UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

IMPLEMENTAÇÃO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO COM GERAÇÃO DE CÓDIGO LVIS E MSIL

GUILHERME LUÍS MABA

BLUMENAU
2013

2013/2-06

GUILHERME LUÍS MABA

IMPLEMENTAÇÃO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO COM GERAÇÃO DE CÓDIGO LVIS E MSIL

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.

Profa. Joyce Martins, Mestre - Orientadora

BLUMENAU
2013

2013/2-06

IMPLEMENTAÇÃO DE UM COMPILADOR PARA UMA LINGUAGEM DE PROGRAMAÇÃO COM GERAÇÃO DE CÓDIGO LVIS E MSIL

Por

GUILHERME LUÍS MABA

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

Presidente: Profa. Joyce Martins, Mestre – Orientadora, FURB

Membro: Prof. Mauro Marcelo Mattos, Doutor – FURB

Membro: Prof. José Roque Voltolini da Silva – FURB

Blumenau, 11 de dezembro de 2013

Dedico este trabalho a todos os meus familiares que me apoiaram em todos os momentos, em especial, à minha avó. Também dedico à minha namorada e amigos.

AGRADECIMENTOS

À minha família, que me apoiou em todos os momentos e decisões da minha vida.

Aos meus amigos, pelos auxílios técnicos e troca de experiências.

À minha orientadora, Joyce Martins, por ter acreditado na conclusão deste trabalho, me cobrando, apoiando e auxiliando quando precisei.

Viver é isso: ficar se equilibrando o tempo todo, entre escolhas e consequências.

Jean-Paul Sartre

RESUMO

Este trabalho relata sobre a especificação da linguagem de programação estruturada MAB e sobre o desenvolvimento do compilador na linguagem Python, que gera código intermediário para as máquinas virtuais *Low Level Virtual Machine* (LLVM) e *Common Language Runtime* (CLR). Para especificar os símbolos léxicos, a sintaxe e a semântica da linguagem foram utilizados, respectivamente, definições regulares, notação *Backus-Naur Form* (BNF) e esquemas de tradução. Os códigos intermediários são gerados em LLVM *Instruction Set* (LVIS), código intermediário de três endereços, e em *MicroSoft Intermediate Language* (MSIL), código intermediário na notação pós-fixa. É feita uma comparação entre os códigos gerados.

Palavras-chave: Linguagens de programação. Máquina virtual de três endereços. Máquina virtual de pilha. LLVM. CLR.

ABSTRACT

This report describes about the specification of the structured programming language MAB and about the development of the compiler in Python language that generates intermediate code for the virtual machines Low Level Virtual Machine (LLVM) and Common Language Runtime (CLR). To specify the lexical symbols, the syntax and the semantics of the language were used respectively, regular definitions, Backus-Naur Form (BNF) notation and schemes of translation. The intermediate codes are generated in LLVM Instruction Set (LVIS), intermediate three-address code, and MicroSoft Intermediate Language (MSIL), intermediate code in the postfix notation. A comparison is made between the generated codes.

Key-words: Programming languages. Three addresses virtual machine. Virtual stack machine. LLVM. CLR.

LISTA DE ILUSTRAÇÕES

Figura 1 – Etapas de compilação 16
Quadro 1 – Exemplo de definição de *tokens* com definições regulares 17
Quadro 2 – Exemplo de definição de regras gramaticais em notação BNF 17
Quadro 3 – Exemplo de esquema de tradução 18
Figura 2 – Árvore sintática com ações semânticas 18
Quadro 4 – Instruções de um código intermediário de três endereços 19
Quadro 5 – Expressão em C e representação em código intermediário três endereços 19
Quadro 6 – Código pós-fixo em uma máquina de pilha 20
Quadro 7 – Algumas instruções em LVIS 21
Quadro 8 – Programa em C++ 21
Quadro 9 – Código objeto em LVIS 22
Quadro 10 – Exemplos de instruções com controle de fluxo em LVIS 22
Quadro 11 – Algumas instruções em MSIL 23
Quadro 12 – Código objeto em MSIL 23
Quadro 13 – Exemplos de instruções com controle de fluxo em MSIL 24
Quadro 14 – Método escrito em Java 25
Quadro 15 – Método em código intermediário LVIS 25
Quadro 16 – Programa na linguagem sequencial imperativa 26
Quadro 17 – Código objeto correspondente em MSIL 26
Quadro 18 – Programa na linguagem orientada a objetos 26
Quadro 19 – Código objeto correspondente em MSIL 27
Quadro 20 – Funções em Python 27
Quadro 21 – Utilizando compilação JIT em funções Python 27
Quadro 22 – Definições regulares 29
Quadro 23 – Palavras reservadas 29
Quadro 24 – Operadores 29
Quadro 25 – Gramática em notação BNF 30
Figura 3 – Diagrama de casos de uso 32
Quadro 26 – Caso de uso: `Criar programa` 32
Quadro 27 – Caso de uso: `Abrir programa` 32
Quadro 28 – Caso de uso: `Salvar programa` 33

Quadro 29 – Caso de uso: `Compilar programa MAB` 33
Quadro 30 – Caso de uso: `Compilar LVIS` 34
Quadro 31 – Caso de uso: `Compilar MSIL` 34
Figura 4 – Diagrama de classes 35
Figura 5 – Diagrama de atividades 37
Quadro 32 – Definição de um *token* no PLY 38
Quadro 33 – Definição de uma regra gramatical no PLY 39
Quadro 34 – Classe `MabLexer` 40
Quadro 35 – Palavras reservadas 41
Quadro 36 – Classe `MabParser` 41
Quadro 37 – Método `p_expression` 42
Quadro 38 – Método `p_error` 42
Quadro 39 – Exemplos de exceções de validações semânticas 42
Quadro 40 – Método `p_read` 43
Quadro 41 – Comando `read` nos dicionários `LVIS_CODE` e `MSIL_CODE` 44
Figura 6 – Interface do compilador 44
Figura 7 – Janela para selecionar um programa 45
Figura 8 – Programa editado 46
Figura 9 – Mensagem de programa salvo com sucesso 46
Figura 10 – Programa compilado 47
Figura 11 – Execução do programa na máquina virtual LLVM 47
Figura 12 – Execução do programa na máquina virtual CLR 48
Quadro 42 – Programa fatorial na linguagem MAB 49
Quadro 43 – Programa na linguagem LVIS 49
Quadro 44 – Programa na linguagem MSIL 50
Quadro 45 – Tipos de dados 51
Quadro 46 – Comparativo dos trabalhos correlatos com o compilador desenvolvido 51

LISTA DE SIGLAS

API – *Application Programming Interface*

BCEL – *Byte Code Engineering Library*

BNF – *Backus-Naur Form*

CLR – *Common Language Runtime*

GALS – Gerador de Analisadores Léxicos e Sintáticos

JIT – *Just-In-Time*

JVM – *Java Virtual Machine*

LLVM – *Low Level Virtual Machine*

LVIS – LLVM *Virtual Instructions Set*

MAB – Linguagem de programação especificada pelo autor (MABa)

MSIL – *MicroSoft Intermediate Language*

PLY – *Python Lex-Yacc*

RF – Requisitos Funcionais

RNF – Requisitos Não Funcionais

UML – *Unified Modeling Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS DO TRABALHO 13
1.2 ESTRUTURA DO TRABALHO 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 COMPILADORES 15
2.1.1 Análise léxica 16
2.1.2 Análise sintática 17
2.1.3 Analisador semântico 17
2.1.4 Geração de código intermediário 19
2.2 LLVM 20
2.3 CLR 22
2.4 TRABALHOS CORRELATOS 24
2.4.1 Tradutor Java – LLVM 24
2.4.2 Compilador de linguagem de programação sequencial imperativa 25
2.4.3 Compilador de linguagem de programação orientada a objetos 26
2.4.4 Pymothoa 27
3 DESENVOLVIMENTO 28
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 28
3.2 ESPECIFICAÇÃO DA LINGUAGEM MAB 28
3.3 ESPECIFICAÇÃO DO COMPILADOR 31
3.3.1 Diagrama de casos de uso 31
3.3.2 Diagrama de classes 34
3.3.3 Diagrama de atividades 37
3.4 IMPLEMENTAÇÃO 38
3.4.1 Técnicas e ferramentas utilizadas 38
3.4.2 Analisador léxico 39
3.4.3 Analisadores sintático e semântico 41
3.4.4 Geração de código intermediário 43
3.4.5 Operacionalidade da implementação 44
3.5 RESULTADOS E DISCUSSÃO 48
4 CONCLUSÕES 52

4.1 EXTENSÕES 53
REFERÊNCIAS BIBLIOGRÁFICAS 54

1 INTRODUÇÃO

A necessidade de escrever sequências de códigos para realizar as computações desejadas vem desde o final da década de 1940, quando foi proposto por John von Neumann que um programa poderia ser armazenado em um computador (LOUDEN, 2004, p. 2). As sequências de códigos, conforme relata Delamaro (2004, p. 1), podem representar alguma forma de dado ou podem ser uma instrução a ser realizada pelo computador. O conjunto destas instruções constitui um programa que define o que e como deve ser executado. Louden (2004, p. 3) afirma que práticas de codificação requerem um planejamento minucioso, porque alguns detalhes das sequências de códigos são desafiadores e às vezes podem ser complexos mesmo que se tenha uma boa fundamentação teórica.

Sendo assim, para desenvolver programas em um alto nível de abstração, sem a necessidade de criar diretamente as sequências de códigos de máquina, foram desenvolvidas as linguagens de programação de alto nível. "Existem milhares de linguagens fonte, que vão de linguagens de programação tradicionais, tais como Fortran e Pascal, às linguagens especializadas que emergiram virtualmente em quase todas as áreas de aplicação de computadores" (AHO et al., 2008, p. 1). Porém, para executar um programa escrito em uma linguagem de alto nível é preciso compilar o mesmo para uma linguagem de máquina, real ou virtual, como *Java Virtual Machine* (JVM), *Low Level Virtual Machine* (LLVM) e *Common Language Runtime* (CLR).

Wennborg (2010, p. 3) descreve que a LLVM é uma máquina virtual que executa programas representados por um conjunto de instruções de três endereços, conhecido por LLVM *Virtual Instruction Set* (LVIS). Esta execução pode ser interpretada pela máquina virtual, pode ser compilada dinamicamente ou executada diretamente utilizando seu código objeto. Já a CLR, conforme Cordeiro (2005, p. 1) define, é a máquina virtual baseada em pilha do *framework* .NET. Esta máquina executa códigos de uma linguagem intermediária, a *MicroSoft Intermediate Language* (MSIL), ou código nativo para máquina virtual.

Diante do exposto, neste trabalho propõe-se especificar uma linguagem de programação e desenvolver um compilador, gerando código LVIS e MSIL, respectivamente, para as máquina virtuais LLVM e CLR do *framework* .NET.

1.1 OBJETIVOS DO TRABALHO

Este trabalho tem como objetivo desenvolver um compilador para uma linguagem de alto nível, gerando código LVIS e MSIL.

Os objetivos específicos do trabalho são:

a) efetuar as análises léxica, sintática e semântica da linguagem de alto nível, detectando e diagnosticando erros de compilação;
b) traduzir os programas na linguagem de alto nível para código da máquina virtual LLVM;
c) traduzir os programas na linguagem de alto nível para código da máquina virtual CLR;
d) possibilitar a execução dos códigos gerados;
e) comparar os códigos gerados.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está disposto em quatro capítulos: introdução, fundamentação teórica, desenvolvimento e conclusões. No próximo capítulo encontram-se os conceitos que serviram de fundamentação para o desenvolvimento do trabalho, incluindo compiladores e as fases do processo de compilação, assim como as máquinas virtuais LLVM e CLR. O capítulo traz também a descrição dos trabalhos correlatos.

O capítulo 3 aborda o desenvolvimento do compilador, especificando os requisitos, as estruturas léxica, sintática e semântica da linguagem, os diagramas e a implementação, concluindo com os resultados obtidos.

O quarto capítulo traz as conclusões do trabalho desenvolvido e sugestões de possíveis extensões.

2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 trata de compiladores, descrevendo os módulos que compõem um compilador conceitual. A seção 2.2 apresenta a máquina virtual LLVM e na seção 2.3 é explicada a máquina virtual CLR. Já na seção 2.4 são descritos trabalhos correlatos.

2.1 COMPILADORES

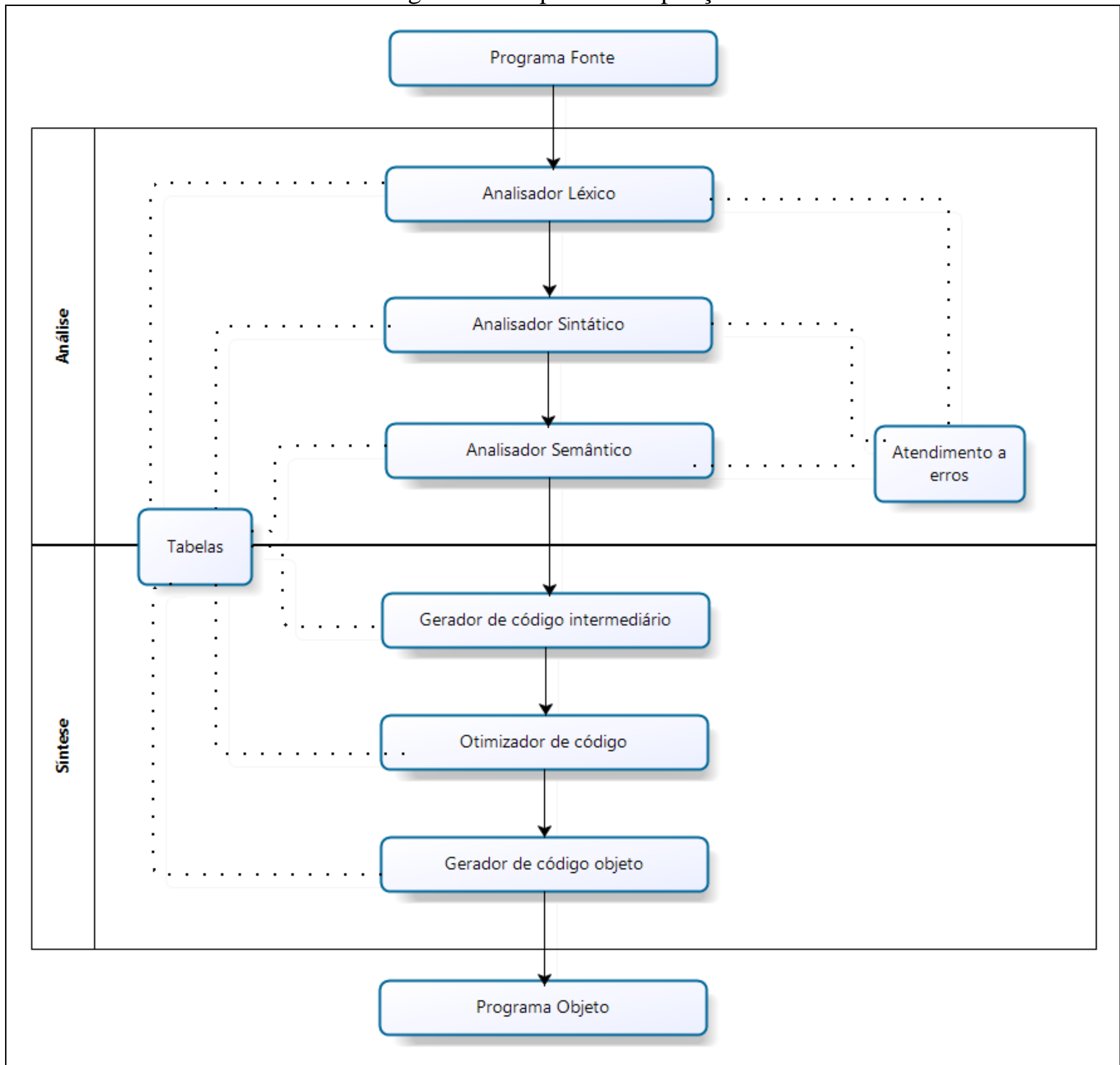
Compiladores são necessários para criação de programas de computador. Segundo Louden (2004, p. 1), um compilador recebe um código fonte de entrada, geralmente escrito em uma linguagem de alto nível, e gera um código objeto para ser executado pela máquina alvo.

Inicialmente o processo de compilação faz a análise do programa fonte em três fases: análise léxica, análise sintática e análise semântica. Na análise léxica são reconhecidos os *tokens*, símbolos básicos das linguagens. Na análise sintática é verificado se os *tokens* estão de acordo com a sequência estabelecida pelas regras gramaticais. Na análise semântica é determinado se os *tokens*, das construções sintáticas, estão semanticamente corretos.

Após o processo de análise, Louden (2004, p. 9) afirma que o compilador passa por mais três fases, sendo elas: gerador de código intermediário, otimizador de código e gerador de código. O gerador de código intermediário gera um código independente da máquina alvo, com o objetivo de facilitar o processo de tradução do código fonte para o código objeto. O otimizador de código transforma o código gerado na fase anterior de tal forma que o código objeto resultante seja mais rápido em tempo de execução. O gerador de código tem como função gerar código objeto em linguagem de máquina.

Como pode ser observado na Figura 1, as três primeiras fases de um compilador são agrupadas e denominadas de análise. As fases seguintes são denomidas de síntese (AHO et al., 2008, p. 5). Nas próximas seções são detalhadas as quatro primeiras fases.

Figura 1 – Etapas de compilação



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

2.1.1 Análise léxica

De acordo com Price e Toscani (2001, p. 7), o analisador léxico é a primeira fase de um compilador, também conhecido como *scanning*. Faz a leitura caracter por caracter de um programa fonte com o objetivo de identificar as unidades léxicas, conhecidas por *tokens*. Os *tokens* podem ser: delimitadores, identificadores, palavras reservadas, constantes numéricas e alfanuméricas, símbolos especiais, entre outros (PRICE; TOSCANI, 2001, p. 7).

Para a definição dos *tokens* são usadas definições regulares. O Quadro 1 apresenta alguns exemplos de definições regulares, onde, por exemplo, `constante_inteira` é um *token* composto por um (`[0-9]`) ou mais dígitos (`[0-9]*`).

Quadro 1 – Exemplo de definição de *tokens* com definições regulares

```
constante_inteira = [0-9][0-9]*
constante_real    = [0-9].[0-9]*
identificador     = [a-zA-Z]*
```

Na análise léxica pode ser identificada alguma sequência de caracteres que não é aceita ou não pertence ao conjunto dos *tokens* definidos. Neste caso, é enviada uma mensagem de erro.

2.1.2 Análise sintática

A partir da lista de *tokens* reconhecidos pelo análisador léxico, o analisador sintático verifica se a sequência dos *tokens* está de acordo com a gramática definida, criando uma estrutura de árvore, chamada de árvore de derivação ou árvore sintática. Este reconhecimento também é chamado de *parsing*. As regras gramaticais são definidas geralmente por uma gramática livre de contexto (PRICE; TOSCANI, 2001, p. 31).

Em uma gramática livre de contexto, as regras gramaticais podem ser representadas pela notação *Backus-Naur Form* (BNF). O Quadro 2 apresenta uma BNF que define expressões aritméticas com os operadores de soma e subtração.

Quadro 2 – Exemplo de definição de regras gramaticais em notação BNF

```
<expressão>   ::= <termo> <expressão_>
<expressão_>  ::= <operador> <termo> <expressão_> | ε
<termo>       ::= número
<operador>    ::= + | -
```

Fonte: adaptado de Price e Toscani (2001, p. 88).

Segundo Aho et al. (2008, p. 74), o analisador sintático faz também a verificação dos erros sintáticos que podem ocorrer, como por exemplo: expressões aritméticas com parênteses posicionados incorretamente, comando não permitido dentro de um bloco, entre outros.

> Frequentemente, boa parte da detecção e recuperação de erros num compilador gira em torno da fase sintática. Isto porque os erros ou são sintáticos por natureza ou são expostos quando o fluxo de *tokens* proveniente do analisador léxico desobedece às regras gramaticais que definem a linguagem de programação. (AHO et al., 2008, p. 73).

2.1.3 Analisador semântico

Segundo Aho et al. (2008, p. 4), a análise semântica utiliza a estrutura hierárquica definida pela análise sintática para: identificar operadores e operandos nas expressões, verificar a declaração e o uso de identificadores, fazer a validação de tipos de operandos em expressões e comandos, entre outros.

> Muitas vezes, o Analisador Sintático opera conjuntamente com o Analisador Semântico, cuja principal atividade é determinar se as estruturas sintáticas analisadas fazem sentido [...]. (PRICE; TOSCANI, 2001, p. 9).

Outra verificação importante do analisador semântico é a validação de unicidade, onde pode ser detectado quais identificadores foram declarados em duplicidade.

Observa-se que não existe uma forma padronizada e automatizada para especificar a análise semântica, assim como existem as definições regulares e a BNF para auxiliar nas análises anteriores. Para isso, podem ser usados esquemas de tradução.

> Um *Esquema de Tradução* é uma extensão de uma gramática livre de contexto, extensão esta realizada através de associação de atributos aos símbolos gramaticais e de ações semânticas às regras de produção. Os esquemas de tradução constituem uma notação apropriada para a especificação de operações que devem ser realizadas por um compilador durante a fase de análise sintática. (PRICE; TOSCANI, 2001, p. 86).

Um esquema de tradução com as ações semânticas especificadas pode ser observado no Quadro 3.

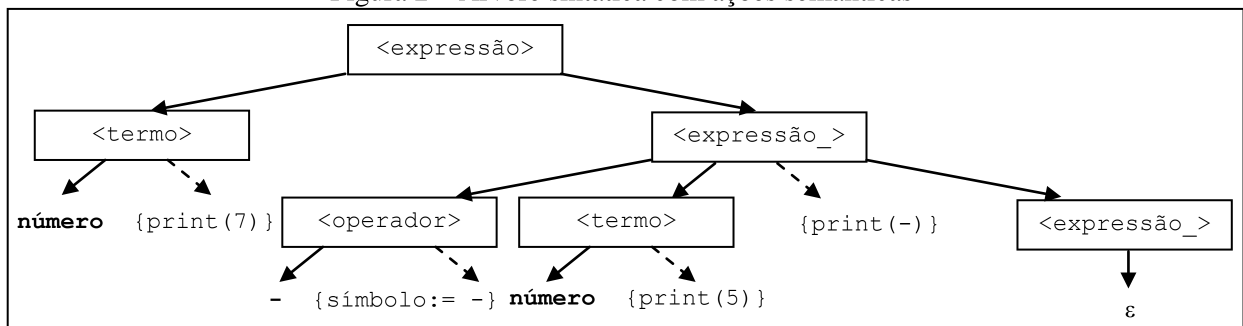
Quadro 3 – Exemplo de esquema de tradução

```
<expressão>    ::= <termo> <expressão_>
<expressão_>   ::= <operador> <termo> {print(símbolo)} <expressão_> | ε
<termo>        ::= número {print(número.lexema)}
<operador>     ::= + {símbolo:= +} | - {símbolo:= -}
```

Fonte: adaptado de Price e Toscani (2001, p. 88).

Na execução das ações semânticas, código representado entre chaves no Quadro 3, é utilizada a estratégia chamada de *depth-first*, onde a árvore sintática correspondente à expressão analisada é avaliada da esquerda para direita, visitando os nós folhas. Na Figura 2 tem-se a árvore sintática com as ações semânticas para a expressão `7 - 5`, que, ao ser avaliada, imprime `7 5 -`.

Figura 2 – Árvore sintática com ações semânticas



Fonte: adaptado de Price e Toscani (2001, p. 88).

Durante a análise semântica do programa fonte é criada uma tabela com o intuito de armazenar informações sobre identificadores de variáveis, de subrotinas, de parâmetros de subrotinas, de definição de tipos, entre outros, dependendo da linguagem fonte. Esta tabela é chamada de tabela de símbolos. A cada ocorrência de um identificador no programa, a tabela de símbolos é acessada, seja para inserir uma informação nova ou para encontrar um identificador e fazer verificações no programa fonte (PRICE; TOSCANI, 2001, p. 14).

2.1.4 Geração de código intermediário

De acordo com Price e Toscani (2001, p. 115), a fase de geração de código é a tranformação da árvore sintática e das informações obtidas nas análises anteriores, em um código objeto resultante. Normalmente os compiladores geram um código intermediário, que traz algumas vantangens, tais como: possibilidade de realizar otimizações na compilação, facilidade de depuração do código e possibilidade de tradução em várias máquinas alvo, já que o código intermediário é independente de máquina.

A representação do código intermediário pode possuir uma variedade de formas e propriedades. As formas mais usuais são chamadas: código de três endereços e notação pós-fixa.

Aho et al. (2008, p. 201) descrevem que as instruções do código intermediário de três endereços podem referenciar, no máximo, três variáveis ou três endereços de memória, sendo compostas por operações binárias ou unárias. Desta forma, duas variáveis podem ser utilizadas com operadores binários e outra para armazenar o resultado da operação. O Quadro 4 mostra instruções de um código intermediário de três endereços.

Quadro 4 – Instruções de um código intermediário de três endereços

```
A := B op C
A := op B
A := B
goto L
if A oprel B goto L
```

Fonte: adaptado de Price e Toscani (2001, p. 117).

Quando o programa fonte possui expressões com mais de uma operação, estas expressões são representadas em duas ou mais instruções. O Quadro 5 traz um exemplo de expressão escrita na linguagem C e sua representação em código intermediário de três endereços.

Quadro 5 – Expressão em C e representação em código intermediário três endereços

```
//expressão em C
X = A + B * 10;

//representação em código intermediário
temp := B * 10
X = A + temp
```

Como pode ser observado no Quadro 5, são utilizadas variáveis temporárias para armazenar os resultados intermediários.

A notação pós-fixa ou notação polonesa, segundo Louden (2004, p. 407), foi projetada para uma máquina hipotética de pilha, com a idéia de criar um único intepretador para os compiladores Pascal independente de plataforma. Esta representação é composta por memória de código, uma memória para armazenar as variáveis e uma pilha para dados temporários. Os

registradores são necessários para manter a pilha e realizar a execução. O Quadro 6 descreve uma representação hipotética de um código intermediário pós-fixo em uma máquina de pilha, baseado na expressão `2*a+(b-3)`.

Quadro 6 – Código pós-fixo em uma máquina de pilha

```
lda 2 //carrega a constante 2 na pilha
lod a //carrega o valor da variável a na pilha
mpi   //faz a multiplicação dos valores na pilha e empilha resultado
lod b //carrega o valor da variável b na pilha
ldc 3 //carrega a constante 3 na pilha
sbi   //faz a subtração dos valores na pilha e empilha resultado
adi   //faz a adição dos valores na pilha e empilha resultado
```

Fonte: adaptado de Louden (2004, p. 408).

2.2 LLVM

LLVM é uma máquina virtual que suporta linguagens com tipagem estática e dinâmica. Lattner (2013c) afirma que, com a evolução da máquina virtual LLVM, é possível desenvolver programas em diversas linguagens de programação, citando-se como exemplo Java, Python, Ruby, entre outras. Em LLVM um programa é executado de três maneiras: pode ser gerado o código objeto para interpretação na máquina virtual, pode ser gerado o código binário para execução direta no compilador da LLVM ou o programa pode ser compilado dinamicamente e executado diretamente pela máquina virtual (WENNBORG, 2010, p. 3).

O conjunto de instruções da máquina, o LVIS, está no formato de código de três endereços, onde cada instrução tem um ou dois operandos, produzindo um único resultado, ou seja, cada instrução possui no máximo três endereços. Lattner e Adve (2002, p. 3) afirmam que as instruções da máquina virtual são similares às instruções dos processadores comuns, mas evitam detalhes específicos, tais como tamanho das palavras, registradores físicos, *pipelines* e operações de baixo nível. Sendo assim, a LLVM não possui um conjunto fixo de registradores definidos. Os registradores são nomeados temporariamente e podem conter valores de tipos primitivos, tais como inteiros e ponto flutuante, ou tipos estruturados, tais como ponteiros, *arrays* e funções. Ainda de acordo com Lattner e Adve (2002, p. 4), as instruções LVIS são polimórficas, ou seja, uma única instrução `add` pode ter vários tipos de operandos, reduzindo o número de instruções distintas. O Quadro 7 traz algumas instruções LVIS.

Quadro 7 – Algumas instruções em LVIS

tipo	instrução	descrição
aritméticas	<result> = add <ty> <op1>, <op2>	retorna a soma de dois operandos do tipo <ty> especificado
	<result> = sub <ty> <op1>, <op2>	retorna a diferença dois operandos
	<result> = mul <ty> <op1>, <op2>	retorna o produto de dois operandos
	<result> = sdiv <ty> <op1>, <op2>	retorna o quociente de dois operandos
relacionais	<result> = icmp <cond> <ty> <op1>, <op2>	retorna um valor lógico (true ou false) a partir da comparação (<cond>) de dois operandos. <cond> pode ser eq (=), ne (<>), ugt (>), uge (≥), ult (<), ule (≤)
lógicas	<result> = and <ty> <op1>, <op2>	retorna o resultado da operação and entre dois operandos
	<result> = or <ty> <op1>, <op2>	retorna o resultado da operação or entre dois operandos
conversão	<result> = fptoui <ty> <value> to <ty2>	converte um tipo ponto flutuante para um tipo inteiro
	<result> = uitofp <ty> <value> to <ty2>	converte um tipo inteiro para um tipo ponto flutuante
retorno	ret <type> <value>	retorna um possível valor ou variável
	ret void	retorna vazio
chamada	<result> = call <ty> <value>	faz a chamada de alguma função com retorno
	call <ty> <value>	faz a chamada de alguma função
entrada / saída	call i32 @puts(i8* <var>)	imprime na saída padrão o valor informado
	call i32 (i8* ...)* scanf(i8* <value>, i32* <value2>)	lê e retorna um valor da entrada padrão
controle de fluxo	br label <dest>	transfere o controle do fluxo para a instrução indicada <dest> (desvio incondicional)
	br i1 <cond>, label <iftrue>, label <iffalse>	verifica se o valor de <cond> é true, em caso positivo, transfere o controle do fluxo para a instrução <iftrue>, caso contrário, transfere para a instrução <iffalse> (desvio condicional)

Fonte: adaptado de Lattner (2013b).

No Quatro 8 pode ser visualizado um exemplo de uma função para somar dois valores inteiros, desenvolvida na linguagem C++, e no Quadro 9 o código objeto correspondente para a máquina virtual LLVM.

Quadro 8 – Programa em C++

```
int sum (int a, int b){
  return a + b;
}
```

Fonte: Continuum Analytics (2010).

Quadro 9 – Código objeto em LVIS

```
define i32 @sum(i32 %a, i32 %b) {
entry:
  %tmp = add i32 %a, %b  ;resulta em tmp = a + b
  ret i32 %tmp           ;retorna tmp
}
```

Fonte: adaptado de Continuum Analytics (2010).

No Quadro 10 podem ser observados exemplos de outras instruções LVIS.

Quadro 10 – Exemplos de instruções com controle de fluxo em LVIS

```
...
%cond = icmp eq i32 %a, %b  ;resulta em a = b
br i1 %cond ,               ;se %cond=true
      label %IfEqual,       ;desvia para %IfEqual
      label %IfUnequal      ;senão desvia para %IfUnequal
IfEqual:   ret i32 1        ;retorna 1
IfUnequal: ret i32 0        ;retorna 0
...
```

Fonte: adaptado de Lattner (2013b).

2.3 CLR

Segundo Cordeiro (2005, p. 1), o *framework* .NET utiliza a máquina virtual CLR na qual são executados os programas compilados nas linguagens disponíveis para a plataforma. Depois de compilados para a linguagem intermediária, a MSIL, a máquina virtual gerencia a execução dos programas.

> MSIL é um conjunto de instruções baseado em pilha desenvolvido para ser facilmente gerada a partir de um código fonte por compiladores e outras ferramentas. Vários tipos de instruções são providas, incluindo instruções aritméticas e operações lógicas, controle de fluxo, acesso direto à memória, manipulação de exceções, e invocação de métodos. (CORDEIRO, 2005, p. 2).

A linguagem MSIL é orientada a objetos e suporta também instruções para conversão de tipos, criação e manipulação de objetos e gestão dos operandos na pilha. Em uma máquina de pilha, conforme descreve Martins (2013), os operandos são colocados na pilha de avaliação até que um operador exija o desempilhamento e a manipulação dos operandos, empilhando o resultado da operação. O Quadro 11 traz algumas instruções MSIL.

Quadro 11 – Algumas instruções em MSIL

tipo	instrução	descrição
aritméticas	add	adiciona dois valores da pilha (topo, topo-1), empilhando o resultado
	div	divide dois valores da pilha, empilhando o resultado
	mul	multiplica dois valores da pilha, empilhando o resultado
	sub	subtrai dois valores da pilha, empilhando o resultado
relacionais	ceq	compara dois valores da pilha, se forem iguais, empilha 1 (true), senão empilha 0 (false)
	cgt	compara dois valores da pilha, se o primeiro valor for maior do que o segundo, empilha 1 (true), senão empilha 0 (false)
	clt	compara dois valores da pilha, se o primeiro valor for menor do que o segundo, empilha 1 (true), senão empilha 0 (false)
lógicas	and	executa a operação and com dois valores da pilha, empilha 1 (true) ou 0 (false), conforme o caso
	or	executa a operação or com dois valores da pilha, empilha 1 (true) ou 0 (false), conforme o caso
conversão	conv.r4	converte o valor que está na pilha para float32, empilhando o resultado
	conv.i4	converte o valor que está na pilha para int32, empilhando o resultado
retorno	ret	retorna um possível valor que está na pilha
chamada	call <method>	faz a chamada do método especificado
entrada / saída	System.Console::ReadLine()	lê um valor da entrada padrão, empilhando o valor lido
	System.Console::WriteLine()	imprime na saída padrão o valor que está na pilha
controle de fluxo	br <label>	transfere o controle do fluxo para a instrução <label> (desvio incondicional)
	brfalse <label>	verifica se o valor da pilha é 0 (false). Em caso positivo, transfere o controle do fluxo para a instrução <label> (desvio condicional)
	brtrue <label>	verifica se o valor da pilha é 1 (true). Em caso positivo, transfere o controle do fluxo para a instrução <label> (desvio condicional)

Fonte: adaptado de Microsoft (2013).

No Quadro 12 pode ser observado o código objeto em MSIL correspondente à função para somar dois valores inteiros (do Quadro 8), enquanto o Quadro 13 traz o código objeto em MSIL correspondente ao código do Quadro 10.

Quadro 12 – Código objeto em MSIL

```
.method public static int32 sum(int32 a, int32 b)
{ ldarg a       //empilha o valor de a
  ldarg b       //empilha o valor de b
  add           //desempilha o valor do topo e o valor do topo-1
                //soma os valores desempilhados, empilha o resultado
  ret           //desempilha e retorna o valor do topo da pilha
}
```

Quadro 13 – Exemplos de instruções com controle de fluxo em MSIL

```
...
  lcd.i4 a          //empilha o valor de a
  ldc.i4 b          //empilha o valor de b
  ceq               //desempilha o valor do topo e o valor do topo-1
                    //compara se os valores desempilhados são iguais,
                    //empilha o resultado
  brfalse IfUnequal //desempilha o valor do topo
                    //se é false, desvia para IfUnequal
  ldc.i4.1          //empilha 1
  br Exit           //desvia para Exit
IfUnequal: ldc.i4.0 //empilha 0
Exit:      ret      //desempilha e retorna o valor do topo da pilha
...
```

O MSIL *Assembler* (`ilasm.exe`) é o programa responsável por gerar o código executável a partir do código objeto em MSIL.

2.4 TRABALHOS CORRELATOS

A seguir são apresentados trabalhos relacionados ao proposto, incluindo um tradutor de programas escritos em Java para máquina virtual LLVM (CANTÚ, 2008), dois compiladores, para linguagens distintas, mas que geram código para plataforma .NET (TOMAZELLI, 2004; LEYENDECKER, 2005) e a ferramenta Pymothoa, uma extensão da linguagem Python com adição de um compilador *Just-In-Time* (JIT) utilizando a LLVM (LARABEL, 2012).

2.4.1 Tradutor Java – LLVM

Cantú (2008) apresenta um tradutor para programas escritos em Java que gera código objeto para máquina virtual LLVM. Cantú (2008, p. 32) afirma que a tradução é baseada no processo de extração de informação do arquivo `.class` e geração das instruções para código da máquina virtual LLVM. O tradutor tem como passos: fazer a leitura das instruções do arquivo `.class`, computar o estado da pilha para cada instrução, analisar as exceções e instruções para a máquina alvo e gerar código objeto para a máquina alvo.

Este projeto traz um estudo das máquinas virtuais JVM e LLVM, identificando as correspondências entre instruções *bytecodes* JVM e instruções LLVM. De acordo com Cantú (2008, p. 34), foi utilizada uma *Application Programming Interface* (API) chamada BCEL[1], que possui um conjunto de bibliotecas para ler os arquivos `.class` e gerar instruções *bytecodes* da linguagem Java. Segundo Cantú (2008, p. 53), para gerar o código objeto em LLVM foram utilizados analisadores gerados pela ferramenta GALS[2] (GESSER, 2002) e a

---

[1] *Byte Code Engineering Library*.
[2] Gerador de Analisadores Léxicos e Sintáticos.

ferramenta llvm-as (LATTNER, 2013a), que é responsável por ler o código objeto e gerar instruções LLVM.

No Quadro 14 pode ser observado um método escrito na linguagem Java. O código correspondente gerado para a máquina virtual LLVM é apresentado no Quadro 15.

Quadro 14 – Método escrito em Java

```
public static int teste( int x, int y ) {
    int z = 19;
    int w = 19;
    return (x-y)*(z-w);
}
```

Fonte: adaptado de Cantú (2008, p. 48).

Quadro 15 – Método em código intermediário LVIS

```
define i32 @teste()
{
entry:
    %intvar2 = alloca i32
    %ptr2 = getelementptr i32* %intvar2
    store i32 19, i32* %ptr2
    %intvar3 = alloca i32
    %ptr3 = getelementptr i32* %intvar3
    store i32 19, i32* %ptr3
    %ptr1temp1 = load i32* %ptr1
    %ptr0temp2 = load i32* %ptr0
    %temp0 = sub i32 %ptr0temp2,%ptr1temp1
    %ptr3temp4 = load i32* %ptr3
    %ptr2temp5 = load i32* %ptr2
    %temp3 = sub i32 %ptr2temp5,%ptr3temp4
    %temp6 = mul i32 %temp0,%temp3
    ret i32 %temp6
}
```

Fonte: adaptado de Cantú (2008, p. 50).

2.4.2 Compilador de linguagem de programação sequencial imperativa

Tomazelli (2004, p. 13) afirma que o trabalho tem como objetivo especificar uma linguagem de programação cujo código objeto resultante do processo de compilação pode ser executado pela máquina virtual CLR. É uma linguagem de programação sequencial imperativa que executa programas do tipo *console*. Os comandos foram definidos na língua portuguesa, sendo *case-sensitive* e semelhantes aos da linguagem C. Os comandos da linguagem proposta incluem a declaração de variáveis locais, um comando de seleção, um comando de repetição, um comando para entrada e outro para saída de dados, definição e uso de módulos (TOMAZELLI, 2004, p. 35).

Um programa fonte da linguagem criada exemplificando a declaração de variáveis pode ser visto no Quadro 16. A partir do programa fonte, o compilador gera código intermediário em MSIL, em um arquivo com extensão `.il`. O código resultante da compilação do programa apresentado no Quadro 16 encontra-se no Quadro 17. O arquivo `.il`

é interpretado pelo `ilasm.exe`, resultando em um arquivo com extensão `.exe`, que pode ser executado na máquina virtual CLR (TOMAZELLI, 2004, p. 5).

Quadro 16 – Programa na linguagem sequencial imperativa

```
programa teste
{
  vazio principal()
  {
    inteiro idade;
    real peso = 80.0; //inicialização
  }
}
```

Fonte: Tomazelli (2004, p. 41).

Quadro 17 – Código objeto correspondente em MSIL

```
.assemly extern macorlib { }
.assemly 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
  }
}
```

Fonte: Tomazelli (2004, p. 41).

2.4.3 Compilador de linguagem de programação orientada a objetos

Leyendecker (2005) apresenta a especificação de uma linguagem orientada a objetos e o desenvolvimento de um compilador que gera código em MSIL. A linguagem tem como características principais (LEYENDECKER, 2005, p. 33): sobrecarga de métodos, herança simples, possibilidade de usar rotinas escritas em outras linguagens .NET e vice-versa. Sua sintaxe é similar à linguagem C para os comandos de controle de fluxo e às linguagens Java e C# para as classes.

No Quadro 18 pode ser observado um exemplo de código fonte escrito na linguagem especificada e no Quadro 19 o respectivo código em MSIL.

Quadro 18 – Programa na linguagem orientada a objetos

```
namespace org.furb.tcc;
using System.String;
using System.Console;

public class HelloWorld {

  public static void main(String[] args)
  {
    Console.WriteLine("Hello, World!");
  }
}
```

Fonte: Leyendecker (2005, p. 54).

Quadro 19 – Código objeto correspondente em MSIL

```
.namespace org.furb.tcc {

.class public ansi auto HelloWorld extends [mscorlib] System.Object {

  .method public void .ctor() cil managed {
    ret
  }

  .method public static hidebysig void main(string[] args) cil managed
  {
    .entrypoint
    ldstr "Hello World!"
    call void [mscorlib]System.Console ::WriteLine(string)
    ret
  }
}
}
```

Fonte: adaptado de Leyendecker (2005, p. 54).

2.4.4 Pymothoa

Segundo Larabel (2012), o projeto Pymothoa estende a linguagem de programação Python com compilação JIT, utilizando a ferramenta de compilação da máquina virtual LLVM. Para tanto, não foi modificada a semântica nem a interpretação da linguagem. Desta forma, um código fonte escrito originalmente em Python irá funcionar normalmente. No entanto, foi especificado um novo dialeto de Python, semelhante à linguagem C, onde variáveis devem ser declaradas e possuem tipagem estática.

No Quadro 20 tem-se funções escritas puramente em Python. Para habilitar a compilação JIT do Pymothoa, adiciona-se uma anotação acima do código das funções, sempre declarando o tipo do retorno e dos argumentos, conforme pode ser verificado no Quadro 21.

Quadro 20 – Funções em Python

```
def reduce_sum(A, n):
    total = 0
    for i in xrange(n):
        total += A[i]
    return total

def main():
    N = 10
    print reduce_sum(range(N), N)

if __name__ == '__main__':
    main()
```

Fonte: Larabel (2012).

Quadro 21 – Utilizando compilação JIT em funções Python

```
from pymothoa.jit import function
from pymothoa.dialect import *
from pymothoa.types import *

@function(ret=Int,args=[Array(Int), Int])
def reduce_sum(A, n):
  # add declaration of total
  var ( total = Int )
  total = 0
```

Fonte: Larabel (2012).

3 DESENVOLVIMENTO

Este capítulo relata sobre o desenvolvimento do compilador, abordando as seguintes seções:

a) requisitos;
b) especificação da linguagem MAB;
c) especificação do compilador, incluindo os diagramas de casos de uso, de classe e de atividades;
d) implementação e operacionalização;
e) resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF) do compilador são:

a) possuir um editor para criar e manter programas na linguagem especificada (RF);
b) permitir salvar arquivos texto com extensão `.mab` (RF);
c) fazer as análises léxica, sintática e semântica do programa, detectando erros de compilação (RF);
d) gerar código intermediário para máquina virtual LLVM (RF);
e) gerar código intermediário para máquina virtual da plataforma .NET (RF);
f) gerar código executável a partir do código intermediário LVIS (RF);
g) gerar código executável a partir do código intermediário MSIL (RF);
h) ser implementado utilizando a linguagem de programação Python (RNF);
i) utilizar a biblioteca PLY para geração dos analisadores léxico e sintático (RNF);
j) utilizar a biblioteca PyQT para criação de interface gráfica (RNF);
k) utilizar o programa QT Designer para desenhar a interface gráfica do compilador (RNF);
l) executar no sistema operacional Linux (RNF).

3.2 ESPECIFICAÇÃO DA LINGUAGEM MAB

A linguagem MAB possui algumas características das linguagens Python e Java. É uma linguagem estruturada projetada para executar na plataforma .NET e LLVM. As características da linguagem são:

a) ser fortemente tipada;
b) ser *case-sentive*;
c) permitir declaração de funções;

d) possuir funções para leitura e escrita no console;

e) possuir funções de conversão de tipos;

f) ser utilizada para fins de estudo na geração de código intermediário.

Definidas as características básicas da linguagem, na etapa seguinte foram especificados seus *tokens*, utilizando as definições regulares apresentadas no Quadro 22.

Quadro 22 – Definições regulares

```
IDENTIFIER: [a-zA-Z][a-zA-Z0-9_]*
BOOL: true | false
INTEGER: [0-9][0-9]*
FLOAT: [0-9]+.[0-9]+
STRING: '(.*?)'
COMMENT_BLOCK: #(.*?)#
```

Conforme apresentado no Quadro 22, um identificador pode iniciar com qualquer letra, minúscula ou maiúscula, seguida de zero ou mais ocorrências de letras, números ou o caracter *underscore* (_). Além disso, a linguagem tem as seguintes constantes:

a) lógicas: definidas pelas palavras reservadas `true` e `false`;

b) numéricas: representam números sem (`INTEGER`) e com (`FLOAT`) casas decimais;

c) cadeia de caracteres (`STRING`): aceitam qualquer caracter entre aspas simples, inclusive sequências de `escape` (`\n`, `\r`, `\t`, `\\`, `\'`, `\"`).

É possível fazer comentários (de bloco ou de linha) que iniciam e terminam com sustenido (`#`).

A linguagem possui as palavras reservadas relacionadas no Quadro 23.

Quadro 23 – Palavras reservadas

```
and       float      or        tofloat
bool      function   print     toint
break     if         println   tostr
const     int        read      void
else      main       return    while
elseif    not        str
```

Os operadores aritméticos, relacionais e lógicos podem ser observados no Quadro 24, enquanto o símbolo `=` é utilizado no comando de atribuição.

Quadro 24 – Operadores

operador	descrição	operador	descrição
`==`	igual	`+`	adição,
`!=`	diferente	`-`	subtração
`>`	maior	`*`	multiplicação
`>=`	maior ou igual	`/`	divisão
`<`	menor	`and`	e
`<=`	menor ou igual	`or`	ou
`+, -`	sinais unários	`not`	negação

Os caracteres de formatação reconhecidos, porém ignorados, são: nova linha (`\n`), tabulação horizontal (`\t`) e retorno do cursor (`\r`).

A gramática da linguagem MAB encontra-se no Quadro 25.

Quadro 25 – Gramática em notação BNF

```
<program> ::= <var_global> <program> | <function> <program> | <empty>
<empty> ::= ""

<var_global> ::= <declaration_var>
<declaration_var> ::= <type> IDENTIFIER |
                      <type> IDENTIFIER = <expression> |
                      const <type> IDENTIFIER = <expression>

<function> ::= function main ( ) <delimiter_function> |
               function <type> ( <parameters> ) <delimiter_function> |
               function void ( <parameters> ) <delimiter_function>

<type> ::= float | bool | int | str

<parameters> ::= <type> IDENTIFIER <parameters2>    | <empty>
<parameters2> ::= , <type> IDENTIFIER <parameters2> | <empty>

<delimiter_function> ::= { <block_function> }

<block_function> ::= <cmd_f> <block_function> | <empty>
<cmd_f>::= <assign>|<cmds_general>|<declaration_var>|<if>|<return>|<while>

<block_general> ::= <cmd_g> <block_general> | <empty>
<cmd_g>:= <assign>|break|<cmds_general>|<declaration_var>|<if>|<while>

<assign> ::= IDENTIFIER = <expression>

<cmds_general> ::= <print> | <read> | <factor>

<print> ::= print <value_str> | println <value_str>
<value_str> ::= STRING | STRING + <value_str> | <tostr> | <tostr> + <value_str>
                IDENTIFIER | IDENTIFIER + <value_str>

<read> ::= read ( IDENTIFIER )

<if> ::= if ( <expression> ) <delimiter> <if2>
<if2> ::= elseif ( <expression> ) <delimiter> <if2> | else <delimiter> | <empty>

<return> ::= return <expression>

<while> ::= while ( <expression> ) <delimiter>

<delimiter> ::= { <block_general> }

<expression> ::= <logical>
<logical> ::= <logical> or <member> | <logical> and <member> | <member>
<member> ::= <relational> | BOOL | not <member>
<relational> ::= <numeric> <operator> <numeric> | <numeric>
<operator> ::= == | >= | > | < | <= | !=
<numeric> ::= <numeric> + <term> | <numeric> - <term> | <term>
<term> ::= <term> * <factor> | <term> / <factor> | <factor>
<factor> ::= INTEGER | FLOAT | BOOL | STRING | IDENTIFIER |
             <tostr> | <tofloat> | <toint> |
             IDENTIFIER () | IDENTIFIER ( <argument_list> ) |
             ( <expression> ) | + <factor> | - <factor>
<argument_list> ::= <argument_list> , <expression> | <expression>

<tostr> ::= tostr ( <expression> )
<toint> ::= toint ( <expression> )
<tofloat> ::= tofloat ( <expression> )
```

Também foram definidas as seguintes verificações semânticas:

a) no comando de atribuição, a variável e a expressão devem ser do mesmo tipo;

b) no comando `read`, a variável deve ser do tipo `str`;
c) nos comandos `print` e `println`, as variáveis devem ser do tipo `str`;
d) qualquer identificador (de função ou de variável) só pode ser utilizado se já foi declarado previamente;
e) nenhum identificador pode ser declarado mais de uma vez;
f) não é permitida atribuição de valores a uma variável constante, apenas quando da declaração;
g) o tipo e a quantidade de argumentos quando da chamada de uma função deve ser igual ao tipo e à quantidade de parâmetros declarados (tipo posicional);
h) o valor de retorno de uma função deve ser do mesmo tipo da função;
i) não é possível converter valores do tipo lógico em valores dos tipos `str`, `float` e `int`.

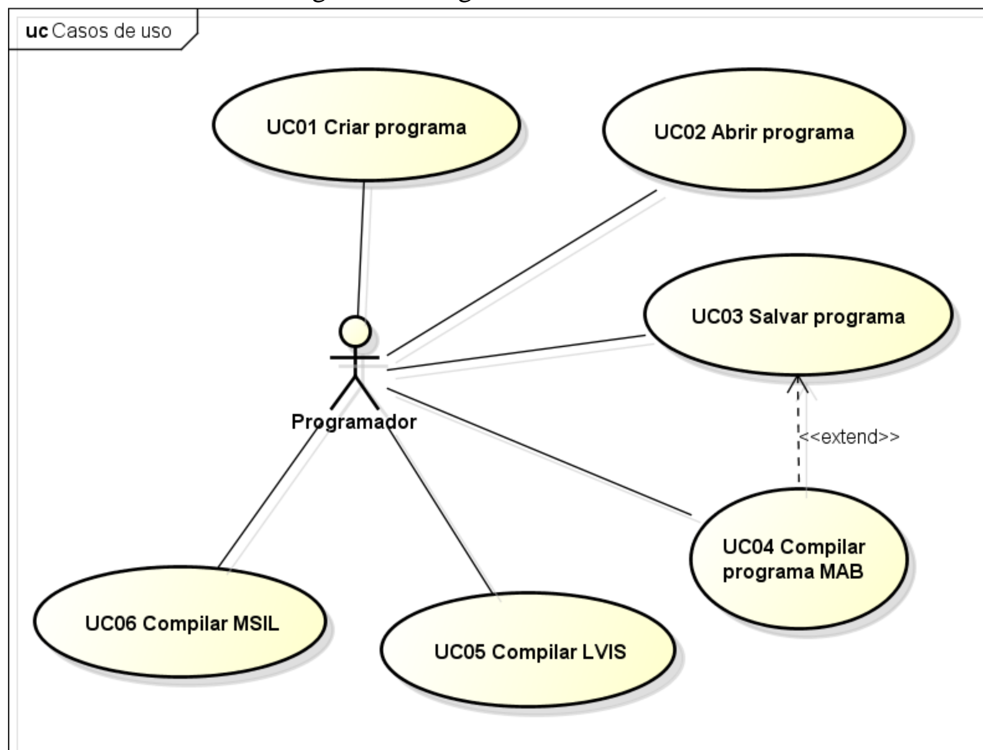
3.3 ESPECIFICAÇÃO DO COMPILADOR

Nesta seção é apresentada a especificação do compilador, incluindo os diagramas de casos de uso, de classes e de atividades da *Unlified Modeling Language* (UML). Os diagramas foram especificados usando a ferramenta Astah (ASTAH COMMUNITY, 2013).

3.3.1 Diagrama de casos de uso

Foi especificado um ambiente de compilação simplificado, com editor de programas, visualizador / compilador de códigos LVIS e MVIS e compilador da linguagem MAB propriamente dito. O ambiente possui seis casos de usos (Figura 3): `Criar programa`, `Abrir programa`, `Salvar programa`, `Compilar programa MAB`, `Compilar LVIS` e `Compilar MSIL`.

Figura 3 – Diagrama de casos de uso



Os casos de uso `Criar programa` (Quadro 26) e `Abrir programa` (Quadro 27) representam a ação inicial do programador. Para que seja possível compilar um programa e gerar código objeto é necessário ter um programa elaborado no editor do ambiente.

Quadro 26 – Caso de uso: `Criar programa`

UC01 – Criar programa	
Descrição	Possibilita criar um programa.
Pré-condições	Nenhuma.
Cenário principal	1. O programador aciona o botão `Novo`.
Pós-condições	Editor e console limpos e prontos para codificação de um novo programa.

Quadro 27 – Caso de uso: `Abrir programa`

UC02 – Abrir programa	
Descrição	Possibilita abrir arquivo contendo um programa codificado na linguagem.
Pré-condições	O arquivo deve conter a extensão `.mab`.
Cenário principal	1. O programador aciona o botão `Abrir`. 2. O ambiente abre uma janela exibindo os arquivos contidos na pasta local. 3. O programador procura e seleciona o arquivo desejado. 4. O ambiente carrega o arquivo no editor.
Pós-condições	Arquivo contendo o programa carregado no editor, com permissão de leitura e escrita.

O programador pode salvar um programa novo ou as alterações feitas em programa existente (Quadro 28).

Quadro 28 – Caso de uso: `Salvar programa`

UC03 – Salvar programa	
Descrição	Possibilita salvar um programa em um arquivo texto.
Pré-condições	Deve ter sido executado o caso de uso UC01 ou o caso de uso UC02.
Cenário principal	Existem duas maneiras de salvar um programa: 1. Salvar um programa novo: 1.1 O ambiente permite informar o nome do programa. 1.2 O programador informa o nome do programa no campo superior do ambiente. 1.3 O programador aciona o botão `Salvar`. 1.4 O ambiente emite uma mensagem informando que o programa foi salvo com sucesso. 2. Salvar um programa existente: 2.1 O programador aciona o botão `Salvar`. 2.2 O ambiente emite uma mensagem informando que o programa foi salvo com sucesso.
Pós-condições	Programa salvo em um arquivo com extensão `.mab`, na mesma pasta do ambiente de compilação, caso seja um programa novo, ou na pasta onde se localiza o arquivo original, caso seja um programa já existente.

A principal funcionalidade do ambiente de compilação está descrita no caso de uso `Compilar programa MAB` (Quadro 29), que especifica quais são os procedimentos para compilar um programa e gerar os códigos intermediários correspondentes.

Quadro 29 – Caso de uso: `Compilar programa MAB`

UC04 – Compilar programa MAB	
Descrição	Possibilita compilar um programa.
Pré-condições	Deve ter sido executado o caso de uso UC01 ou o caso de uso UC02.
Cenário principal	1. O programador aciona o botão `Compilar`. 2. O ambiente executa o caso de uso UC03. 3. O ambiente apresenta mensagem com o resultado da compilação. 4. O ambiente mostra os códigos LVIS e MSIL nos seus respectivos editores.
Cenário de exceção	No passo 2, o ambiente detecta um erro. Neste caso, são informados a linha e o erro ocorrido. O programador pode retornar para o caso de uso UC01, para o caso de uso UC02 ou até mesmo encerrar a execução.
Pós-condições	Códigos intermediários em LVIS e MSIL gerados na pasta do ambiente de compilação, em arquivo com extensão `.LL` e `.IL`, respectivamente.

Os casos de uso `Compilar LVIS` (Quadro 30) e `Compilar MSIL` (Quadro 31) são executados após o caso de uso `Compilar programa MAB` (Quadro 29). Servem para compilar os códigos intermediários gerados, em LVIS e em MSIL, e gerar o código executável para a respectiva máquina virtual.

Quadro 30 – Caso de uso: `Compilar LVIS`

UC05 – Compilar LVIS	
Descrição	Possibilita compilar o código intermediário LVIS, gerando código executável.
Pré-condições	Deve ter sido executado o caso de uso UC04 com sucesso.
Cenário principal	1. O programador aciona o botão `Compilar`, localizado acima do editor que contém o código intermediário LVIS. 2. O ambiente emite uma mensagem informando que o código executável foi gerado com sucesso.
Cenário de exceção	No passo 1, o ambiente detecta um erro. Neste caso, são informados a linha e o erro ocorrido. O programador pode retornar para o caso de uso UC01, para o caso de uso UC02 ou até mesmo encerrar a execução.
Pós-condições	Código executável para a máquina LLVM gerado na pasta do ambiente de compilação.

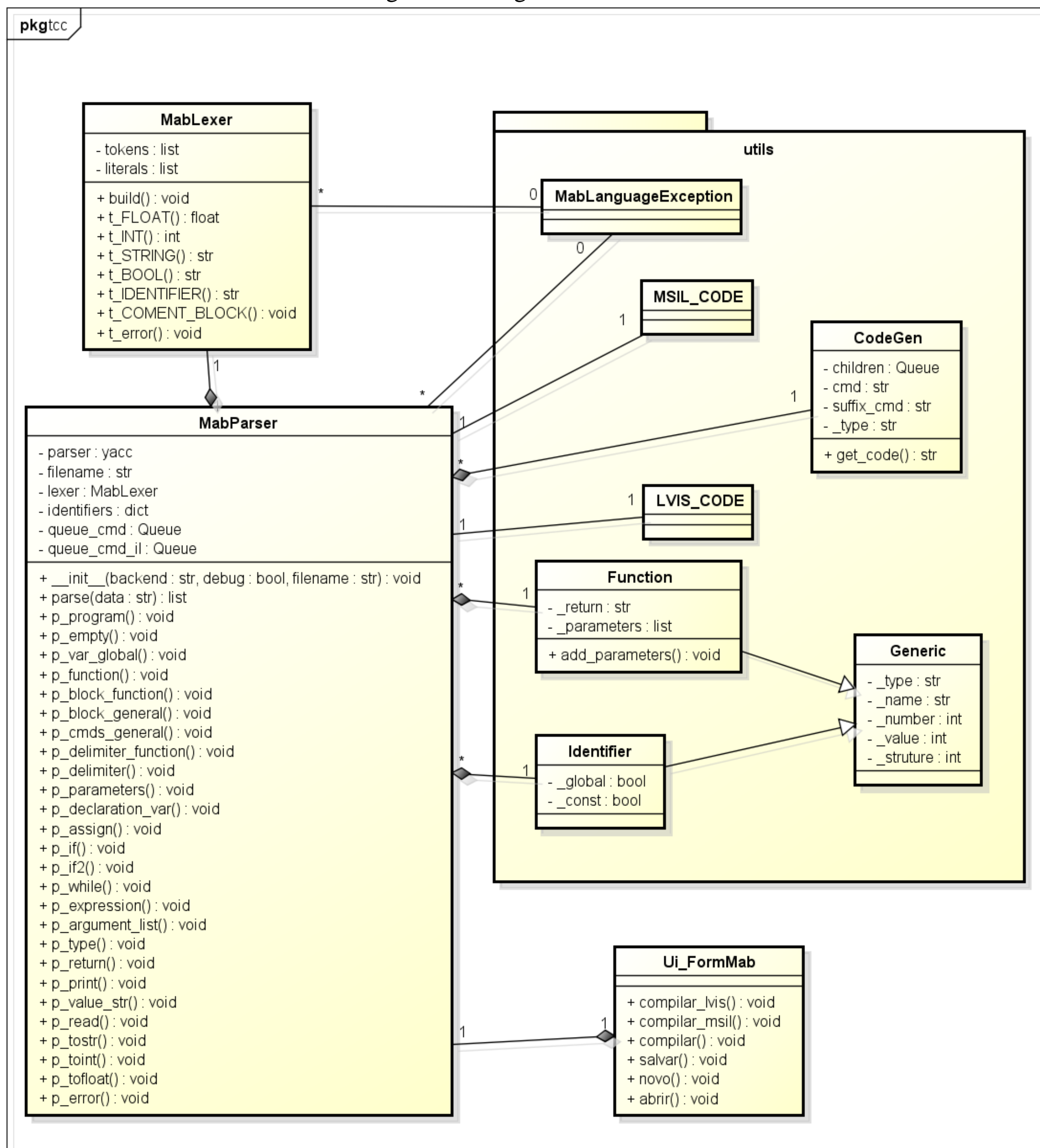
Quadro 31 – Caso de uso: `Compilar MSIL`

UC06 – Compilar MSIL	
Descrição	Possibilita compilar o código intermediário MSIL, gerando código executável.
Pré-condições	Deve ter sido executado o caso de uso UC04 com sucesso.
Cenário principal	1. O programador aciona o botão `Compilar`, localizado acima do editor que contém o código intermediário MSIL. 2. O ambiente de compilação emite uma mensagem informando que o código executável foi gerado com sucesso.
Cenário de exceção	No passo 1, o ambiente detecta um erro. Neste caso, são informados a linha e o erro ocorrido. O programador pode retornar para o caso de uso UC01, para o caso de uso UC02 ou até mesmo encerrar a execução.
Pós-condições	Código executável para a máquina .NET gerado na pasta do ambiente de compilação.

3.3.2 Diagrama de classes

Nesta seção é apresentado o diagrama de classes do ambiente de compilação desenvolvido, que pode ser observado na Figura 4. As classes foram separadas em dois pacotes: `tcc` e `utils`. No pacote `tcc` encontram-se as classes: `MabLexer`, `MabParser` e `UiFormMab`, enquanto no pacote `utils` são apresentadas as classes: `MabLanguageException`, `MSIL_CODE`, `LVIS_CODE`, `CodeGen`, `Function`, `Identifier` e `Generic`.

Figura 4 – Diagrama de classes



A classe `Ui_FormMab` é a classe responsável pela interface do ambiente de compilação. Sendo assim, tem a interação com o usuário. Além da manipulação de arquivos (`novo`, `abrir`, `salvar`), esta classe possui métodos (`compilar`, `compilar_lvis`, `compilar_msil`) que necessitam de uma instância de objeto da classe `MabParser`.

As funcionadades da classe `MabParser` são: definir a gramática da linguagem, realizar a análise sintática, a análise semântica e a geração de código intermediário em LVIS e em MSIL. A definição das regras gramáticais são encontradas nos métodos iniciados com `p_`, onde cada um destes métodos é responsável por fazer a análise sintática de sua(s) regra(s) e também a geração de código intermediário. A classe `MabParser` também contém os métodos:

p_error, __init__ e parse. O método p_error é chamado quando ocorre alguma exceção nas regras gramaticais, como por exemplo alguma sequência de código não prevista em nenhuma das regras. Desta forma, o método lança uma exceção para informar o erro detectado. O método __init__ é o método construtor da classe, responsável pela inicialização de todas as variáveis utilizadas pela classe, como por exemplo: dicionário de identificadores (identifiers), as filas de comandos (queue_cmd, queue_cmd_il) e o nome do arquivo (filename). O método parse tem como objetivo receber o programa na linguagem MAB e iniciar a execução das análises léxica, sintática e semântica e a geração de código intermediário.

Para realizar a análise léxica, a classe MabParser possui um atributo lexer, que é uma instância da classe MabLexer. A classe MabLexer é responsável por definir os *tokens* da linguagem através de métodos como: t_FLOAT, t_INT, t_STRING, t_BOOL, t_IDENTIFIER e t_COMENT_BLOCK, que definem, respectivamente, os números com casas decimais, os números sem casas decimais, as cadeia de caracteres, os valores lógicos, os identificadores e os comentários de bloco. A classe MabLexer possui um atributo chamado tokens, que é uma lista de todos os possíveis *tokens* definidos para a linguagem, inclusive palavras reservadas e símbolos especiais. Também possui o método t_error que é chamado quando o analisador léxico encontra algum caracter ou sequência de caracteres inválidos para a linguagem.

O pacote utils contém as classes auxiliares do ambiente de compilação. Uma destas classes é a MabLanguageException, que nada mais é que a classe de exceção própria da linguagem MAB, utilizada para lançar erros conhecidos de compilação (léxicos, sintáticos e semânticos).

Outras classes importantes são Function e Identifier, que são utilizadas pela classe MabParser quando é reconhecida uma função ou um identificador na análise de um programa fonte, alimentando a tabela de símbolos com instâncias destas classes. As classes Function e Identifier são uma especialização da classe Generic, que possui atributos e métodos genéricos para caracterizar qualquer tipo de identificador.

Os dicionários MSIL_CODE e LVIS_CODE são responsáveis por traduzir cada comando da linguagem no momento da geração de código em seu código intermediário da máquina virtual desejada, CLR ou LLVM. Assim, por exemplo, quando for necessário gerar código para o comando read, basta buscar o comando no dicionário da linguagem intermediária desejada para obter o código intermerdiário para o mesmo.

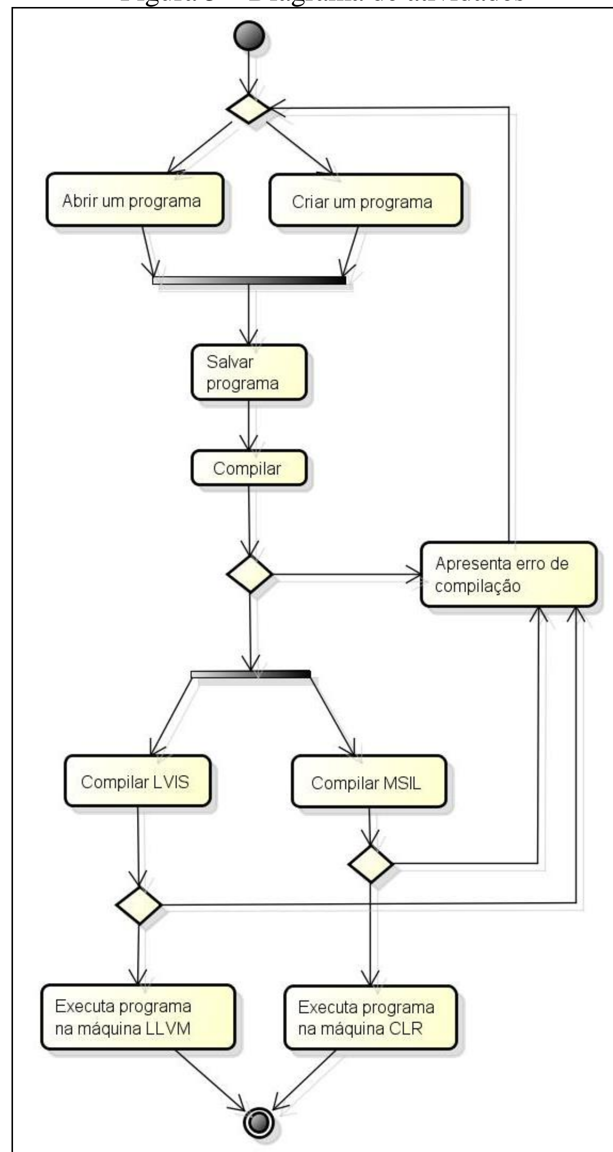
A classe CodeGen representa o código intermediário de cada linguagem alvo. Depois

das análises léxica, sintática e semântica, basta chamar o método `get_code` para obter o código final.

3.3.3 Diagrama de atividades

Com objetivo de mostrar o fluxo geral do ambiente de compilação, apresenta-se na Figura 5 um diagrama de atividades. Observa-se que são duas as atividade iniciais: criar um novo programa ou abrir um já existente. Em seguida, tem-se a opção de salvar o programa criado ou editado. Na sequência pode-se compilar o programa, gerando dois códigos intermediários, em LVIS e em MSIL, que são compilados e executados independentemente. Caso ocorra algum problema de compilação, é apresentado o erro ao usuário, retornando as atividades iniciais.

Figura 5 – Diagrama de atividades



3.4 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas no desenvolvimento. São descritos os principais módulos implementados e a operacionalidade da implementação.

3.4.1 Técnicas e ferramentas utilizadas

O ambiente de compilação foi desenvolvimento em Python, utilizando as bibliotecas PLY e PyQT e a ferramenta QT Designer. Foi utilizado como ambiente de desenvolvimento o editor Vim (MOOLENAAR, 2012).

A linguagem Python, de acordo com Lutz e Ascher (2004, p. 36), é uma linguagem de propósito geral, 100% orientada a objetos: todos os tipos, as contruções e as funções são objetos em Python. É uma linguagem multiplataforma. Um programa desenvolvido em Python pode ser portado para Unix ou Windows. Possui diversas funções embutidas para acesso aos serviços do sistema operacional, manipulação de tipos e arquivos, criação de interface gráfica, entre outras. Algumas características da linguagem são: tipagem dinâmina, gerenciamento de memória automático através do *garbage collector* e tratamento de exceções.

O PLY é um gerador de analisadores léxico e sintático, puramente escrito na liguagem Python e separado em dois módulos (BEAZLEY, 2011). O primeiro módulo é responsável pelo reconhecimento dos *tokens* a partir da especificação de definições regulares. Faz a análise léxica do texto de entrada informando os *tokens* reconhecidos ou identificando erros léxicos. O outro módulo é responsável por fazer a análise sintática da linguagem especificada através de uma gramática livre de contexto. Também faz o tratamento de erros sintáticos. Ao contrário das ferramentas com a mesma funcionalidade que geram como saída o código que implementa os analisadores léxico e sintático, o PLY não gera arquivos extras, somente gera tabelas para efetuar as análises, sendo os reconhecimentos léxico e sintático efetuados pelos módulos descritos.

No Quadro 32 é apresentada a especificação de um *token* em PLY. Já no Quadro 33 é apresentada a especificação de uma regra gramatical. Junto com a regra gramatical devem ser implementadas também as verificações semânticas e a geração de código do compilador.

Quadro 32 – Definição de um *token* no PLY

```
def t_INT(self, t):
  r"[0-9][0-9]*"

  t.value = int(t.value)
  return t
```

Quadro 33 – Definição de uma regra gramatical no PLY

```
def p_declaration_var(self, p):
'''declaration_var :
      type IDENTIFIER |
      CONST type IDENTIFIER EQUAL expression |
      type IDENTIFIER EQUAL expression'''

# aqui deve-se fazer as verificações semânticas
# e geração de código
```

De acordo com Lutz e Ascher (2004, p. 507), Python é uma linguagem que não possui nativamente uma ferramenta para desenvolvimento de intefaces gráficas. Para construção de telas em Python é necessário utilizar bibliotecas externas, sendo as mais utilizadas PyGTK e PyQT. PyQT oferece um conjunto de objetos para desenvolvimento de interfaces, incluindo: um conjunto substancial de *widgets*, classes para acesso a bancos de dados, *parser eXtensible Markup Language* (XML) e suporte a ActiveX no Windows.

Para usar a biblioteca PyQT existe uma ferramenta chamada QT Designer que facilita o desenho das telas e possui diversos *widgets* prontos para serem utilizados. QT Designer gera um arquivo com extensão `.ui`, que é convertido em um arquivo Python (GARDNER, 2013).

3.4.2 Analisador léxico

No desenvolvimento do compilador, o primeiro módulo especificado e implementado foi o analisador léxico. Utilizando a ferramenta PLY, foi implementada a classe `MabLexer` com a especificação das definições regulares e o tratamento de erros léxicos da linguagem MAB. Esta classe utiliza o módulo do PLY responsável por efetuar a análise léxica. No Quadro 34 é possível observar um trecho do código desta classe.

Quadro 34 – Classe `MabLexer`

```
class MabLexer:

tokens = [
'FLOAT','INT','STRING','BOOL','COMMA', 'IDENTIFIER', 'EQUAL', 'BREAK_LINE', \
'PAR_OPEN', 'PAR_CLOSE', 'DELIMITER_OPEN', 'DELIMITER_CLOSE', 'COMP_EQUAL', \
'COMP_NOT_EQUAL', 'COMP_GREATER', 'COMP_GREATER_EQUAL', 'COMP_LESS',
'COMP_LESS_EQUAL'] + reserved.values()

literals = ['+','-','*','/']

t_ignore = " \r\t"
t_EQUAL = r"="
t_COMMA = r","
t_PAR_OPEN = r"\("
t_DELIMITER_OPEN = r"\{"
t_PAR_CLOSE = r"\)"
t_DELIMITER_CLOSE = r"\}"

t_COMP_EQUAL = r"=="
t_COMP_NOT_EQUAL = r"!="
t_COMP_GREATER = r">"
t_COMP_GREATER_EQUAL = r">="
t_COMP_LESS = r"<"
t_COMP_LESS_EQUAL = r"<="

def build(self, **kwargs):
  self.lexer = lex.lex(module=self, **kwargs)

def t_BREAK_LINE(self, t):
  r"\n"
  t.lexer.lineno += len(t.value)
  if DEBUG_LEX:
    print '[LEX] BREAK LINE: ' + str(t.lexer.lineno)
  #ignora quebra de linha
  pass

def t_FLOAT(self, t):
  r"[0-9]+\.[0-9]+"
  t.value = float(t.value)
  if DEBUG_LEX:
    print "[LEX] FLOAT: %s" % t.value
  return t

def t_INT(self, t):
  r"[0-9][0-9]*"
  t.value = int(t.value)
  if DEBUG_LEX:
    print "[LEX] INT: %s" % t.value
  return t
...
```

Como observado no Quadro 34, a classe possui os atributos `tokens` e `literals` que definem respectivamente os *tokens* e os caracteres literais da linguagem. Quando o *token* é definido como um atributo, como `t_EQUAL`, ele é uma constante literal que representa geralmente um operador (aritmético ou relacional), algum delimitor (`\r`, `\t`) ou um símbolo especial. Quando o *token* é definido como um método, como `t_FLOAT`, é porque é especificado a partir de uma definição regular. Nesta mesma classe também foi criado um dicionário com as palavras reservadas da linguagem (Quadro 35).

Quadro 35 – Palavras reservadas

```
reserved = {
  'while':'DEF_WHILE',
  'if':'DEF_IF',
  'else':'DEF_ELSE',
  'elseif':'DEF_ELSEIF',
  'int':'DEF_INT',
  'float':'DEF_FLOAT',
  'bool':'DEF_BOOL',
  'str':'DEF_STR',
  'main':'DEF_MAIN',
  'function':'DEF_FUNCTION',
  'break':'BREAK',
  'void':'VOID',
  'return':'RETURN',
  'and':'AND',
  'or':'OR',
  'not':'NOT',
  'global':'GLOBAL',
  'const':'CONST',
  'tostr':'TOSTR',
  'toint':'TOINT',
  'tofloat':'TOFLOAT',
  'read':'READ',
  'print':'PRINT',
  'println':'PRINTLN',
}
```

3.4.3 Analisadores sintático e semântico

A próxima etapa foi a implementação dos analisadores sintático e semântico. Para implementar estes módulos foi criada uma classe chamada `MabParser`, que possui o analisador léxico (`MabLexer`) como atributo de classe (Quadro 36). Sendo assim, a classe inicialmente executa o analisador léxico para reconhecer os *tokens* do programa fonte processado.

Quadro 36 – Classe `MabParser`

```
class MabParser:

def __init__(self, backend='LLVM', debug=False, filename=""):
  self.lexer = MabLexer()
...
```

Nesta mesma classe foram definidas as regras sintáticas da linguagem. Como foi utilizada a ferramenta PLY, estas regras foram escritas em métodos separados, um para cada não terminal da gramática. Em algums métodos foram especificadas mais de uma regra gramatical. Um exemplo é o método `p_expression` que utiliza-se de várias regras sintáticas para definir uma expressão na linguagem. Pode-se observar a definição deste método no Quadro 37.

Quadro 37 – Método `p_expression`

```
def p_expression(self, p):
'''expression : logical
      logical : logical OR member | logical AND member | member
      member : relational | BOOL | NOT member
      relational : numeric operator numeric | numeric
      operator : COMP_EQUAL
               | COMP_NOT_EQUAL
               | COMP_GREATER
               | COMP_GREATER_EQUAL
               | COMP_LESS
               | COMP_LESS_EQUAL
      numeric : numeric '+' term | numeric '-' term | term
      term : term '*' fator | term '/' fator | factor
      factor :   INT
               | FLOAT
               | STRING
               | tostr
               | tofloat
               | toint
               | IDENTIFIER
               | IDENTIFIER PAR_OPEN PAR_CLOSE
               | IDENTIFIER PAR_OPEN argument_list PAR_CLOSE
               | PAR_OPEN expression PAR_CLOSE
               | '+' factor
               | '-' factor'''
```

Quando o compilador detecta algum erro sintático no programa processado, é chamado o método `p_error` (Quadro 38), que lança uma exceção (`MABSyntaxError`), apresentando o símbolo que ocasionou o erro. Um exemplo de erro sintático é a falta de um delimitador em um bloco `while`.

Quadro 38 – Método `p_error`

```
def p_error(self, p):
  raise MABSyntaxError("Erro na linha {0}, encontrado '{1}'.".
                      format(p.lineno, p.value))
```

Na classe `MabParser`, além de definidas as regras gramaticais, também é implementada a análise semântica. A análise semântica busca informações sobre os identificadores na tabela de símbolos para realizar as validações semânticas. No Quadro 39 podem ser observadas exceções decorrentes de validações semânticas.

Quadro 39 – Exemplos de exceções de validações semânticas

```
...
raise MABFunctionAlreadyDeclared(u"Função '{0}' já declarada".format(p[3]))
...
raise MABCommandNotFound(u"Comando 'return' não encontrado na função '{0}'" \
.format(p[3]))
...
raise MABSyntaxError(u"Identificador '{0}' já declarado como
global".format(p[3]))
...
raise MABSyntaxError(u"Função '{0}' precisa de parâmetros (".format(p[1])+
','.join(func.get_parameters()) +")")
...
raise MABSyntaxError(u"Função '{0}' esperava argumento do tipo\ {1}, mas
informado {2}".format(p[1], func.get_parameters()[index],tp))
...
raise MABSyntaxError(u'Impossível converter tipo booleano para inteiro')
...
```

3.4.4 Geração de código intermediário

A geração de código intermediário LVIS e MSIL foi implementada na classe `MabParser`, caracterizando tradução dirigida pela sintaxe, isso é, a geração de código foi implementada nos próprios métodos que definem as regras gramaticais. No Quadro 40 é apresentado o método que faz a geração de código do comando `read`.

Quadro 40 – Método `p_read`

```
...
def p_read(self, p):
'''read : READ PAR_OPEN IDENTIFIER PAR_CLOSE'''

   if self.is_first_parse():
     pass
   else:
     if self.identifiers.has_key(p[3]):
       code = CodeGen()
       var2 = self.get_name_expression()
       code.set_comand(LVIS_CODE.get('read', '').format(var2, p[3]))
       self.queue_cmd.put(code)

       #MSIL
       x = CodeGen()
       x.set_comand(MSIL_CODE.get('read', '').format(p[3]))
       self.queue_cmd_il.put(x)

   if self.debug:
      print '[YACC] READ: ' + p[1] + p[2] + str(p[3]) + p[4]
...
```

Como o compilador foi implementado em duas passagens, a geração de código intermediário é realizada somente na segunda. Na geração do código é utilizada a classe chamada `CodeGen`, que recebe o comando na linguagem intermediária. Uma instância de `CodeGen` é adicionada na fila de comandos `queue_cmd` ou `queue_cmd_il`, que são os respectivos códigos para as máquinas virtuais LLVM e CLR. Uma importante estrutura da classe `CodeGen` é o atributo `children`, que poderá ser nulo ou receber mais instâncias de `Codegen`. Esta estrutura é necessária para gerar código dos blocos de comandos, como funções, comandos de repetição (`while`) e seleção (`if-elseif-else`).

Os dicionários `LVIS_CODE` e `MSIL_CODE` são estruturas criadas para auxiliar o desenvolvimento, retornando o código intermediário de um comando específico, que dependendo de qual for, possui parâmetros a serem informados. Como exemplo, tem-se, no Quadro 41, o comando `read`, em LVIS e em MSIL, que encontra-se nos dicionários `LVIS_CODE` e `MSIL_CODE`, respectivamente.

Quadro 41 – Comando `read` nos dicionários `LVIS_CODE` e `MSIL_CODE`

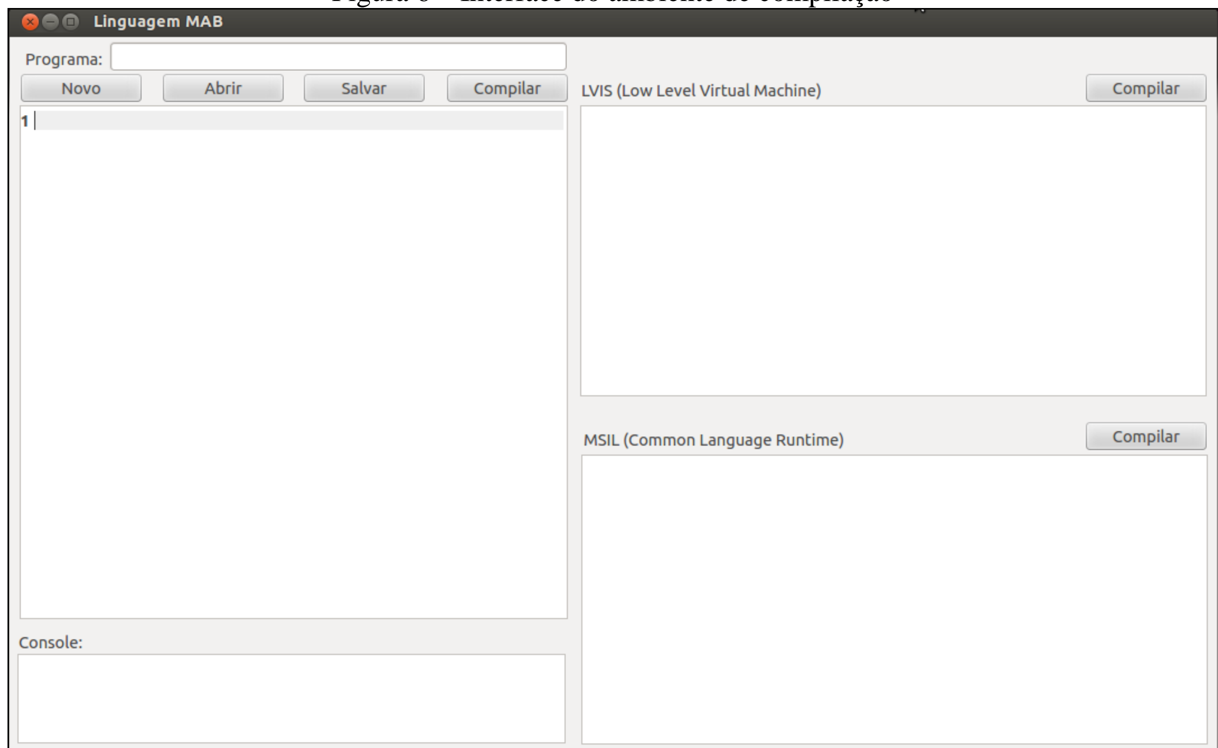
```
// dicionário de dados LVIS_CODE
LVIS_CODE = { ...,
'read': 'call i32 (i8*, ...)* @scanf( i8* getelementptr inbounds ([2 x i8]* \
@str_read, i32 0, i32 0), [40 x i8]* %{1} ) \n',
...}

// dicionário de dados MSIL_CODE
MSIL_CODE = { ...,
'read': 'call string [mscorlib]System.Console::ReadLine()\nstloc {0}\n',
...}
```

3.4.5 Operacionalidade da implementação

O uso do ambiente de compilação da linguagem MAB é bem intuitivo. Na Figura 6 pode ser observada a interface do ambiente desenvolvido. Para executá-lo é necessário ter instalado o compilador da liguagem Python e as bibliotecas PyQT e PLY.

Figura 6 – Interface do ambiente de compilação

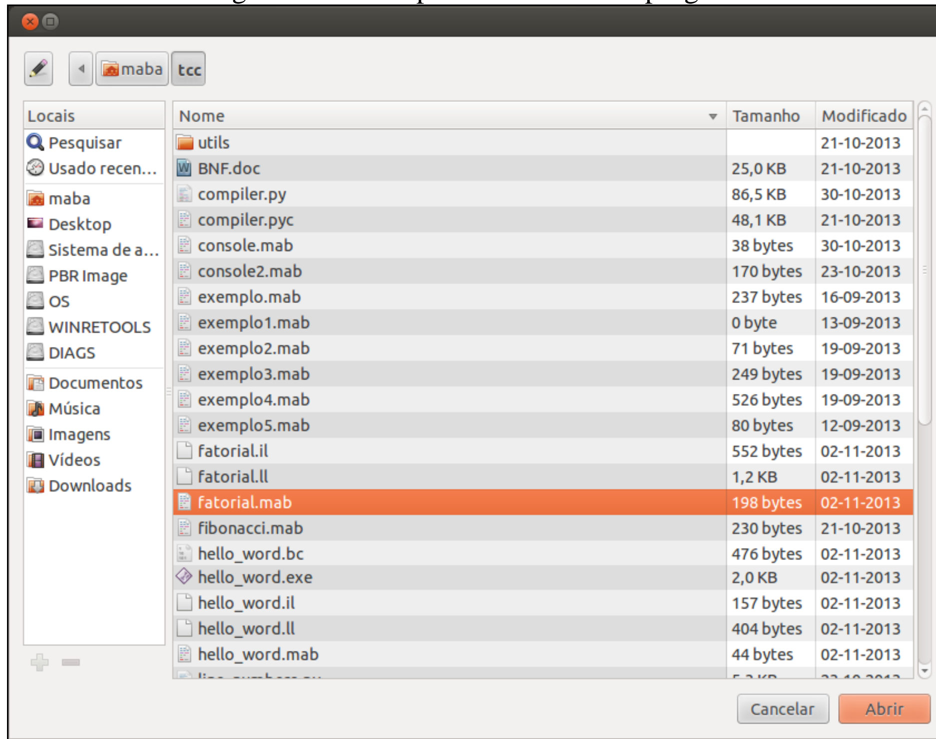


No lado esquerdo da interface encontram-se os botões `Novo`, `Abrir`, `Salvar` e `Compilar`. Abaixo tem-se o editor, para criar e/ou alterar um programa na linguagem, e o console, onde são apresentadas as mensagens do compilador. No lado direito existem os editores onde são mostrados os códigos intermediários em LVIS e em MSIL, sendo que cada editor possui um botão `Compilar`.

Ao pressionar o botão `Novo`, o editor, o console e os editores dos códigos intermediários são limpos. Assim, o programador pode codificar um novo programa. Ao

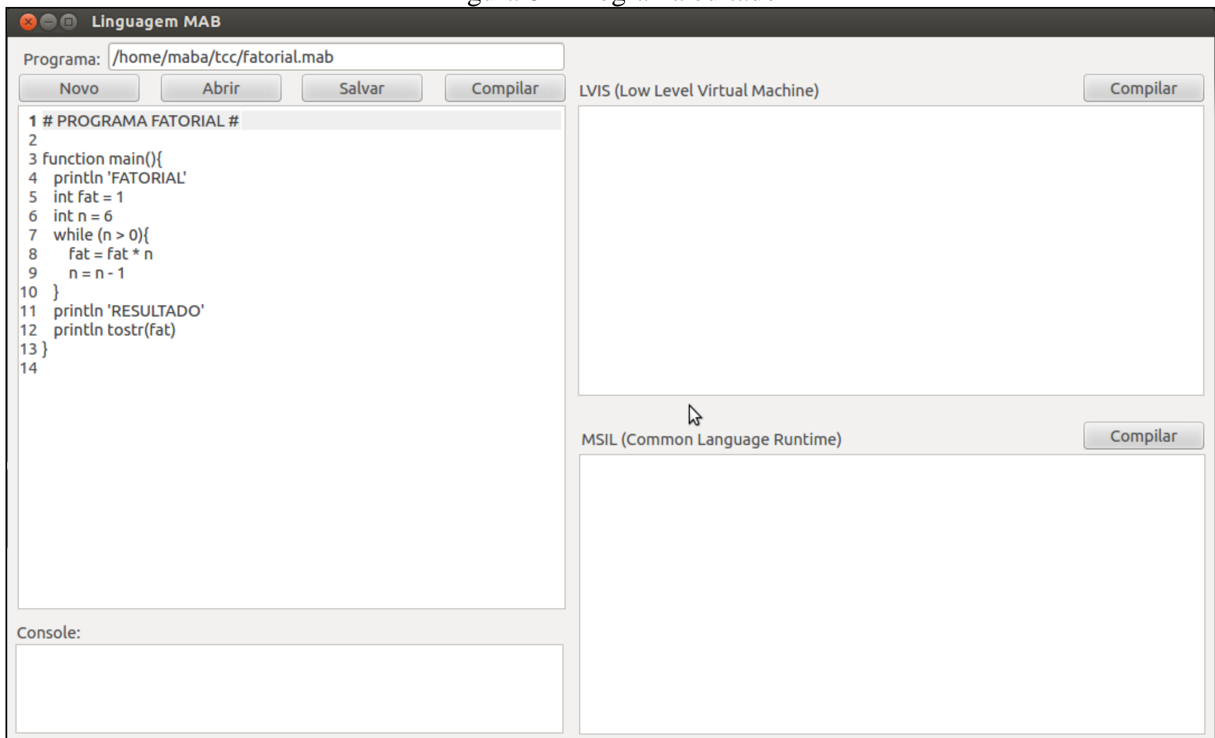
pressionar o botão `Abrir`, é aberta uma janela (Figura 7) para que o programador possa selecionar um programa que tenha permissão de acesso.

Figura 7 – Janela para selecionar um programa



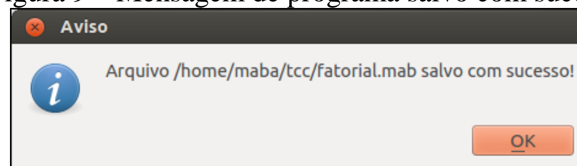
Depois de selecionado o arquivo, o mesmo é carregado no editor do ambiente e pode ser editado, salvo ou compilado. O mesmo procedimento ocorre para programas novos, mas neste caso é necessário informar o nome do programa no campo superior acima do editor. Na Figura 8 é apresentado um programa em edição.

Figura 8 – Programa editado



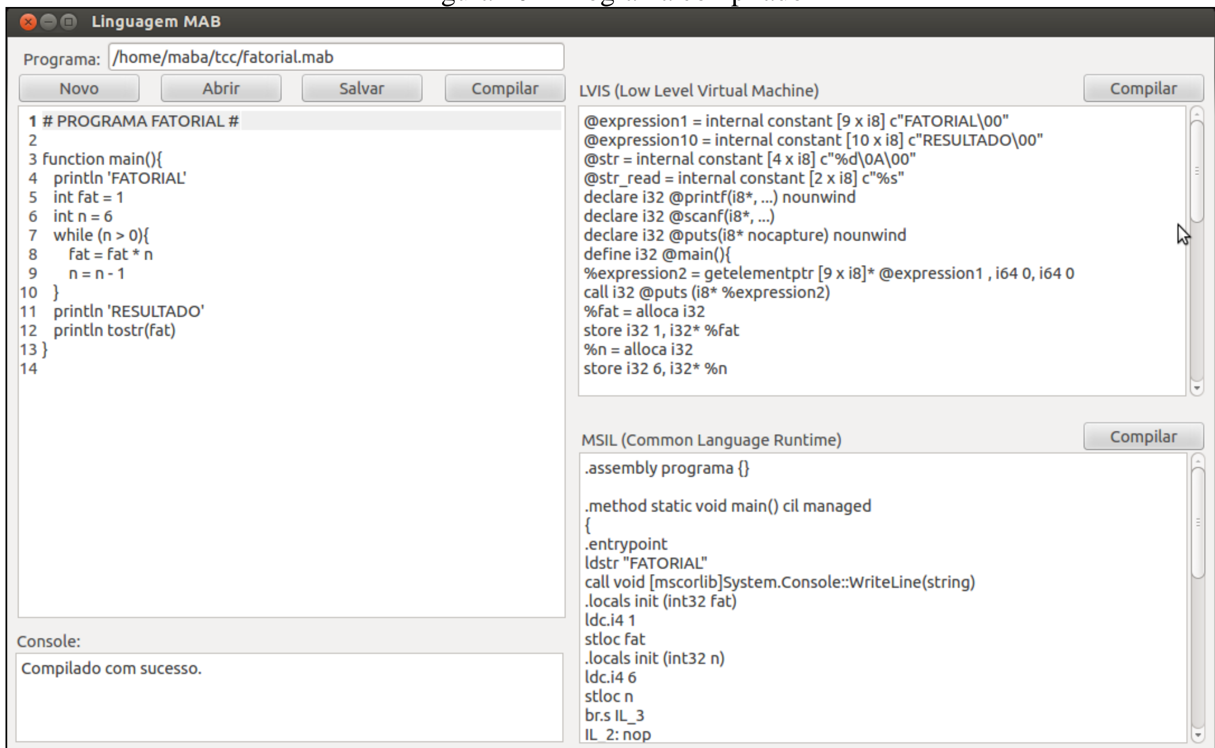
Em qualquer momento o programador pode salvar o programa, sendo mostrado um aviso de que o arquivo foi salvo (Figura 9).

Figura 9 – Mensagem de programa salvo com sucesso



Para compilar um programa é necessário pressionar o botão `Compilar`. É exibida uma mensagem no console informando que a compilação foi realizada com sucesso ou relatando algum erro de compilação. O resultado de um programa compilado com sucesso são os códigos intermediários em LVIS e em MSIL gerados e apresentados nos respectivos editores, conforme pode ser visualizado na Figura 10.

Figura 10 – Programa compilado



Para gerar o código executável do programa em cada máquina virtual (LLVM e CLR), basta o programador apertar o botão `Compilar` associado a cada editor de código intermediário. No console é apresentada uma mensagem indicando se o código executável foi gerado com sucesso para a máquina virtual correspondente.

Após gerado os códigos executáveis para máquinas virtuais CLR e LLVM, os mesmos podem ser executados pelo terminal do sistema operacional. Na Figura 11 é apresentada a execução do programa na máquina virtual LLVM, utilizando o aplicativo *lli*. Na Figura 12 é apresentada a execução do programa na máquina do *framework* .NET.

Figura 11 – Execução do programa na máquina virtual LLVM

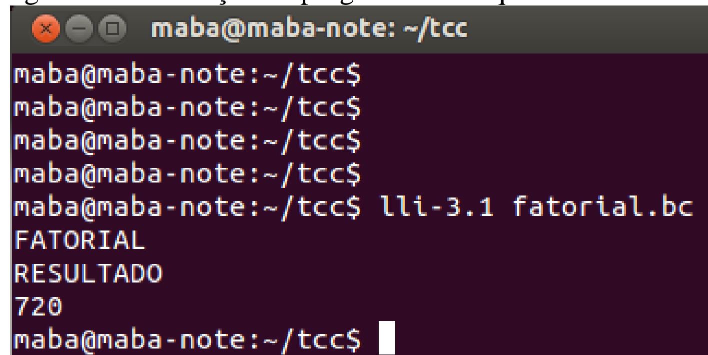
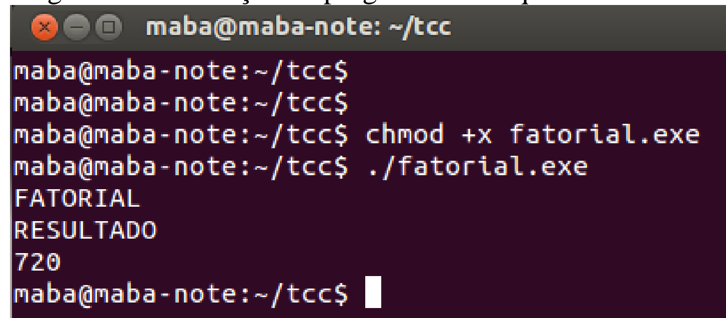


Figura 12 – Execução do programa na máquina virtual CLR



3.5 RESULTADOS E DISCUSSÃO

Neste trabalho foi apresentada a especificação de uma liguagem de programação e o desenvolvimento de um compilador que faz a geração de código intermediário para as máquinas virtuais LLVM e CLR. Os resultados obtidos alcançaram os objetivos. Apesar da linguagem MAB não ser tão completa quanto as linguagens de alto nível existentes, ela possui comandos e estruturas suficientes para analisar e comparar os códigos gerados nas diferentes máquinas virtuais.

No desenvolvimento da interface gráfica foram utilizadas as ferramentas PyQT e QT Designer, facilitando a integração com as classes do compilador, implementadas em Python.

Para as análises léxica, sintática e semântica foi utilizada a biblioteca PLY, que se mostrou muito eficiente e de fácil utilização. A estrutura da ferramenta torna legível cada função de seus módulos e facilita a leitura dos *tokens* e regras gramaticais definidas para linguagem. Isso acontece em função do PLY utilizar as *docstrings*[3] nos métodos em Python para realizar estas configurações. Foi realizado um ajuste durante o desenvolvimento do trabalho, pois não se tinha entendimento completo da ferramenta PLY. Inicialmente, a especificação dos *tokens* e da gramática da linguagem foi feita usando programação funcional. Verificou-se, no entanto, que o PLY possui suporte à orientação a objetos. Desta forma, o compilador foi reestruturado em classes, tendo como principais a `MabLexer` e a `MabParser`. Durante o desenvolvimento do trabalho percebeu-se que o compilador necessitava de classes auxiliares para facilitar a geração de código intermediário.

Na última fase do desenvolvimento do compilador foi implementada a geração de código para cada máquina virtual utilizando a tradução dirigida pela sintaxe. Para gerar código LVIS foram criados atributos auxiliares na classe `MabParser`, pois o LVIS é um código intermediário de três endereços, enquanto a geração de código foi implementada na

---

[3] *Docstrings* são comentários informados no início de um programa, em uma função, em uma classe ou em um método, não sendo ignorados pelo interpretador Python, como os demais tipos de comentários (LUTZ; ASCHER, 2004, p. 189).

ordem pós-fixa. Já para geração do código MSIL não foram necessários tantos atributos de controle, porque a CLR é uma máquina de pilha e o MSIL segue a notação pós-fixa.

Códigos intermediários em LVIS e em MSIL gerados pelo compilador a partir de um programa escrito (Quadro 42) na linguagem MAB são apresentados nos Quadro 43 e Quadro 44, respectivamente. O programa faz o cálculo do fatorial de um número.

Quadro 42 – Programa fatorial na linguagem MAB

```
# PROGRAMA FATORIAL #

function main(){
  println 'FATORIAL'
  int fat = 1
  int n = 6
  while (n > 0){
    fat = fat * n
    n = n - 1
  }
  println 'RESULTADO'
  println tostr(fat)
}
```

Quadro 43 – Programa na linguagem LVIS

```
1    @expression1 = internal constant [9 x i8] c"FATORIAL\00"
2    @expression10 = internal constant [10 x i8] c"RESULTADO\00"
3    @str = internal constant [4 x i8] c"%d\0A\00"
4    @str_read = internal constant [2 x i8] c"%s"
5    declare i32 @printf(i8*, ...) nounwind
6    declare i32 @scanf(i8*, ...)
7    declare i32 @puts(i8* nocapture) nounwind
8
9    define i32 @main(){
10   %expression2 = getelementptr [9 x i8]* @expression1 , i64 0, i64 0
11   call i32 @puts (i8* %expression2)
12   %fat = alloca i32
13   store i32 1, i32* %fat
14   %n = alloca i32
15   store i32 6, i32* %n
16
17   br label %while1
18   while1:
19   %expression3 = load i32* %n
20   %expression4 = icmp ugt i32 %expression3, 0
21   br i1 %expression4, label %whileblk1, label %whilefim1
22   whileblk1:
23   %expression6 = load i32* %fat
24   %expression7 = load i32* %n
25   %expression5 = mul i32 %expression6, %expression7
26   store i32 %expression5, i32* %fat
27   %expression9 = load i32* %n
28   %expression8 = sub i32 %expression9, 1
29   store i32 %expression8, i32* %n
30
31   br label %while1
32   whilefim1:
33   %expression11 = getelementptr [10 x i8]* @expression10 , i64 0, i64 0
34   call i32 @puts (i8* %expression11)
35   %expression12 = load i32* %fat
36   call i32 (i8*, ...)* @printf(i8* getelementptr ([4 x i8]* @str, i32 0, i32
37   0), i32 %expression12)
38   ret i32 0
39   }
```

Quadro 44 – Programa na linguagem MSIL

```
1    .assembly programa {}
2
3    .method static void main() cil managed {
4    .entrypoint
5    ldstr "FATORIAL"
6    call void [mscorlib]System.Console::WriteLine(string)
7    .locals init (int32 fat)
8    ldc.i4 1
9    stloc fat
10   .locals init (int32 n)
11   ldc.i4 6
12   stloc n
13   br.s IL_3
14   IL_2: nop
15   ldloc fat
16   ldloc n
17   mul
18   stloc fat
19   ldloc n
20   ldc.i4 1
21   sub
22   stloc n
23   IL_3: ldloc n
24   ldc.i4 0
25   cgt
26   brtrue.s IL_2
27   ldstr "RESULTADO"
28   call void [mscorlib]System.Console::WriteLine(string)
29   ldloc fat
30   call string [mscorlib]System.Convert::ToString(int32)
31   call void [mscorlib]System.Console::WriteLine(string)
32   ret
33   }
```

Fazendo uma comparação dos dois códigos gerados, observa-se que na notação pós-fixa da MSIL (Quadro 44) para executar, por exemplo, uma operação de multiplicação (`mul` na linha 17) ou subtração (`sub` na linha 21) é necessário empilhar valores (linhas 15 e 16, ou linhas 19 e 20) e realizar a operação, que empilha o resultado. Na LVIS (Quadro 43) este tipo de operação (linhas 25 e 28) armazena o resultado em uma variável temporária. Mas caso seja necessário armazenar um valor (de uma constante ou de uma variável temporária) em uma variável declarada no programa ou usar o valor de uma variável declarada no programa, deve-se utilizar os comandos `load` (linha 23) e `store` (linha 26).

Foram identificadas algumas umas semelhanças entre os códigos. Em ambos é necessário declarar a função `main`, a primeira função a ser executada pela máquina virtual, a partir da qual pode-se invocar outras funções. Além disso, tanto em LVIS quanto em MSIL a chamada de funções é feita com o comando `call` seguido do tipo e do identificador da função desejada.

Também foi observado uma diferença no tamanho dos códigos gerados, nos programas testados, o código intermediário MSIL ficou mais extenso que o código equivalente LVIS,

isto devido o MSIL utilizar notação pós-fixada. No processo de geração de código e execução dos programas nas máquinas virtuais, não houve diferença significativa entre LLVM e CLR.

Os tipos de dados suportados pela linguagem MAB são `int` (para valores inteiros), `float` (para valores decimais), `str` (para cadeias de caracteres) e `bool` (para valores lógicos). Foi escolhido para cada caso um tipo equivalente nas linguagens intermediárias (Quadro 45).

Quadro 45 – Tipos de dados

linguagem MAB	LVIS	MSIL
int	i32	int32
float	Double	float32
str	i8*	string
bool	i1	bool

De maneira geral, o código MSIL ficou menos extenso que o código LVIS. Isso acontece devido ao LVIS ser um código de três endereços, sendo necessário criar variáveis temporárias e realizar alocação de memória. No caso do MSIL, utiliza-se sempre a pilha como estrutura de dados para armazenar os valores temporários, o que, para algumas situações, reduz a quantidade de comandos gerados.

Com relação aos trabalhos correlatos, o compilador desenvolvido é mais semelhante aos dois compiladores que geram código para plataforma .NET (TOMAZELLI, 2004 e LEYENDECKER, 2005), porém ambos trabalhos somente geram código para máquina virtual CLR, enquanto o tradutor de Cantú (2008) só gera código para a máquina LLVM e o Pymothoa estende a linguagem de programação Python com compilação JIT. No Quadro 46 é apresentada uma comparação entre o compilador desenvolvido e os trabalhos correlatos apresentados.

Quadro 46 – Comparativo dos trabalhos correlatos com o compilador desenvolvido

característica / ferramenta	CANTU (2008)	TOMAZELLI (2004)	LEYENDECKER (2005)	Pymothoa	compilador MAB
multiplataforma	X				
linguagem estruturada		X		X	X
linguagem orientada a objetos	X		X	X	
código intermediário em MSIL		X	X		X
código intermediário em LVIS	X				X
compilação JIT				X	

4 CONCLUSÕES

Atualmente existem várias tecnologias e plataformas de desenvolvimento afim de facilitar o trabalho dos programadores, focando em características a favor da legibilidade, da reusabalidade e da portabilidade de código, além do tempo de desenvolvimento. Ao tratar da questão da portabilidade e da reusabilidade, pode-se gerar código intermediário, um código independente de plataforma, para uma máquina virtual abstrata.

Este trabalho apresenta o desenvolvimento de um ambiente de compilação simplificado para a linguagem de programação MAB, que gera código intermediário para as máquina virtuais LLVM e CLR. Apesar da linguagem MAB não possuir todos os recursos das linguagens de alto nível atuais, é possível desenvolver programas na linguagem e obter os códigos intermediários em LVIS e em MSIL equivalentes. As construções sintáticas especificadas foram suficientes para identificar as instruções específicas e funcionamento das máquinas virtuais LLVM e CLR. Observa-se que foi necessário um estudo detalhado da fase de geração de código intermediário do compilador, já que as máquinas virtuais possuem tipos distintos de código. O código intermediário gerado para máquina LLVM (LVIS) é um código que utilizada o conceito de máquina de três endereços, já a máquina CLR (MSIL) é uma máquina de pilha. A principal dificuldade em desenvolver o compilador para as duas máquinas foi que, além de executar as análises léxica, sintática e semântica independente da máquina alvo, foi necessário gerenciar a árvore do código gerado para cada linguagem, sendo que cada máquina utiliza uma técnica distinta da outra.

A grande vantagem de gerar código intermerdiário para uma máquina virtual é que este código é independente de plataforma e também pode sofrer otimizações em sua geração. A desvantagem deste processo é o custo de processamento que a máquina virtual precisa para interpretar o código intermediário.

As bibliotecas utilizadas para desenvolver o compilador foram importantes para a conclusão do trabalho. Com o PLY foi possível fazer as definições regulares da linguagem, especificar as regras sintáticas, implementar as validações semânticas necessárias e a geração do código intermediário. Com o uso do PyQT diminui-se o tempo de desenvolvimento da interface gráfica do compilador, pois é uma biblioteca simples de ser utilizada, com documentação acessível e totalmente escrita em Python, facilitando a integração com as demais classes implementadas. Também foi utilizada a ferramenta QT Designer para desenhar a interface do compilador, agilizando e facilitando esta etapa do desenvolvimento

4.1 EXTENSÕES

O trabalho desenvolvido alcançou os objetivos propostos. Porém, sugere-se as seguintes extensões:

a) estender a linguagem MAB adicionando estruturas de dados homogêneas e suporte à programação orientada a objetos;
b) disponibilizar o compilador para a plataforma Windows;
c) possibilitar a utilização de funções nativas das máquinas virtuais LLVM e CLR;
d) implementar a geração de código para uma máquina real;
e) permitir importar funções de outro(s) arquivo(s) da linguagem MAB.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V. et al. **Compiladores:** princípios, técnicas e ferramentas. 2. ed. Tradução Daniel Vieira. São Paulo: Pearson Addison-Wesley, 2008.

ASTAH COMMUNITY. **It's time to try Astah for free!** [S.l.], 2013. Disponível em: <http://astah.net/editions/community>. Acesso em: 12 abr. 2013.

BEAZLEY, David M. **PLY (Python Lex-Yacc)**. [S.l.], [2011?]. Disponível em: <http://www.dabeaz.com/ply/ply.html>. Acesso em: 10 abr. 2013.

CANTÚ, Eduardo M. **Geração de código para máquina virtual LLVM a partir de programas escritos na linguagem de programação Java (tradutor Java - LLVM)**. 2008. 41 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis. Disponível em: <https://projetos.inf.ufsc.br/arquivos_projetos/projeto_839/tcc-eduardo-mello-cantu-2008-2-final.pdf>. Acesso em: 08 abr. 2013.

CONTINUUM ANALYTICS. **Examples and LLVM tutorials**: a simple function. [S.l.], 2010. Disponível em: <http://www.llvmpy.org/llvmpy-doc/0.9/doc/firstexample.html>. Acesso em: 03 abr. 2013.

CORDEIRO, Marcelo C. **Arquitetura da máquina virtual .NET.** [S.l.], 2005. Disponível em: <http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/002092-net.pdf>. Acesso em: 04 jun. 2013.

DELAMARO, Márcio E. **Como construir um compilador utilizando ferramentas Java**. 4. ed. São Paulo: Novatec, 2004.

GARDNER, Jonathan E. **PyQT tutorial**. [S.l.], [2013]. Disponível em: <https://wiki.python.org/moin/JonathanGardnerPyQtTutorial>. Acesso em: 15 nov. 2013.

GESSER, Carlos E. **GALS:** Gerador de Analisadores Léxicos e Sintáticos. [S.l.], [2002]. Disponível em: <http://gals.sourceforge.net/help.html>. Acesso em: 4 jun. 2013.

LARABEL, Michael. **Pymothoa**: a JIT extension to Python. [S.l.], 2012. Disponível em: <https://code.google.com/p/pymothoa/>. Acesso em: 05 maio 2013.

LATTNER, Chris. **llvm-as**: LLVM assembler. [S.l.], 2013a. Disponível em: <http://llvm.org/docs/CommandGuide/llvm-as.html>. Acesso em: 04 jun. 2013.

______. **LLVM:** language reference manual. [S.l.], 2013b. Disponível em: <http://llvm.org/docs/LangRef.html>. Acesso em: 05 jun. 2013.

LATTNER, Chris. **The arquiteture of open source applications:** elegance, evolution, and a few fearless hacks. [S.l.], 2013c. Disponível em: <http://www.aosabook.org/en/llvm.html>. Acesso em: 04 abr. 2013.

LATTNER, Chris; ADVE, Vikram. **The LLVM instrution set and compilation strategy**. [S.l.], 2002. Disponível em: <http://llvm.org/pubs/2002-08-09-LLVMCompilationStrategy.pdf>. Acesso em: 14 abr. 2013.

LEYENDECKER, Gustavo Z. **Especificação e compilação de uma linguagem de programação orientada a objetos para a plataforma Microsoft .NET**. 2005. 88 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://campeche.inf.furb.br/tccs/2005-II/2005-2gustavozadroznyleyendeckervf.pdf>. Acesso em: 04 jun. 2013.

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

LUTZ, Mark; ARCHER, David. **Aprendendo Python**. São Paulo: Bookman, 2004.

MARTINS, Joyce. **Apostila.doc**. Blumenau, 2013. Não paginado. Notas de aula. 1 arquivo (518 kbytes). Word 2010.

MICROSOFT. **MSDN library**: opCodes. [S.l.], 2013. Disponível em: <http://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.aspx>. Acesso em: 05 jun. 2013.

MOOLENAAR, Bram. **VIM:** an open-source text editor. [S.l.], [2012?]. Disponível em: <http://www.vim.org/>. Acesso em: 01 jun. 2013.

PRICE, Ana M. A.; TOSCANI, Simão S. **Implementação de linguagens de programação:** compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

TOMAZELLI, Giancarlo. **Implementação de um compilador para uma linguagem de programação com geração de código Microsoft .NET Intermediate Language**. 2004. 83 f. Trabalho de Conclusão de curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://campeche.inf.furb.br/tccs/2004-I/2004-1giancarlotomazellivf.pdf>. Acesso em: 04 jun. 2013.

WENNBORG, Hans. **Emulator speed-up using JIT and LLVM**. [S.l.], 2010. Disponível em: <http://www.llvm.org/pubs/2010-01-Wennborg-Thesis.pdf>. Acesso em: 18 abr. 2013.