são sempre convertidos para o tipo mais abrangente da expressão.

Entretanto, no caso das atribuições, o compilador faz a conversão do tipo resultante da expressão para o tipo da variável a ser atribuída. Este comportamento pode levar a perda de dados e, portanto, o programador deve estar atento aos tipos envolvidos nas atribuições e expressões.

3.2 COMPILADOR

O compilador para a linguagem proposta foi especificado usando a *Unified Modeling Language* (UML) com a ferramenta Rational Rose Enterprise Edition. O compilador foi implementado utilizando a ferramenta Microsoft Visual C++ 6.0.

A ferramenta *ProGrammar* foi utilizada para proceder a análise léxica e sintática do programa fonte, a partir da gramática definida para a linguagem. A análise semântica e a geração de código são realizadas através da varredura da árvore de *parsing* gerada pela ferramenta.

A seção 3.2.1 descreve o uso da ferramenta *ProGrammar* no desenvolvimento do compilador. A seção 3.2.2 descreve a especificação do compilador. A implementação é descrita na seção 3.2.3.

3.2.1 USO DA FERRAMENTA *PROGRAMMAR*

Para tornar-se funcional, a ferramenta *ProGrammar* necessita de uma gramática para efetuar a análise de um texto de programa fonte. A gramática para a linguagem proposta, definida na seção 3.1.2 deve ser adaptada para uma forma que seja compreendida pela ferramenta *ProGrammar*.

3.2.1.1 ESPECIFICAÇÃO LÉXICA PARA A FERRAMENTA

As definições dos *tokens* utilizados pela gramática da linguagem proposta são mostrados na Tabela 7.

Tabela 7 - Definições dos *tokens* da gramática

Token	**Definição regular**	**Descrição**
`boolean_literal`	`"verdadeiro" | "falso"`	Os literais `verdadeiro` ou `falso`.
`numeric_literal`	`numeric ["." numeric] | "." numeric`	Um número seguido de . e número ou um . seguido de número.
`numeric`	`Definido internamente pela ferramenta.`	Um conjunto de no mínimo um ou mais números entre 0 e 9.
`string`	`quotedstring`	Um conjunto de 0 ou mais caracteres, exceto a combinação nova linha + retorno de cursor, delimitados por ".
`quotedstring`	`Definido internamente pela ferramenta.`	
`character`	`"□" □.□ "□"`	Um único caractere.
`ident`	`identifier`	Um conjunto composto por uma letra entre a-z e A-Z, seguido ou não de dígitos, letras ou o caractere _.
`identifier`	`Definido internamente pela ferramenta.`	

3.2.1.2 ESPECIFICAÇÃO SINTÁTICA PARA A FERRAMENTA

O Quadro 28 exibe a sintaxe da gramática da linguagem proposta no formato compreendido pela ferramenta *ProGrammar*. O formato é similar a BNF. A Tabela 8 exibe alguns atributos que podem ser utilizados na especificação dos símbolos da gramática.

Tabela 8 - Simbolos especiais para a ferramenta *ProGrammar*

Atributo	**Descrição**
SPACE	Especifica um conjunto de caracteres que devem ser considerados como espaço em branco ou o nome do símbolo que define um espaço em branco.
HIDELITERALS	Não insere os literais na árvore gramatical.
HIDEREPEATERS	Não insere os repetidores na árvore gramatical.
HIDDEN	Não insere os nós para este símbolo na árvore gramatical.
SHOWDELIMITERS	Insere os delimitadores na árvore gramatical.
TERMINAL	O símbolo é representado como um terminal na árvore gramatical.
? #VALUE	Avalia o valor mais recentemente recuperado do texto analisado.

Fonte: Norken Technologies (2004).

```
grammar KL <SPACE = space_symbol, HIDELITERALS,
HIDEREPEATERS>
{
  compilation_unit ::= [program_declaration] ;

  program_declaration ::=
    "programa" program_name
    "{" [method_declarations] "}" ;

  method_declarations ::=
    {method_declaration} ;

  method_declaration ::=
    type_specifier
    method_name
    "(" [parameter_list] ")"
    statement_block
    ;

  type_specifier<TERMINAL> ::=
      "logico"
    | "byte"
    | "caractere"
    | "inteiro"
    | "real"
    | "cadeia"
    | "vazio"
    ;

  parameter_list ::=
    {parameter, ","} ;

  parameter ::=
    type_specifier
    ident ;

  statement_block ::=
    "{" [{statement}] "}" ;

  statement ::=
      statement_block
    | variable_declaration
    | if_statement
    | while_statement
    | return_statement
    | write_statement
    | read_statement
    | assign_statement
    | call_statement
    ;

  variable_declaration ::=
    type_specifier
    ident
    ["=" variable_initializer] ";" ;

  variable_initializer ::=
     expression ;

  if_statement ::=
    "se"
    "(" expression ")"
    statement
    ["senao" statement] ;

  while_statement ::=
    "enquanto"
    "(" expression ")"
    statement ;

  return_statement ::=
    "retorne" [expression] ";" ;

  write_statement ::=
    "escreva" "(" argument_list ")" ";" ;

  read_statement ::=
    "leia" "(" {ident, ","} ")" ";" ;

  assign_statement ::=
    ident
    assign_op
    expression ";" ;

  assign_op ::= ( "=" | "*=" | "/=" | "%=" | "+=" | "-=" ) ;

  call_statement ::=
    ident "(" [argument_list] ")" ";" ;

  argument_list ::=
    {expression, ","} ;

  expression<SHOWDELIMITERS> ::=
    or_term ;

  or_term      ::= {and_term, or_level_op} ;
  and_term     ::= {cmp_term, and_level_op} ;
  cmp_term     ::= {add_term, cmp_level_op} ;
  add_term     ::= {mult_term, add_level_op} ;
  mult_term    ::= {factor, mult_level_op} ;

  factor ::=
      reference_term
    | literal_expression
    | casting_expression
    | "(" expression ")"
    | negation_op factor
    | unary_op factor
    ;

  or_level_op   ::= "||" ;
  and_level_op  ::= "&&" ;
  cmp_level_op  ::= "==" | "!=" | "<" | ">" | ">=" | "<=" ;
  add_level_op  ::= "+" | "-" ;
  mult_level_op ::= "*" | "/" | "%" ;
  unary_op      ::= "-" | "+" ;
  negation_op   ::= "!" ;

  reference_term ::=
      ident ["(" [argument_list] ")"];

  literal_expression ::=
      numeric_literal
    | boolean_literal
    | string
    | character
    ;

  casting_expression ::=
    "(" type_specifier ")" expression ;

  numeric_literal<TERMINAL> ::=
    (numeric ["." numeric] | "." numeric) ;

  boolean_literal<TERMINAL> ::=
    "verdadeiro" | "falso" ;

  string<TERMINAL> ::= quotedstring ;

  character<TERMINAL,SPACE=""> ::=
    "\'" '.' "\'" ;

  keyword ::=
      "programa"
    | "logico"
    | "byte"
    | "caractere"
    | "inteiro"
    | "real"
    | "cadeia"
    | "vazio"
    | "retorne"
    | "escreva"
    | "leia"
    | "se"
    | "senao"
    | "enquanto"
    | "verdadeiro"
    | "falso"
    ;

  method_name<TERMINAL>  ::= ident ;
  program_name<TERMINAL> ::= ident ;

  ident<TERMINAL> ::=
    identifier (? #value !::= keyword; ) ;

  space_symbol<TERMINAL,HIDDEN> ::=
    { ("\32" | "\r" | "\n" | "\t")
    | "//" *("\n")
    | "/*" *("*/") "*/", 0 } ;

};
```

Quadro 28 - Sintaxe da linguagem proposta no formato compreendido pela ferramenta *ProGrammar*

3.2.2 ESPECIFICAÇÃO DO COMPILADOR

O compilador (Figura 10) para a linguagem proposta deve ser capaz de compilar um programa fonte em código MSIL a ser processado pelo montador ILAsm. O compilador deve ser capaz também de informar erros léxicos, sintáticos e semânticos no programa fonte. O compilador para a linguagem proposta é definido na classe `CKLCompiler` (Figura 11). A ferramenta *ProGrammar* é referenciada através do atributo `m_pParser`, que é um ponteiro para a interface `ParserInterface` (Figura 12). A interface `ParserInterface` contém os métodos disponibilizados pela ferramenta *ProGrammar* para efetuar a análise de um texto segundo uma gramática. Esta análise é realizada através do método `Parse()`. O resultado desta análise é uma árvore gramatical, que pode ser percorrida através dos métodos da interface `ParserInterface`. A árvore gramatical é percorrida em duas passagens. Na primeira passagem, os métodos do programa são armazenados em uma tabela de métodos da classe `CKLProgram`. A tradução para o código MSIL e a atualização dos atributos da classe `CKLProgram` é realizada na segunda passagem. Desta maneira, o compilador para a linguagem proposta constitui em um compilador de duas passagens.

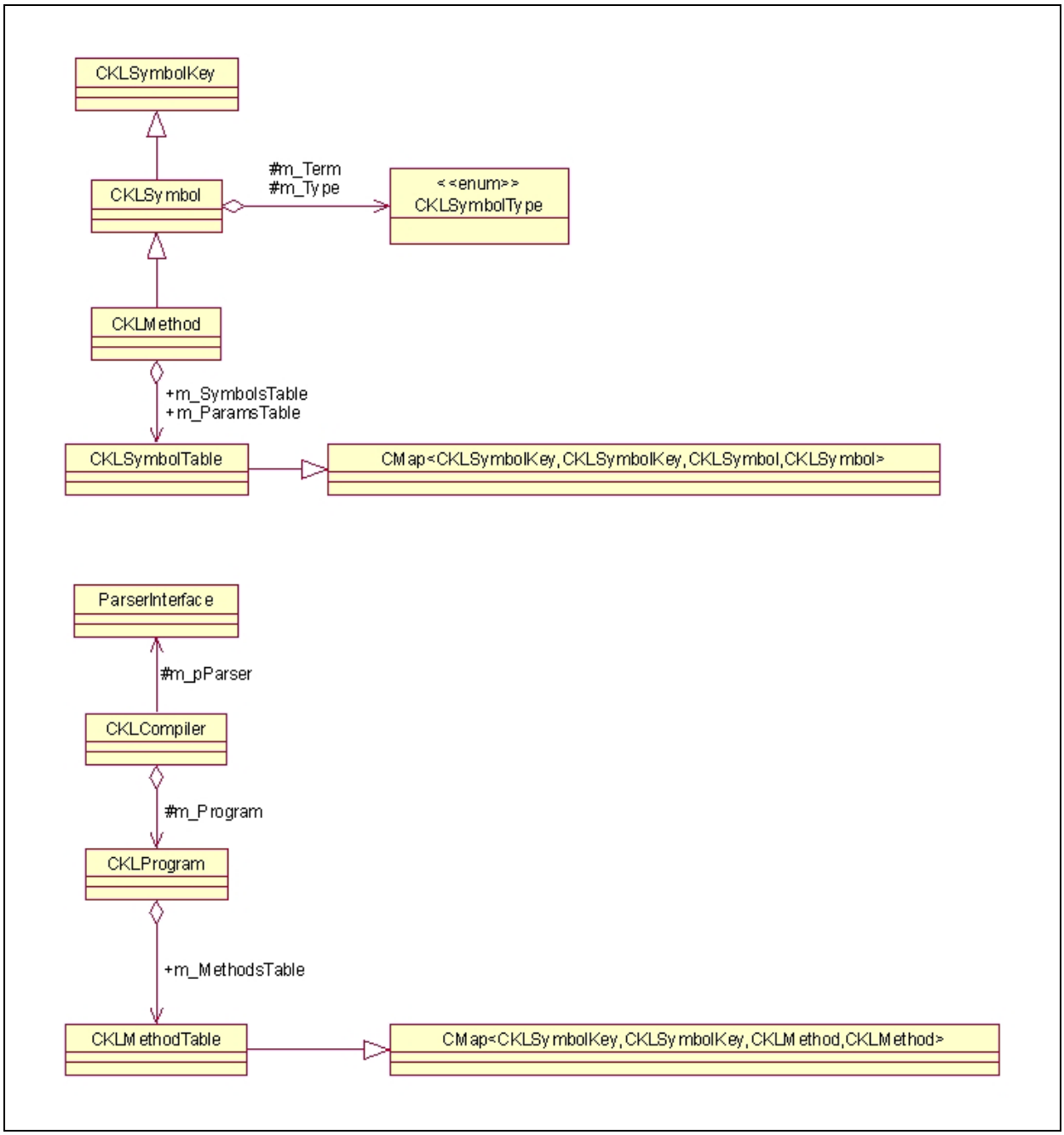


Figura 10 - Diagrama de classes do compilador para a linguagem proposta

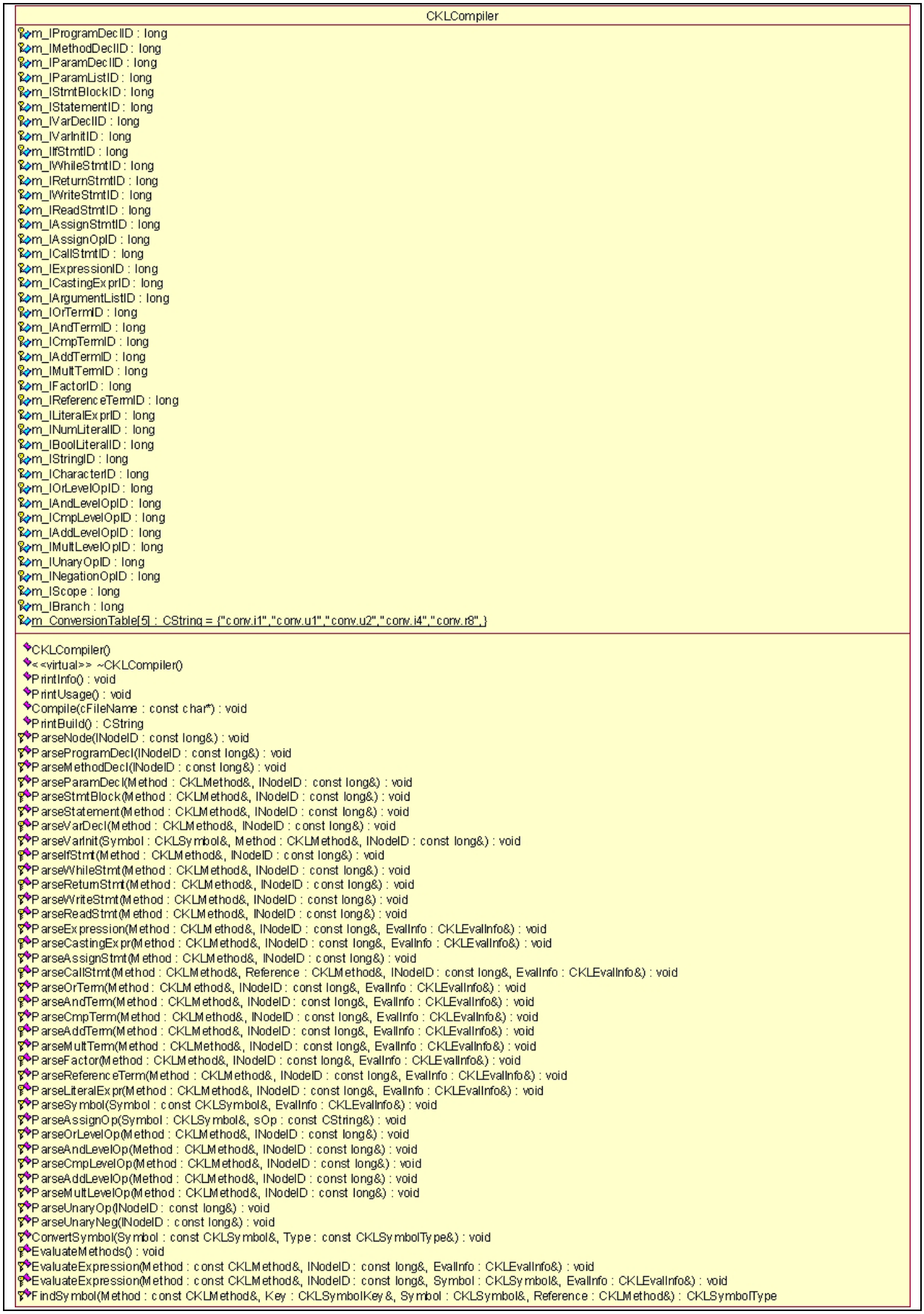


Figura 11 - Classe `CKLCompiler`

Figura 12 - Interface `ParserInterface`

A classe CKLProgram (Figura 13) é responsável pelo armazenamento das informações que são recuperadas a partir da árvore gramatical gerada pelo método Parse() da interface ParserInterface. A classe CKLCompiler é associada a classe CKLProgram através do atributo m_Program (Figura 14).

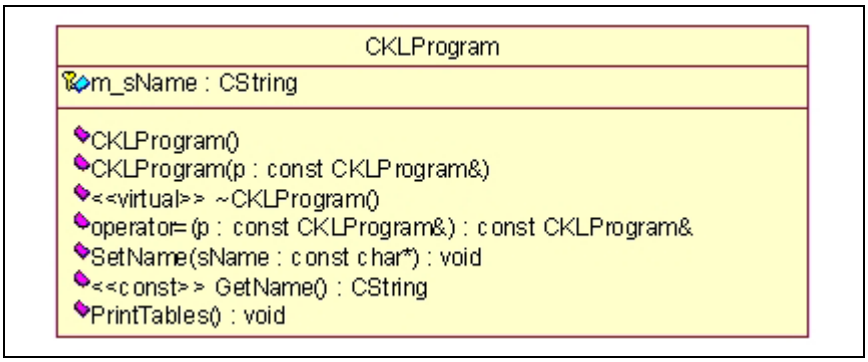


Figura 13 - Classe `CKLProgram`

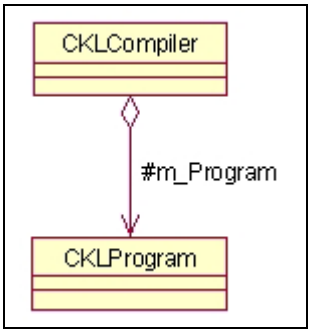


Figura 14 - Associação `CKLCompiler` - `CKLProgram`

A classe `CKLProgram` contém uma tabela de módulos (`m_MethodsTable`) definido pela classe `CKLMethodTable` (Figura 15). A classe `CKLMethodTable` é uma tabela *hash* para a classe `CKLMethod` (Figura 16). A classe `CKLMethod` é derivada de `CKLSymbol`, que por sua vez é derivada de `CKLSymbolKey`.

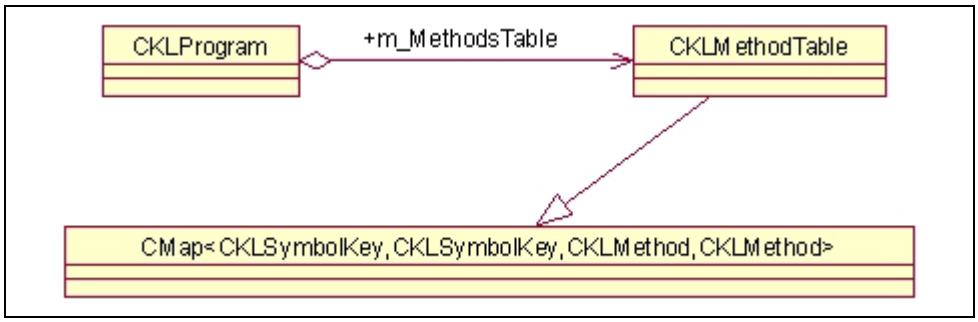


Figura 15 - Associação `CKLProgram` - `CKLMethodTable`

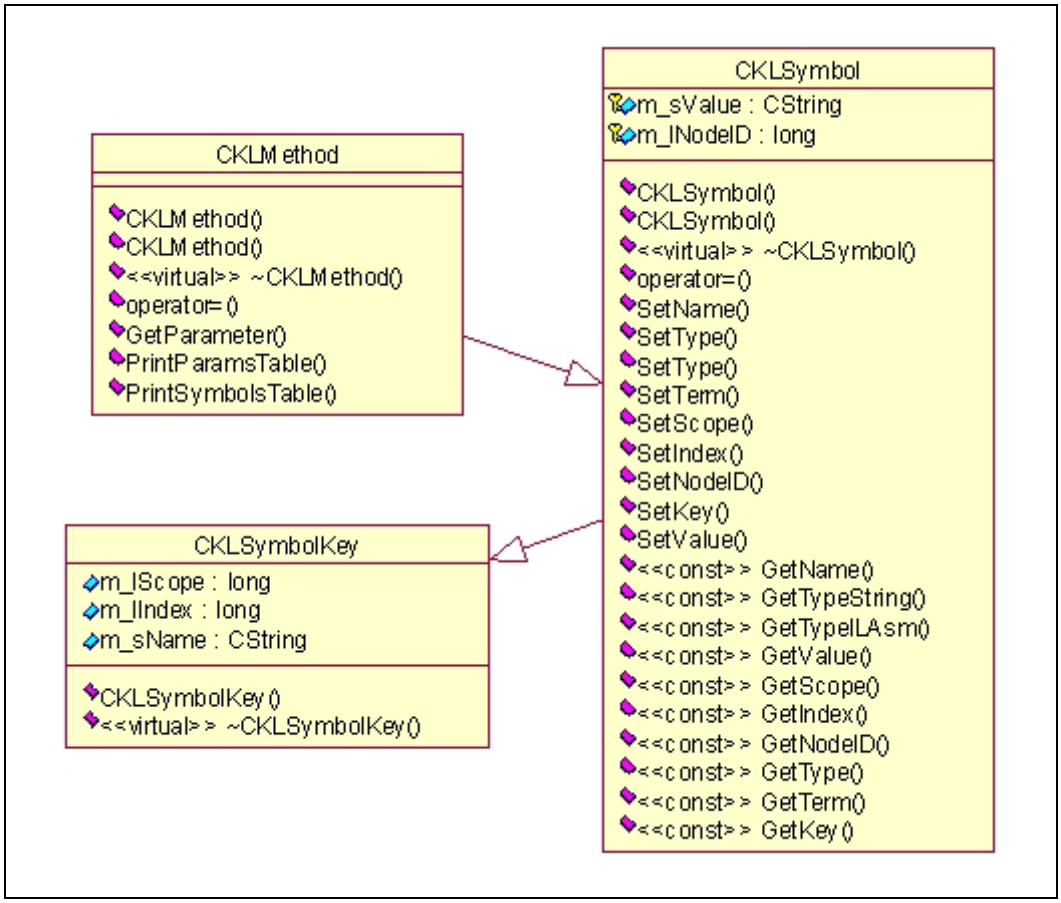


Figura 16 - Classe `CKLMethod`

A classe `CKLMethod` contém duas tabelas de símbolos (Figura 17). Estas tabelas de símbolos são referenciadas através dos atributos:

a) `m_ParamsTable`: contém a tabela de símbolos dos parâmetros do módulo;

b) `m_SymbolsTable`: contém a tabela de símbolos das variáveis do módulo.

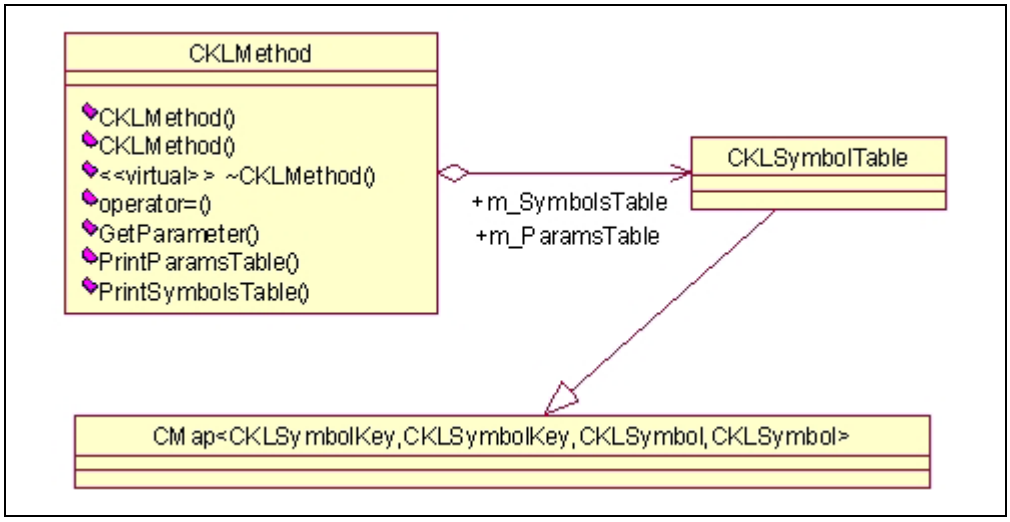


Figura 17 - Associação `CKLMethod` - `CKLSymbolTable`

As tabelas de símbolos são definidas pela classe `CKLSymbolTable` (Figura 18). A classe `CKLSymbolTable` implementa uma tabela *hash* para a classe `CKLSymbol` (Figura 19).

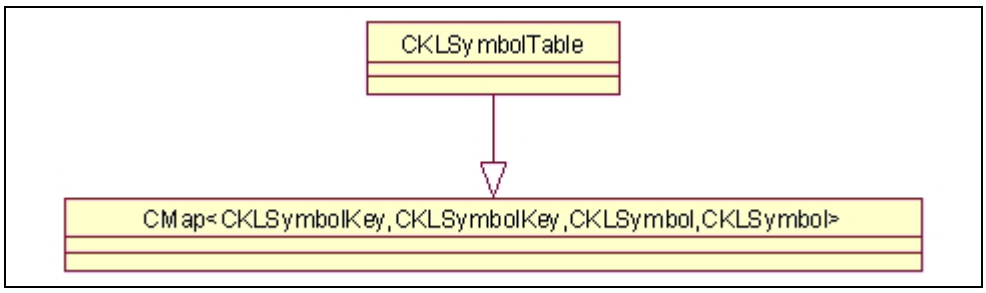


Figura 18 - Classe `CKLSymbolTable`

As chaves para busca nas tabelas *hash* `m_MethodsTable`, `m_ParamsTable` e `m_SymbolsTable` são definidas na classe `CKLSymbolKey` (Figura 20). A chave para essas tabelas são formadas pelo seu escopo (`m_lScope`) mais o seu nome (`m_sName`). Dessa forma, é possível declarar símbolos com o mesmo nome em escopos distintos.

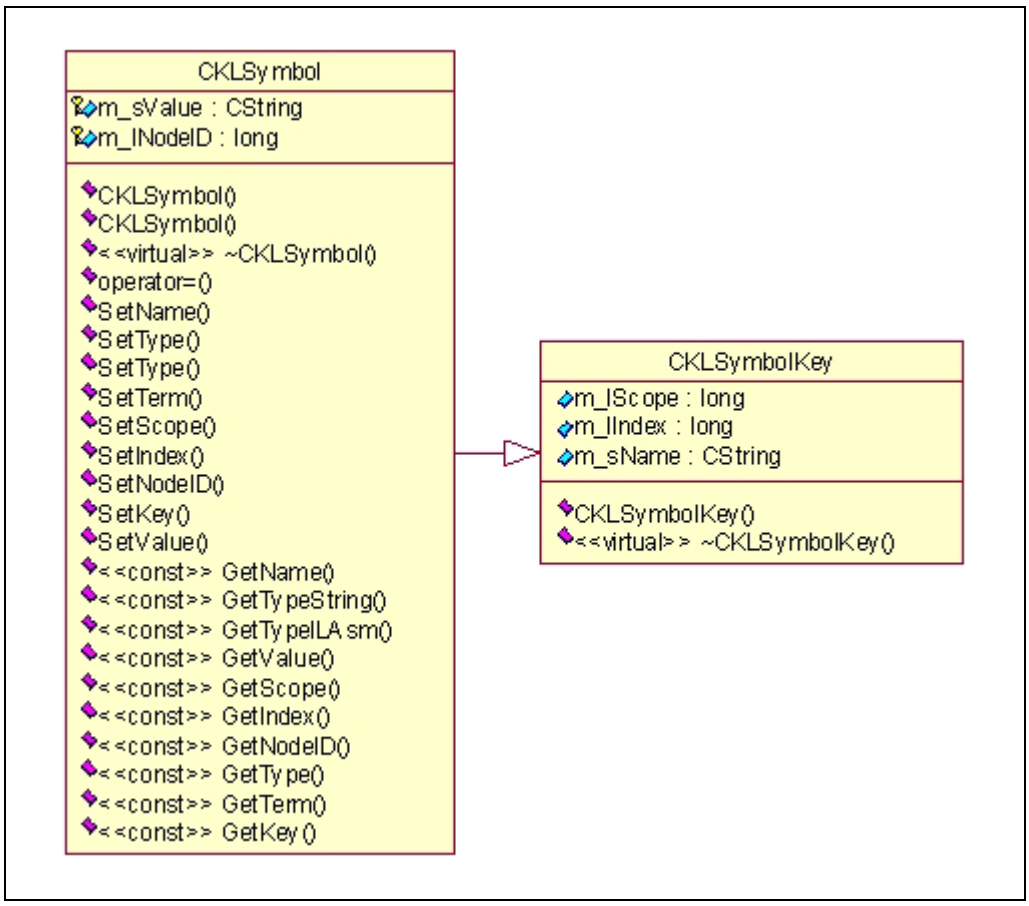


Figura 19 - Classe `CKLSymbol`

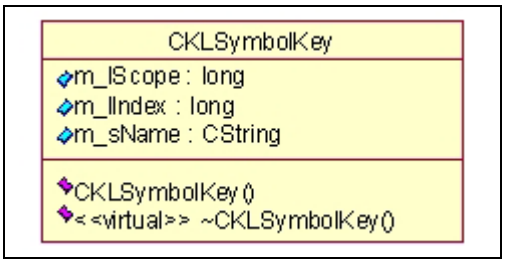


Figura 20- Classe `CKLSymbolKey`

A Figura 21 apresenta o diagrama de atividades do processo de compilação. A seção 3.2.3 exibe os detalhes do processo.

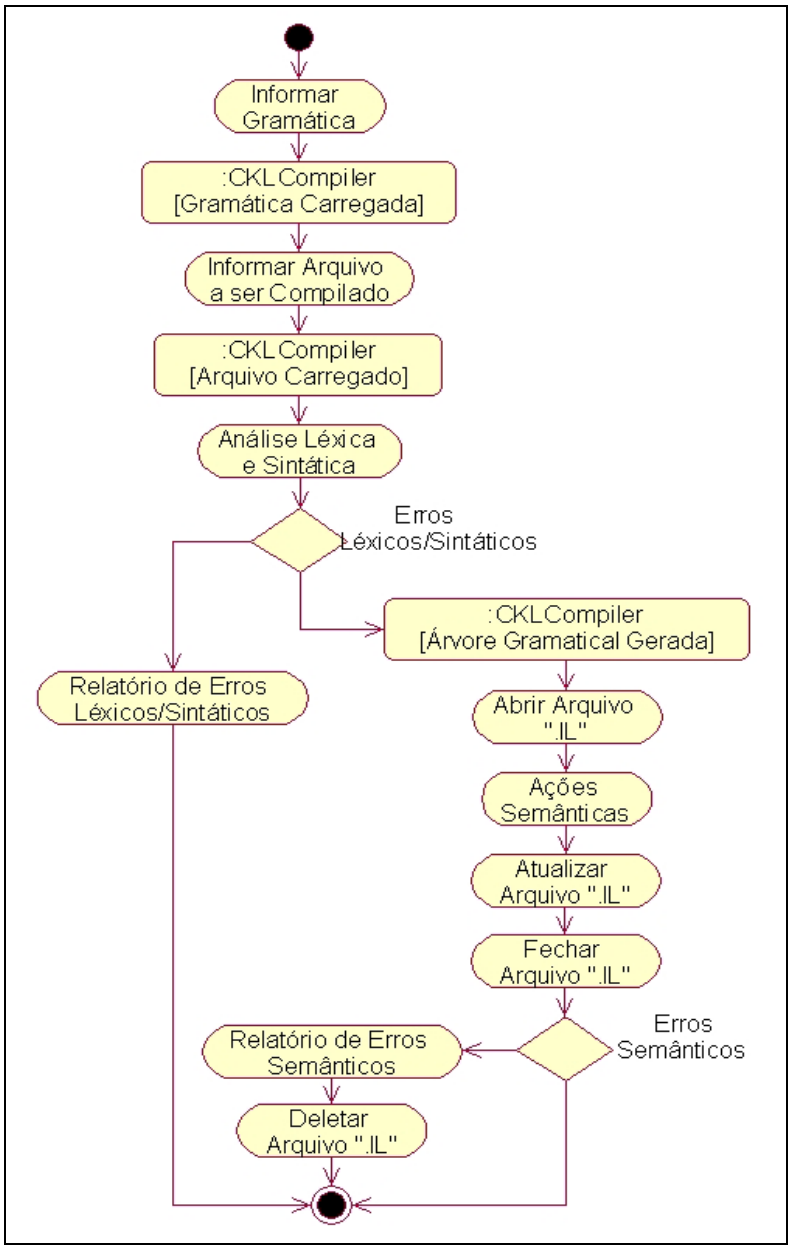


Figura 21- Diagrama de atividades do processo de compilação

3.2.3 IMPLEMENTAÇÃO DO COMPILADOR

No construtor da classe `CKLCompiler`, a gramática da linguagem proposta definida na seção 3.2.1.2 é atribuída para a interface `ParserInterface` através do método `LoadBinaryGrammar()` (Quadro 29).

O processo de compilação é iniciado através do método `Compile()` da classe `CKLCompiler`. O parâmetro deste método é o nome do arquivo a ser compilado.

```
.
.
// inicializa parser interface
m_pParser = GetParserInterface();

// carrega a gramatica a partir do resource
HINSTANCE hInst =
  AfxFindResourceHandle(MAKEINTRESOURCE(IDG_KL_GRAMMAR), "PGGrammar");

HRSRC hRsrc =
  ::FindResource(hInst, MAKEINTRESOURCE(IDG_KL_GRAMMAR), "PGGrammar");

BOOL bLoadGrammar = FALSE;

if (hRsrc)
{
  HGLOBAL hGlobal = LoadResource(hInst, hRsrc);
  if (hGlobal)
  {
    const char* pData = (const char*) LockResource(hGlobal);
    if (pData)
    {
      bLoadGrammar = m_pParser->LoadBinaryGrammar(pData);
    }
    UnlockResource(hGlobal);
    FreeResource(hGlobal);
  }
}
.
.
```

Quadro 29 - Atribuindo a gramática para a interface `ParserInterface`

Em seguida, o nome do arquivo é informado para a interface `ParserInterface` através do método `SetInputFileName()` (Quadro 30). O arquivo informado é carregado pela interface. Através do método `Parse()` (Quadro 30), é iniciado o processo de análise léxica e sintática do texto do arquivo segundo a gramática definida.

```
.
void CKLCompiler::Compile(const char* cFileName)
{
  if (m_pParser)
  {
    cout << endl;
    cout << _T("Compilando ") << cFileName << _T("...") << endl << endl;

    m_pParser->SetInputFilename(cFileName);

    m_pParser->Parse();
.
```

Quadro 30 - Informando o arquivo a ser compilado

Caso o processo de *parsing* ocorra sem nenhum erro, uma árvore gramatical é gerada em memória pela ferramenta. Esta árvore gramatical é percorrida, executando as ações semânticas definidas na seção 3.2.3.1. As ações semânticas realizam a tradução do texto na linguagem proposta para um texto na linguagem MSIL. O texto na linguagem MSIL é atualizado num arquivo com a extensão ".IL", criado durante o processo (Quadro 31).

```
.
.
if (m_pParser->GetNumErrors() == 0)
{
  // abre arquivo saída
  CString sFileOut = cFileName;
  int iPos = sFileOut.ReverseFind('.');
  sFileOut.Delete(iPos, sFileOut.GetLength()-iPos);
  sFileOut += _T(".IL");
  CFileException e;

  if (m_FileOut.Open(sFileOut, CFile::modeCreate | CFile::modeWrite, &e))
  {
    // enumera métodos
    EvaluateMethods();
    // traduz programa
    ParseProgramDecl(m_pParser->GetChild(m_pParser->GetChild(
      m_pParser->GetRoot(), 0), 0));

    // fecha arquivo saída
    m_FileOut.Close();
    .
    .
```

Quadro 31 - Processamento da árvore gramatical gerada

Se existirem erros durante a análise léxica e sintática do texto do arquivo, estes erros são informados conforme o código ilustrado no Quadro 32. Neste caso, nenhuma ação semântica é executada e, portanto, a tradução do texto na linguagem proposta para a linguagem MSIL não ocorre.

```
.
.
for (long lE=0; lE<m_pParser->GetNumErrors(); lE++)
{
  long lError = m_pParser->GetErrorCode(lE);

  if (m_pParser->GetErrorSeverity(lE) == 0)
  {
     cout << _T("erro KL") << lError << _T(": ") <<
      (const char*)m_pParser->GetErrorDescription(lE)<< endl;
  }
  else
  {
     cout << _T("erro KL") << lError << ": linha " <<
       (const char*)m_pParser->GetErrorDetail(lE, _T("LineNumber"))
       << _T(": esperado ") << (const char*)
       m_pParser->GetErrorDetail(lE,_T(("Expecting")) << endl;
  }
}
.
.
```

Quadro 32 - Tratamento de erros

3.2.3.1 AÇÕES SEMÂNTICAS

As ações semânticas são definidas através dos métodos `Parse()` da classe `CKLCompiler`. Para cada regra de produção definida na gramática, existe um método `Parse` correspondente. Cada método `Parse()` tem como parâmetro a identificação do nodo (`lNodeID`) da árvore gramatical gerada, além de outras informações.

O `lNodeID` é utilizado para recuperar as informações e percorrer outros nodos da árvore gramatical . Cada ação semântica pode gerar uma tradução de código que será gravado no arquivo texto ".IL" criado anteriormente na chamada `Compile()`.

Os identificadores das produções são carregados no construtor da classe `CKLCompiler` (Quadro 33). Estes identificadores são usados para definir qual ação semântica será executada.

```
.
.
// carrega identificadores das produções
m_lProgramDeclID   = m_pParser->GetSymbolID("program_declaration");
m_lMethodDeclID    = m_pParser->GetSymbolID("method_declaration");
m_lParamListID     = m_pParser->GetSymbolID("parameter_list");
m_lParamDeclID     = m_pParser->GetSymbolID("parameter");
m_lStmtBlockID     = m_pParser->GetSymbolID("statement_block");
m_lStatementID     = m_pParser->GetSymbolID("statement");
m_lVarDeclID       = m_pParser->GetSymbolID("variable_declaration");
m_lVarInitID       = m_pParser->GetSymbolID("variable_initializer");
m_lIfStmtID        = m_pParser->GetSymbolID("if_statement");
m_lWhileStmtID     = m_pParser->GetSymbolID("while_statement");
m_lReturnStmtID    = m_pParser->GetSymbolID("return_statement");
m_lWriteStmtID     = m_pParser->GetSymbolID("write_statement");
m_lReadStmtID      = m_pParser->GetSymbolID("read_statement");
m_lAssignStmtID    = m_pParser->GetSymbolID("assign_statement");
m_lCallStmtID      = m_pParser->GetSymbolID("call_statement");
m_lExpressionID    = m_pParser->GetSymbolID("expression");
m_lCastingExprID   = m_pParser->GetSymbolID("casting_expression");
m_lArgumentListID  = m_pParser->GetSymbolID("argument_list");
m_lOrTermID        = m_pParser->GetSymbolID("or_term");
m_lAndTermID       = m_pParser->GetSymbolID("and_term");
m_lCmpTermID       = m_pParser->GetSymbolID("cmp_term");
m_lAddTermID       = m_pParser->GetSymbolID("add_term");
m_lMultTermID      = m_pParser->GetSymbolID("mult_term");
m_lFactorID        = m_pParser->GetSymbolID("factor");
m_lReferenceTermID = m_pParser->GetSymbolID("reference_term");
m_lLiteralExprID   = m_pParser->GetSymbolID("literal_expression");
m_lNumLiteralID    = m_pParser->GetSymbolID("numeric_literal");
m_lBoolLiteralID   = m_pParser->GetSymbolID("boolean_literal");
m_lStringID        = m_pParser->GetSymbolID("string");
m_lCharacterID     = m_pParser->GetSymbolID("character");
m_lAssignOpID      = m_pParser->GetSymbolID("assign_op");
m_lOrLevelOpID     = m_pParser->GetSymbolID("or_level_op");
m_lAndLevelOpID    = m_pParser->GetSymbolID("and_level_op");
m_lCmpLevelOpID    = m_pParser->GetSymbolID("cmp_level_op");
m_lAddLevelOpID    = m_pParser->GetSymbolID("add_level_op");
m_lMultLevelOpID   = m_pParser->GetSymbolID("mult_level_op");
m_lUnaryOpID       = m_pParser->GetSymbolID("unary_op");
m_lNegationOpID    = m_pParser->GetSymbolID("negation_op");
.
.
```

Quadro 33 - Identificadores das produções da gramática

3.2.3.2 GERAÇÃO DO CÓDIGO MSIL

Para cada ação semântica definida, é executada a tradução do texto da linguagem proposta para o texto em linguagem MSIL correspondente. O Quadro 34 exibe a ação semântica da regra `program_declaration` definida na gramática da linguagem.

```
void CKLCompiler::ParseProgramDecl(const long& lNodeID)
{
  // nome programa
  m_Program.SetName(m_pParser->GetValue(m_pParser->GetChild(lNodeID, 0)));

  CString sCode;

  // gera codigo do cabeçalho
  sCode.Format("\n.assembly extern mscorlib { }\n.assembly %s { }\n
    .module %s.exe\n\n.class public %s\n{\n",
    m_Program.GetName(), m_Program.GetName(), m_Program.GetName());

  m_FileOut.WriteString(sCode);

  // metodos programa
  for (long lM=0; lM<m_pParser->GetNumChildren(
    m_pParser->GetChild(lNodeID, 1)); lM++)
  {
    ParseMethodDecl(m_pParser->GetChild(
      m_pParser->GetChild(lNodeID, 1), lM));
  }
  m_FileOut.WriteString("}\n");
}
```

Quadro 34 - Ação semântica para a regra de produção `program_declaration`

A geração de código é realizada nodo a nodo. A ação semântica `ParseProgramDecl()`, executa a ação semântica `ParseMethodDecl()` para cada módulo definido. A ação semântica `ParseMethodDecl()` irá executar outras ações semânticas como a declaração de variáveis, comandos de seleção, comandos de repetição e assim sucessivamente, até que toda a árvore gramatical seja percorrida e analisada. O apêndice A exibe a implementação de algumas ações semânticas da linguagem proposta.

3.2.3.3 MONTADOR ILASM

O montador *Intermediate Language Assembler* (ILAsm) é utilizado para compilar o arquivo ".IL" produzido pelo compilador da linguagem proposta em um arquivo ".EXE" para ser executado na plataforma Microsoft .NET.

Caso alguns erros léxicos, sintáticos ou semânticos não tenham sido identificados pelo compilador da linguagem proposta, o montador ILAsm irá acusar erros na montagem do arquivo ".EXE". Esta é uma situação indesejável, uma vez que o usuário não está familiarizado com as instruções do MSIL e dificilmente conseguirá resolver os erros encontrados.

3.3 AMBIENTE DE PROGRAMAÇÃO

O ambiente de programação tem por objetivo facilitar o desenvolvimento de programas na linguagem proposta. Ele oferece suporte a um editor de textos, opções para configurar e executar ferramentas de compilação (compilador e montador) e também permite a execução dos programas criados na linguagem proposta. A Figura 22 exibe o diagrama de casos de uso do ambiente de programação.

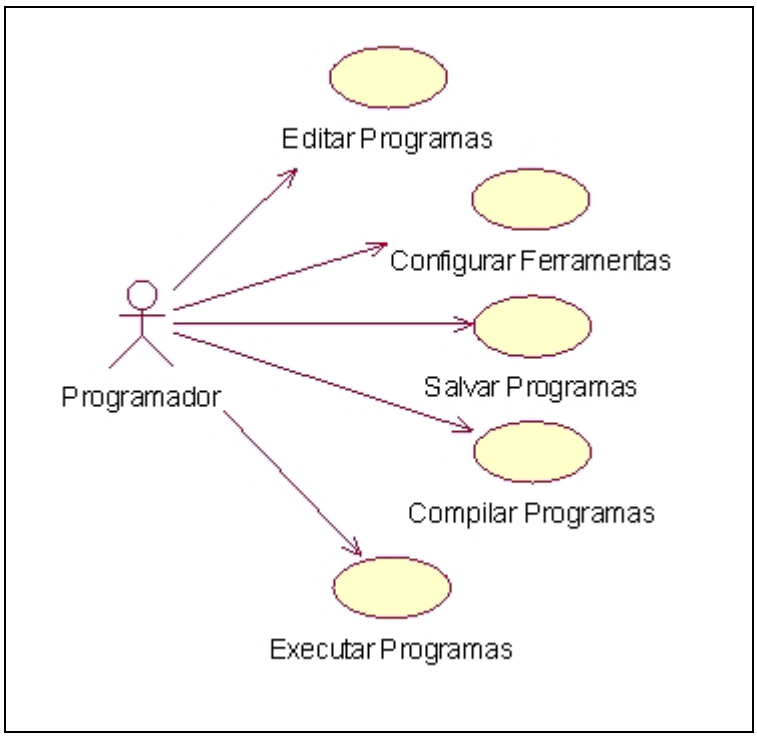


Figura 22 - Casos de uso do ambiente de programação

A Tabela 9 descreve os casos de uso exibidos na Figura 22.

Tabela 9 - Descrição dos casos de uso do ambiente de programação

Caso de Uso	**Descrição**
Editar programas	O programador cria um novo programa fonte através do comando Novo ou o programador abre um programa fonte através do comando Abrir. O programador faz a edição do programa fonte.
Configurar ferramentas	O programador configura as pastas onde estão localizadas as ferramentas para compilação: o compilador e o montador.
Salvar programas	Através do comando Salvar o programador salva as alterações efetuadas no programa fonte.
Compilar programas	O programador compila o programa fonte através do comando Compilar '.KL' no menu Compilar e monta um arquivo executável através do comando Compilar '.IL'
Executar programas	O programador executa o programa compilado através do comando Executar no menu Compilar.

A Figura 23 exibe a interface do ambiente de programação da linguagem proposta. A interface contém um editor de textos, um painel para visualização dos processos de compilação e montagem, uma barra de menus, uma barra de ferramentas com os principais comandos e uma barra de *status* que exibe a linha e coluna do cursor, o estado do arquivo (somente leitura ou não), o modo de edição e o estado do teclado (*CAPS LOCK* e *NUM LOCK*).

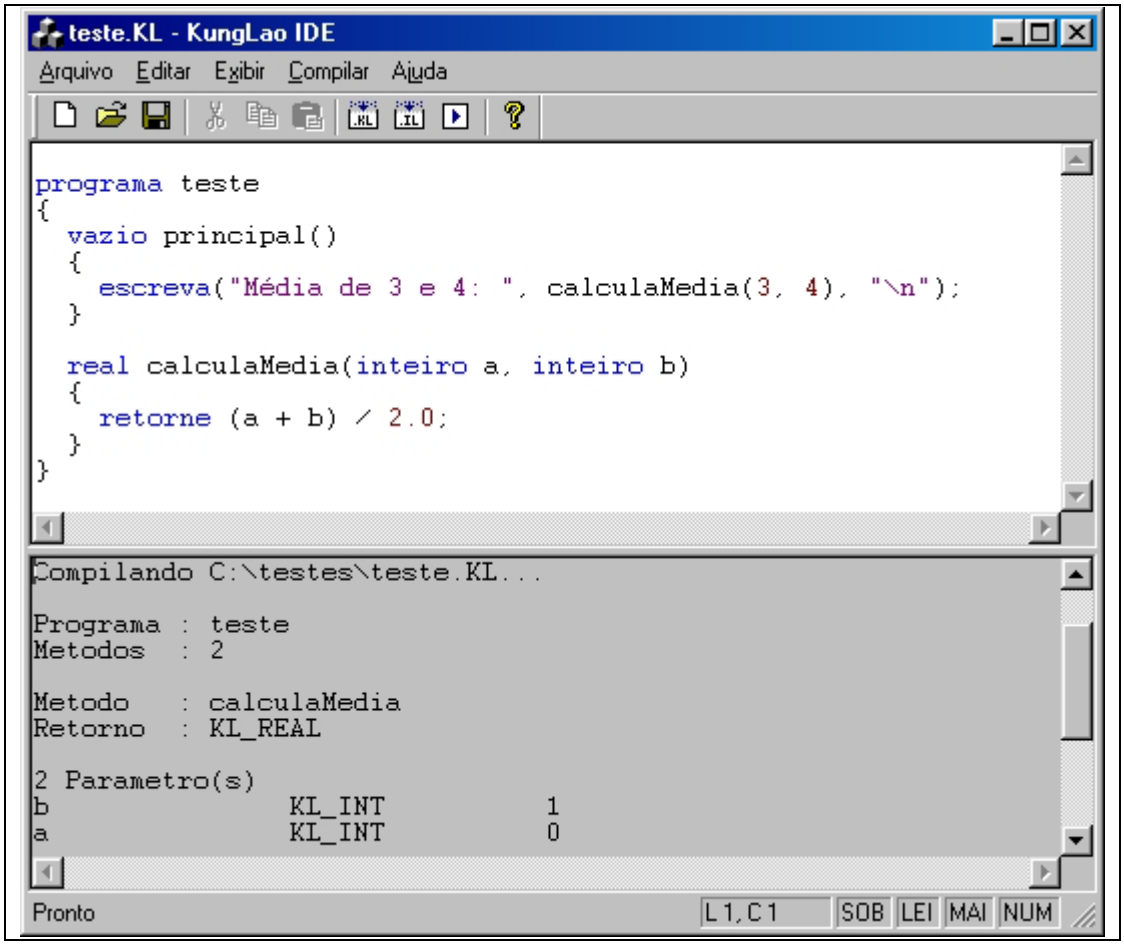


Figura 23- Ambiente de programação da linguagem proposta

A Figura 24 exibe a tela de configuração das ferramentas (compilador e montador).

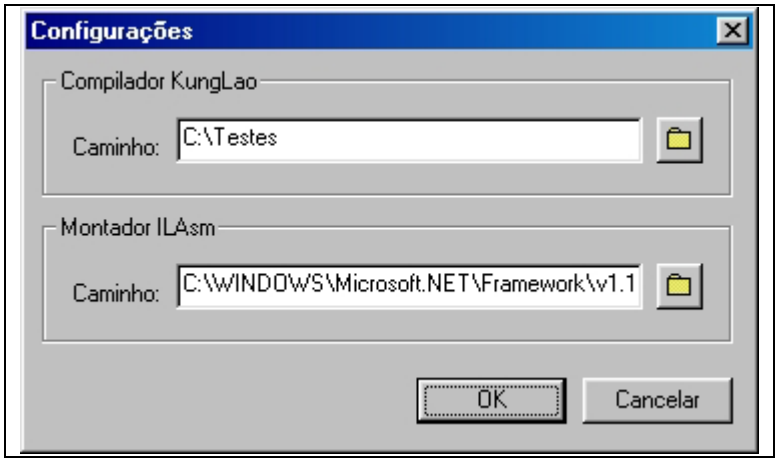


Figura 24 - Configuração das ferramentas de compilação

Na tela de configurações, o usuário configura as pastas onde estão localizadas as ferramentas para compilação. Caso as ferramentas não sejam encontradas nas pastas informadas, uma mensagem de erro é exibida e o usuário ficará impossibilitado de compilar, montar e executar os programas editados.

O ambiente de programação (Figura 25) foi implementado utilizando a ferramenta Microsoft Visual C++ 6.0 com a biblioteca de classes da MFC. Para a implementação do ambiente, foram utilizadas várias classes disponibilizadas por desenvolvedores no *site Code Project* (CODE PROJECT, 2004).

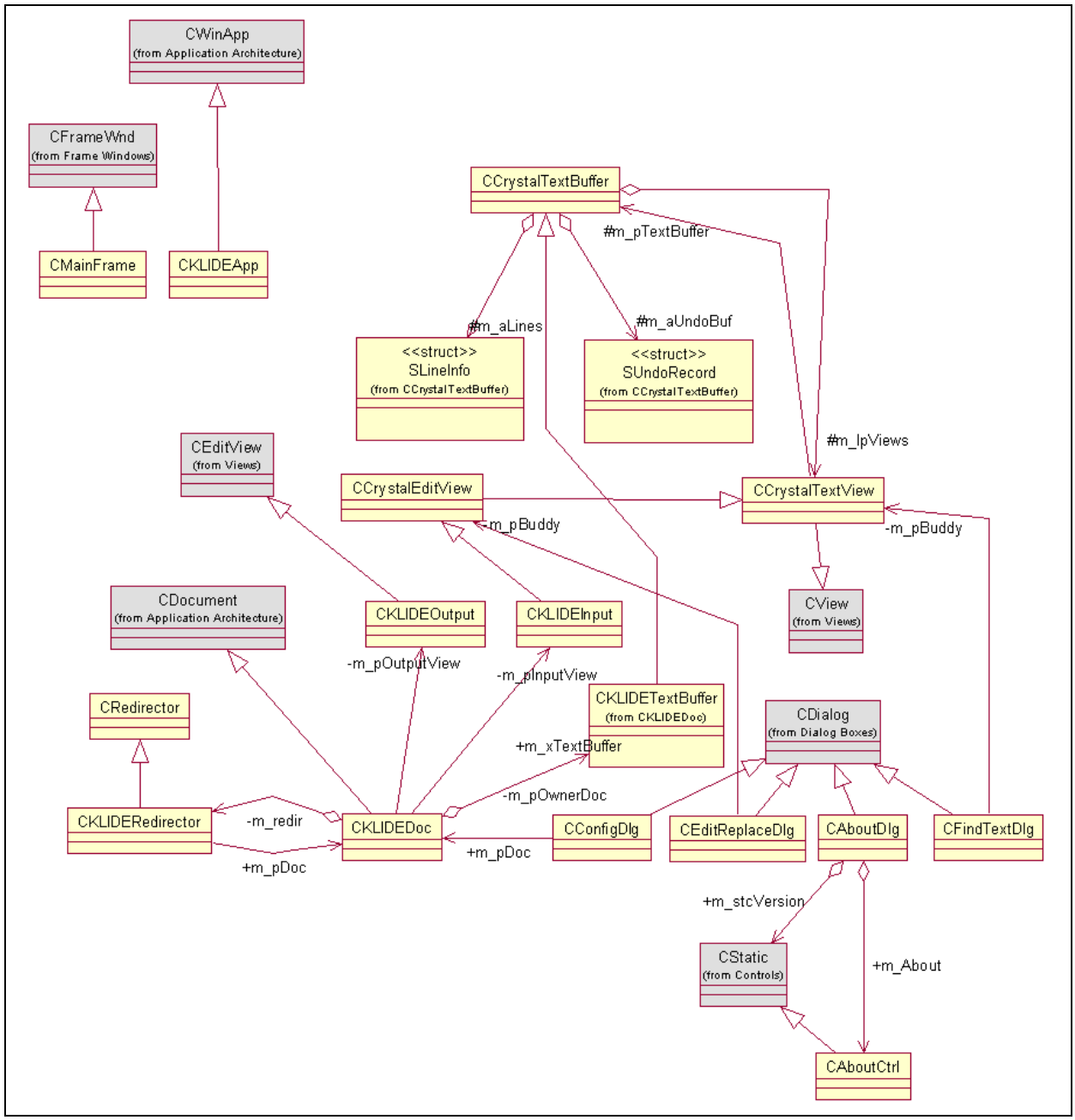


Figura 25 - Diagrama de classes do ambiente de programação

3.4 RESULTADOS E DISCUSSÕES

Existem várias limitações na linguagem desenvolvida durante o trabalho. Entre estas limitações, estão o uso de vetores e a passagem de parâmetros por referência nos módulos. Outra limitação é a indisponibilidade de comandos para fazer alocação dinâmica de memória. Além disso, o tipo `cadeia` somente pode ser utilizado com o operador de atribuição (=).

Como limitações do compilador, destaca-se o tratamento dos erros léxicos, sintáticos e semânticos. Nos erros léxicos e sintáticos, o compilador somente exibe a linha onde ocorreu o erro e como mensagem, exibe a próxima produção da gramática esperada. Esta mensagem de erro é gerada pela ferramenta *ProGrammar*. As mensagens nestes casos devem ser refinadas a fim de descrever os erros com maiores detalhes para o programador. A ferramenta *ProGrammar* disponibiliza comandos para o tratamento de erros, mas os mesmos não foram estudados.

Nos erros semânticos, a principal limitação está na verificação dos tipos nas expressões. O compilador sempre converte os operandos para o tipo mais abrangente, mas não emite nenhuma mensagem de erro ao programador quando uma conversão dos tipos pode levar à perda de dados. A única mensagem de erro sobre conversão de tipos ocorre quando da utilização do tipo `cadeia` nas expressões.

A linguagem e o compilador desenvolvidos durante o trabalho podem ser empregados em disciplinas como a introdução à programação, auxiliando os estudantes destas disciplinas.

4 CONCLUSÕES

O objetivo principal do trabalho, construir um compilador para uma linguagem de programação simplificada e em português foi alcançado. A linguagem foi especificada utilizando o método formal BNF. O uso da BNF para definição da sintaxe e orientação para a definição da semântica da linguagem foi indispensável na criação da gramática para a implementação do compilador.

A utilização da ferramenta *ProGrammar* para a construção do compilador foi de extrema importância para o trabalho, pois, como foi uma ferramenta de fácil aprendizado, acelerou e simplificou o processo de desenvolvimento do compilador. As utilizações da ferramenta Microsoft Visual C++ 6.0, Rational Rose Enterprise Edition e das classes da MFC também auxiliaram consideravelmente no desenvolvimento do trabalho.

Para a implementação das ações semânticas foi necessário um estudo detalhado das instruções do MSIL e da plataforma Microsoft .NET. Isto foi possível graças à bibliografia e ao trabalho de Gough (2002), que também projetou uma linguagem de programação tendo como alvo a plataforma Microsoft .NET. A linguagem projetada por Gough (2002) é bem mais complexa, contendo diversas construções e suporte a orientação a objetos.

A principal contribuição deste trabalho é o estudo do MSIL e da plataforma Microsoft .NET. A partir das informações disponibilizadas neste trabalho, outros compiladores poderão ser projetados. Além disso, compiladores existentes poderão ter como código alvo o MSIL ao invés de código *assembler*, o qual é mais complexo.

Como possíveis extensões para o trabalho, destacam-se melhorias na linguagem de programação proposta, tais como:

a) suporte a orientação a objetos;
b) passagem de parâmetros por referência nos módulos;
c) implementação de vetores;
d) alocação dinâmica de memória.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V; SETHI, Ravi; ULMAN, Jeffrey D. **Compiladores**: princípios, técnicas e ferramentas. Tradução Daniel de Ariosto Pinto. Rio de Janeiro: Livros Técnicos e Científicos, 1995.

BORLAND SOFTWARE CORPORATION. **C#Builder for the Microsoft .NET framework:** Borland .NET solutions, [s.l.], 2004. Disponível em: <http://www.borland.com/csharpbuilder/>. Acesso em: 4 abr. 2004.

CHERRY, Michael; DEMICHILLIE, Greg. **A plataforma de desenvolvimento .NET**. [S.l.]: Microsoft, 2002. 24 p.

CODE PROJECT. **The code project - free source code and tutorials**, [s.l], 2004. Disponível em: <http://www.codeproject.com/>. Acesso em: 20 abr. 2004.

ELLIS, Margaret A, STROUSTRUP, Bjarne. **C++**: manual de referência comentado. Tradução PubliCare Serviços de Informática. Rio de Janeiro: Campus, 1993.

GERMAN NATIONAL RESEARCH CENTER FOR INFORMATION TECHNOLOGY. **Catalog of compiler construction tools**, [s.l.], 2003. Disponível em: <http://www.first.gmd.de/cogent/catalog/>. Acesso em: 13 abr. 2004.

GOUGH, Kevin John. **Compiling for the .NET common language runtime (CLR)**. Upper Saddle River: Prentice Hall, 2002.

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

MICROSOFT CORPORATION. **Welcome to the MSDN library**, [s.l], 2004. Disponível em: <http://msdn.microsoft.com/library/default.asp>. Acesso em: 10 abr. 2004.

NORKEN TECHNOLOGIES. **Norken Technologies, inc.**, [s.l], 2004. Disponível em: <http://www.programmar.com/>. Acesso em: 20 fev. 2004.

PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. **Implementação de linguagens de programação**: compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

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

SERGE, Lidin. **Inside Microsoft .NET IL assembler**. Redmond: Microsoft Press, 2002.

SUN MICROSYSTEMS. **Java 2 platform, enterprise edition (J2EE)**, [s.l], 2004. Disponível em: <http://java.sun.com/j2ee/>. Acesso em: 10 abr. 2004.

APÊNDICE A – Ações semânticas

Neste apêndice são ilustradas algumas ações semânticas do compilador para a linguagem proposta.

```
void CKLCompiler::ParseMethodDecl(const long& lNodeID)
{
  CKLMethod Method;

  // tipo retorno metodo
  Method.SetType(m_pParser->GetValue(m_pParser->GetChild(lNodeID, 0)));
  // nome metodo
  Method.SetName(m_pParser->GetValue(m_pParser->GetChild(lNodeID, 1)));

  Method = m_Program.m_MethodsTable[Method.GetKey()];

  CString sCode;

  // gera codigo de declaracao de metodo
  sCode.Format("\t.method public static %s %s(", Method.GetTypeILAsm(), Method.GetName());

  m_FileOut.WriteString(sCode);

  long lNodeSymbolID  = m_pParser->GetNodeSymbolID(m_pParser->GetChild(lNodeID, 2));

  long lChildStmtNode = m_pParser->GetChild(lNodeID, 2);

  // parametros metodo
  if (lNodeSymbolID == m_lParamListID)
  {
    long lNumParams = m_pParser->GetNumChildren(m_pParser->GetChild(lNodeID, 2));

    for (long lP=0; lP<lNumParams; lP++)
    {
       ParseParamDecl(Method, m_pParser->GetChild(m_pParser->GetChild(lNodeID, 2), lP));

      if (lP < lNumParams-1)
      {
        m_FileOut.WriteString(",");
      }
    }

   // bloco comandos metodo
    lChildStmtNode = m_pParser->GetChild(lNodeID, 3);
   }

  m_FileOut.WriteString(")\n");

  // metodo sem parametros, traduz bloco comandos metodo
  ParseStmtBlock(Method, lChildStmtNode);

  // atualiza tabela metodos
  m_Program.m_MethodsTable[Method.GetKey()] = Method;
}
```

Quadro 35 - Ação semântica da declaração de métodos

```
void CKLCompiler::ParseParamDecl(CKLMethod& Method, const long& lNodeID)
{
  CKLSymbol Symbol;

  // nome parametro
  Symbol.SetName(m_pParser->GetValue(m_pParser->GetChild(lNodeID, 1)));

   Symbol = Method.m_ParamsTable[Symbol.GetKey()];

   CString sCode;

   // gera codigo declaracao parametro
   sCode.Format("%s A_%d", Symbol.GetTypeILAsm(), Symbol.GetIndex());

   m_FileOut.WriteString(sCode);
}
```

Quadro 36 - Ação semântica da declaração de parâmetros de um método

```
void CKLCompiler::ParseVarDecl(CKLMethod& Method, const long& lNodeID)
{
  CKLSymbol Symbol;

  // escopo variavel
  Symbol.SetScope(m_lScope);
  // tipo variavel
  Symbol.SetType(m_pParser->GetValue(m_pParser->GetChild(lNodeID, 0)));
  // nome variavel
  Symbol.SetName(m_pParser->GetValue(m_pParser->GetChild(lNodeID, 1)));
  // tipo termo
  Symbol.SetTerm(KL_VARIABLE);
  // indice variavel
   Symbol.SetIndex(Method.m_SymbolsTable.GetCount());

   CString sCode;

   // gera codigo para declaracao variavel
   sCode.Format("\t\t.locals (%s V_%d)\n", Symbol.GetTypeILAsm(), Symbol.GetIndex());

   m_FileOut.WriteString(sCode);

  // adiciona tabela simbolos
  Method.m_SymbolsTable[Symbol.GetKey()] = Symbol;

   // inicializacao variavel
   long lNodeSymbolID = m_pParser->GetNodeSymbolID(m_pParser->GetChild(lNodeID, 2));

   if (lNodeSymbolID == m_lVarInitID)
   {
     ParseVarInit(Symbol, Method, m_pParser->GetChild(lNodeID, 2));
   }
}
```

Quadro 37 - Ação semântica da declaração de variáveis

```
void CKLCompiler::ParseIfStmt(CKLMethod& Method, const long& lNodeID)
{
  // pega numero de comandos, para definir tipo de comando de selecao
  long lC = m_pParser->GetNumChildren(lNodeID);

  // labels de desvios
  long lBranchElse = 0;
  long lBranchEnd  = 0;

  if (lC == 2)
  {
    // comando de selecao nao possui clasula senao
    lBranchEnd  = m_lBranch;
    m_lBranch++;
  }
  else if (lC == 3)
  {
    // comando de selecao possui clausula senao
    lBranchElse = m_lBranch;
    lBranchEnd  = ++m_lBranch;

    m_lBranch++;
  }

  // avalia expressao logica
  CKLSymbol   sE;
  CKLEvalInfo EvalInfo;

  EvaluateExpression(Method, m_pParser->GetChild(lNodeID, 0), sE, EvalInfo);
  ParseExpression(Method, m_pParser->GetChild(lNodeID, 0), EvalInfo);

  CString sCode;

  if (lC == 2)
  {
    // instrucao de desvio para final do comando
    sCode.Format("\t\tbrfalse\tlb%06x\n", lBranchEnd);

    m_FileOut.WriteString(sCode);

    // avalia comandos
    ParseStatement(Method, m_pParser->GetChild(lNodeID, 1));

    // label para final do comando
    sCode.Format("\tlb%06x:\n", lBranchEnd);

    m_FileOut.WriteString(sCode);
  }
  else if (lC == 3)
  {
    // instrucao de desvio para senao do comando
    sCode.Format("\t\tbrfalse\tlb%06x\n", lBranchElse);

    m_FileOut.WriteString(sCode);

    // avalia comandos se
    ParseStatement(Method, m_pParser->GetChild(lNodeID, 1));

    // instrucao de desvio para final, label para senao do comando
    sCode.Format("\t\tbr\tlb%06x\n\tlb%06x:\n", lBranchEnd, lBranchElse);

    m_FileOut.WriteString(sCode);

    // avalia comandos senao
    ParseStatement(Method, m_pParser->GetChild(lNodeID, 2));

    // label para final do comando
    sCode.Format("\tlb%06x:\n", lBranchEnd);

    m_FileOut.WriteString(sCode);
  }
}
```

Quadro 38 - Ação semântica do comando de seleção `se`

```
void CKLCompiler::ParseWhileStmt(CKLMethod& Method, const long& lNodeID)
{
  // labels para desvios
  long lBranchLoop = 0;
  long lBranchEnd  = 0;

  lBranchLoop  = m_lBranch;
  lBranchEnd   = ++m_lBranch;

  m_lBranch++;

  CString sCode;

  // label repeticao
  sCode.Format("\tlb%06x:\n", lBranchLoop);

  m_FileOut.WriteString(sCode);

  // avalia expressao logica
  CKLSymbol sE;
  CKLEvalInfo EvalInfo;

  EvaluateExpression(Method, m_pParser->GetChild(lNodeID, 0), sE, EvalInfo);
  ParseExpression(Method, m_pParser->GetChild(lNodeID, 0), EvalInfo);

  // instrucao de desvio para final
  sCode.Format("\t\tbrfalse\tlb%06x\n", lBranchEnd);

  m_FileOut.WriteString(sCode);

  // avalia comandos
  ParseStatement(Method, m_pParser->GetChild(lNodeID, 1));

  // instrucao de desvio para repeticao
  sCode.Format("\t\tbr\tlb%06x\n", lBranchLoop);

  m_FileOut.WriteString(sCode);

  // label para final do comando
  sCode.Format("\tlb%06x:\n", lBranchEnd);

  m_FileOut.WriteString(sCode);
}
```

Quadro 39 - Ação semântica do comando de repetição `enquanto`

ANEXO A – Conjunto de instruções do Microsoft Intermediate Language

Neste anexo são apresentadas as instruções do MSIL. A Tabela 10 mostra os tipos de dados que podem estar na pilha de avaliação. A Tabela 11 mostra os tipos dos parâmetros das instruções.

Tabela 10 - Tipos de dados da pilha de avaliação

Tipo de Dado	**Descrição**
int32	Inteiro de 32 *bits* com sinal
int64	Inteiro de 64 *bits* com sinal
Float	Ponto flutuante de 80 *bits*
o	Referência para objeto
*	Tipo não especificado

Fonte: Serge (2002, p. 421)

Tabela 11 - Tipos dos parâmetros das instruções

Tipo de Parâmetro	**Descrição**
int8	Inteiro de 8 *bits* com sinal
int32	Inteiro de 32 *bits* com sinal
uint8	Inteiro de 8 *bits* sem sinal
uint32	Inteiro de 32 *bits* sem sinal
float32	Ponto flutuante de 32 *bits*
float64	Ponto flutuante de 64 *bits*
<Method>	*Token MethodDef* ou *MemberRef*
<Field>	*Token FieldDef* ou *MemberRef*
<Type>	*Token TypeDef, TypeRef* ou *TypeSpec*
<Signature>	*Token StandAloneSig*
<String>	Cadeia definida pelo usuário

Fonte: Serge (2002, p. 421)

A Tabela 12 mostra todo o conjunto de instruções do MSIL, contendo o código de operação (*opcode*), o nome da instrução, os parâmetros da instrução, o tipo de dado retirado da pilha (*pop*) e o tipo do dado inserido na pilha de avaliação (*push*) durante a execução das instruções.

Tabela 12 - Instruções do MSIL

Opcode	**Nome**	**Parâmetro(s)**	***Pop***	***Push***
00	nop	-	-	-
01	break	-	-	-
02	ldarg.0	-	-	*
03	ldarg.1	-	-	*
04	ldarg.2	-	-	*
05	ldarg.3	-	-	*
06	ldloc.0	-	-	*
07	ldloc.1	-	-	*
08	ldloc.2	-	-	*
09	ldloc.3	-	-	*
0A	stloc.0	-	*	-
0B	stloc.1	-	*	-
0C	stloc.2	-	*	-
0D	stloc.3	-	*	-
0E	ldarg.s	uint8	-	*
0F	ldarga.s	uint8	-	&
10	starg.s	uint8	*	-
11	ldloc.s	uint8	-	*
12	ldloca.s	uint8	-	&
13	stloc.s	uint8	*	-
14	ldnull	-	-	&=0
15	ldc.i4.m1ldc.i4.M1	-	-	int32=-1
16	ldc.i4.0	-	-	int32=0
17	ldc.i4.1	-	-	int32=1
18	ldc.i4.2	-	-	int32=2
19	ldc.i4.3	-	-	int32=3
1A	ldc.i4.4	-	-	int32=4
1B	ldc.i4.5	-	-	int32=5
1C	ldc.i4.6	-	-	int32=6
1D	ldc.i4.7	-	-	int32=7
1E	ldc.i4.8	-	-	int32=8
1F	ldc.i4.s	int8	-	int32
20	ldc.i4	int32	-	int32
21	ldc.i8	int64	-	int64
22	ldc.r4	float32	-	Float
23	ldc.r8	float64	-	Float
25	dup	-	*	*,*
26	pop	-	*	-
27	jmp	<Method>	-	-
28	call	<Method>	N arguments	Ret.value
29	calli	<Signature>	N arguments	Ret.value
2A	ret	-	*	-
2B	br.s	int8	-	-
2C	brfalse.sbrnull.sbrzero.s	int8	int32	-
2D	brtrue.sbrinst.s	int8	int32	-
2E	beq.s	int8	*,*	-

Opcode	**Nome**	**Parâmetro(s)**	***Pop***	***Push***
2F	bge.s	int8	*,*	-
30	bgt.s	int8	*,*	-
31	ble.s	int8	*,*	-
32	blt.s	int8	*,*	-
33	bne.un.s	int8	*,*	-
34	bge.un.s	int8	*,*	-
35	bgt.un.s	int8	*,*	-
36	ble.un.s	int8	*,*	-
37	blt.un.s	int8	*,*	-
38	br	int32	-	-
39	brfalsebrnullbrzero	int32	int32	-
3A	brtruebrinst	int32	int32	-
3B	beq	int32	*,*	-
3C	bge	int32	*,*	-
3D	bgt	int32	*,*	-
3E	ble	int32	*,*	-
3F	blt	int32	*,*	-
40	bne.un	int32	*,*	-
41	bge.un	int32	*,*	-
42	bgt.un	int32	*,*	-
43	ble.un	int32	*,*	-
44	blt.un	int32	*,*	-
45	switch	(uint32=N) + N(int32)	*,*	-
46	ldind.i1	-	&	int32
47	ldind.u1	-	&	int32
48	ldind.i2	-	&	int32
49	ldind.u2	-	&	int32
4A	ldind.i4	-	&	int32
4B	ldind.u4	-	&	int32
4C	ldind.i8ldind.u8	-	&	int64
4D	ldind.i	-	&	int32
4E	ldind.r4	-	&	Float
4F	ldind.r8	-	&	Float
50	ldind.ref	-	&	&
51	stind.ref	-	&,&	-
52	stind.i1	-	int32,&	-
53	stind.i2	-	int32,&	-
54	stind.i4	-	int32,&	-
55	stind.i8	-	int32,&	-
56	stind.r4	-	Float,&	-
57	stind.r8	-	Float,&	-
58	add	-	*,*	*
59	sub	-	*,*	*
5A	mul	-	*,*	*
5B	div	-	*,*	*
5C	div.un	-	*,*	*

Opcode	**Nome**	**Parâmetro(s)**	***Pop***	***Push***
5D	rem	-	*,*	*
5E	rem.un	-	*,*	*
5F	and	-	*,*	*
60	or	-	*,*	*
61	xor	-	*,*	*
62	shl	-	*,*	*
63	shr	-	*,*	*
64	shr.un	-	*,*	*
65	neg	-	*	*
66	not	-	*	*
67	conv.i1	-	*	int32
68	conv.i2	-	*	int32
69	conv.i4	-	*	int32
6A	conv.i8	-	*	int64
6B	conv.r4	-	*	Float
6C	conv.r8	-	*	Float
6D	conv.u4	-	*	int32
6E	conv.u8	-	*	int64
6F	callvirt	<Method>	N arguments	Ret.value
70	cpobj	<Type>	&,&	-
71	ldobj	<Type>	&	*
72	ldstr	<String>	-	o
73	newobj	<Method>	N arguments	o
74	castclass	<Type>	o	o
75	isinst	<Type>	o	int32
76	conv.r.un	-	*	Float
79	unbox	<Type>	o	&
7A	throw	-	o	-
7B	ldfld	<Field>	o/&	*
7C	ldflda	<Field>	o/&	&
7D	stfld	<Field>	o/&,*	-
7E	ldsfld	<Field>	-	*
7F	ldsflda	<Field>	-	&
80	stsfld	<Field>	*	-
81	stobj	<Type>	&,*	-
82	conv.ovf.i1.un	-	*	int32
83	conv.ovf.i2.un	-	*	int32
84	conv.ovf.i4.un	-	*	int32
85	conv.ovf.i8.un	-	*	int64
86	conv.ovf.u1.un	-	*	int32
87	conv.ovf.u2.un	-	*	int32
88	conv.ovf.u4.un	-	*	int32
89	conv.ovf.u8.un	-	*	int64
8A	conv.ovf.i.un	-	*	int32
8B	conv.ovf.u.un	-	*	int64
8C	box	<Type>	*	o
8D	newarr	<Type>	int32	o

Opcode	**Nome**	**Parâmetro(s)**	***Pop***	***Push***
8E	ldlen	-	o	int32
8F	ldelema	<Type>	int32,o	&
90	ldelem.i1	-	int32,o	int32
91	ldelem.u1	-	int32,o	int32
92	ldelem.i2	-	int32,o	int32
93	ldelem.u2	-	int32,o	int32
94	ldelem.i4	-	int32,o	int32
95	ldelem.u4	-	int32,o	int32
96	ldelem.i8ldelem.u8	-	int32,o	int64
97	ldelem.i	-	int32,o	int32
98	ldelem.r4	-	int32,o	Float
99	ldelem.r8	-	int32,o	Float
9A	ldelem.ref	-	int32,o	o/&
9B	stelem.i	-	int32,int32,o	-
9C	stelem.i1	-	int32,int32,o	-
9D	stelem.i2	-	int32,int32,o	-
9E	stelem.i4	-	int32,int32,o	-
9F	stelem.i8	-	int64,int32,o	-
A0	stelem.r4	-	Float,int32,o	-
A1	stelem.r8	-	Float,int32,o	-
A2	stelem.ref	-	o/&,int32,o	-
B3	conv.ovf.i1	-	*	int32
B4	conv.ovf.u1	-	*	int32
B5	conv.ovf.i2	-	*	int32
B6	conv.ovf.u2	-	*	int32
B7	conv.ovf.i4	-	*	int32
B8	conv.ovf.u4	-	*	int32
B9	conv.ovf.i8	-	*	int64
BA	conv.ovf.u8	-	*	int64
C2	refanyval	<Type>	*	&
C3	ckfinite	-	*	Float
C6	mkrefany	<Type>	&	*
D0	ldtoken	<Type>/ <Field>/ <Method>	-	&
D1	conv.u2	-	*	int32
D2	conv.u1	-	*	int32
D3	conv.i	-	*	int32
D4	conv.ovf.i	-	*	int32
D5	conv.ovf.u	-	*	int32
D6	add.ovf	-	*,*	*
D7	add.ovf.un	-	*,*	*
D8	mul.ovf	-	*,*	*
D9	mul.ovf.un	-	*,*	*
DA	sub.ovf	-	*,*	*
DB	sub.ovf.un	-	*,*	*
DC	endfinallyendfault	-	-	-

Opcode	**Nome**	**Parâmetro(s)**	***Pop***	***Push***
DD	leave	int32	-	-
DE	leave.s	int8	-	-
DF	stind.i	-	int32,&	-
E0	conv.u	-	*	int32
FE 00	arglist	-	*	&
FE 01	ceq	-	*,*	int32
FE 02	cgt	-	*,*	int32
FE 03	cgt.un	-	*,*	int32
FE 04	clt	-	*,*	int32
FE 05	clt.un	-	*,*	int32
FE 06	ldftn	<Method>	-	&
FE 07	ldvirtftn	<Method>	o	&
FE 09	ldarg	uint32	-	*
FE 0A	ldarga	uint32	-	&
FE 0B	starg	uint32	*	-
FE 0C	ldloc	uint32	-	*
FE 0D	ldloca	uint32	-	&
FE 0E	stloc	uint32	*	-
FE 0F	localloc	-	int32	&
FE 11	endfilter	-	int32	-
FE 12	unaligned.	uint8	-	-
FE 13	volatile.	-	-	-
FE 14	tail.	-	-	-
FE 15	initobj	<Type>	&	-
FE 17	cpblk	-	int32,&,&	-
FE 18	initblk	-	int32,int32,&	-
FE 1A	rethrow	-	-	-
FE 1C	sizeof	<Type>	-	int32
FE 1D	refanytype	-	*	&

Fonte: Serge (2002, p. 422)