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

Guilherme Luís Maba

Joyce Martins

Roteiro

- Introdução
- Objetivos
- Fundamentação teórica
- Desenvolvimento
- Operacionalidade
- Resultados e discussões
- Conclusões
- Extensões

Introdução

- A partir da necessidade de escrever programas em um alto nível de abstração, sem criar diretamente código de máquina, foram desenvolvidas as linguagens de alto nível.

- Estudo de máquinas virtuais e geração de código.

- Especificação de uma linguagem própria.

Objetivos

Desenvolver um compilador para uma linguagem de alto nível, gerando código LVIS e MSIL.

- Os objetivos específicos são:
  - efetuar as análises léxica, sintática e semântica da linguagem de alto nível, detectando e diagnosticando erros de compilação;
  - traduzir os programas na linguagem de alto nível para código da máquina virtual LLVM;

Objetivos

- traduzir os programas na linguagem de alto nível para código da máquina virtual CLR;
- possibilitar a execução dos códigos gerados;
- comparar os códigos gerados.

Fundamentação Teórica

- Compiladores
  - Análise léxica, sintática e semântica
  - Geração de código intermediário
    - Código de três endereços
    - Notação pós-fixada

- LLVM (*Low Level Virtual Machine*)

- CLR (*Common Language Runtime*)

Análise léxica, sintática e semântica

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

número = [0-9][0-9]*
```

Código de três endereços

- Três variáveis ou três endereços de memória, sendo as instruções compostas por operações binárias ou unárias.

```
//expressão em C
X = A + B * 10;

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

- Quando possui expressões com mais de uma operação, estas expressões são representadas em duas ou mais instruções.

Notação pós-fixada

- Projetada para uma máquina hipotética de pilha, usando uma memória de código, uma memória para as variáveis e uma pilha para dados temporários.

```
//expressão em C
X = A + B * 10;

//representação em notação polonesa
lod a
lod b
lda 10
mpi
adi
std x
```

LLVM

- Suporta linguagens com tipagem estática e dinâmica (Java, Ruby, Python).

- Código de três endereços: LLVM *Instruction Set* (LVIS).

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

CLR

- *Framework* .NET.

- Código intermediário em notação pós-fixada: MSIL (*MicroSoft Intermediate Language*).

```
.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 valores faz a soma
          //empilha resultado

  ret     //retorna valor que está na pilha
}
```

Trabalhos Correlatos

Tradutor - Java LLVM (CANTU, 2008)

- Entrada: arquivo `.class` de programas escritos na Linguagem Java.

- Saída: código correspondente para máquina LLVM.

- Ferramentas: GALS e BCEL (API em Java).

Trabalhos Correlatos

Linguagem sequencial imperativa (TOMAZELLI, 2004)

- Entrada: linguagem sequencial imperativa especificada, programas executados no *console*, *case-sensitive* e semelhante a linguagem C.

- Saída: código MSIL.

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

Trabalhos Correlatos

Linguagem orientada a objetos (LEYENDECKER, 2005)

- Entrada: linguagem de programação orientada a objetos especificada, com sobrecarga de métodos, herança simples, rotinas do *framework* .NET.

- Saída: código MSIL.

```
namespace org.furb.tcc;
using System.String;
using System.Console;
public class HelloWorld {
  public static void main(String[] args)
  {
    Console.WriteLine("Hello, World!");
  }
}
```

Trabalhos Correlatos

Pymothoa

- Entrada: extensão do linguagem Python.
- Saída: utiliza máquina virtual LLVM para compilação de suas funções.

```
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
```

Desenvolvimento

Requisitos Funcionais

- Possuir um editor para criar e manter programas na linguagem especificada.
- Permitir salvar arquivos texto com extensão `.mab`.
- Fazer as análises léxica, sintática e semântica do programa, detectando erros de compilação.
- Gerar código intermediário para máquina virtual LLVM e plataforma .NET.
- Gerar código executável a partir do código intermediário LVIS e MSIL.

Desenvolvimento

Requisitos não funcionais

- Ser implementado utilizando a linguagem de programação Python.
- Utilizar a biblioteca PLY para geração dos analisadores léxico e sintático.
- Utilizar a biblioteca PyQT para criação de interface gráfica.
- Utilizar o programa QT Designer para desenhar a interface gráfica do compilador.
- Executar no sistema operacional Linux.

Desenvolvimento

Especificação da linguagem MAB

- Linguagem estruturada projetada para executar na plataforma .NET e LLVM.
- As características da linguagem são:
  - ser fortemente tipada;
  - ser *case-sentive*;
  - permitir a declaração de funções;
  - possuir funções para leitura e escrita no console;
  - possuir funções de conversão de tipos;
  - ser utilizada para fins de pesquisa na geração de código intermediário.

Desenvolvimento

Especificação da linguagem MAB – Definições Regulares

- Especificação dos *tokens*.

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

- Nova linha (\n), tabulação horizontal (\t) e retorno do cursor (\r) são ignorados.

Desenvolvimento

Especificação da linguagem MAB – Palavras reservados

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

Desenvolvimento

Especificação da linguagem MAB – 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	não

Desenvolvimento

Especificação da linguagem MAB – Gramática

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

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

<delimiter_function> ::= { <block_function> }

<block_function> ::= <cmd_f> <block_function> | <empty>
<cmd_f> ::= <assign> | <cmds_general>

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

<read> ::= read ( IDENTIFIER )
```

Desenvolvimento

Especificação da linguagem MAB – Verificações semânticas

- No comando de atribuição, a variável e a expressão devem ser do mesmo tipo.
- No comando `read`, a variável deve ser do tipo `str`.
- Nos comandos `print` e `println`, as variáveis devem ser do tipo `str`.
- Qualquer identificador (de função ou de variável) só pode ser utilizado se já foi declarado previamente.

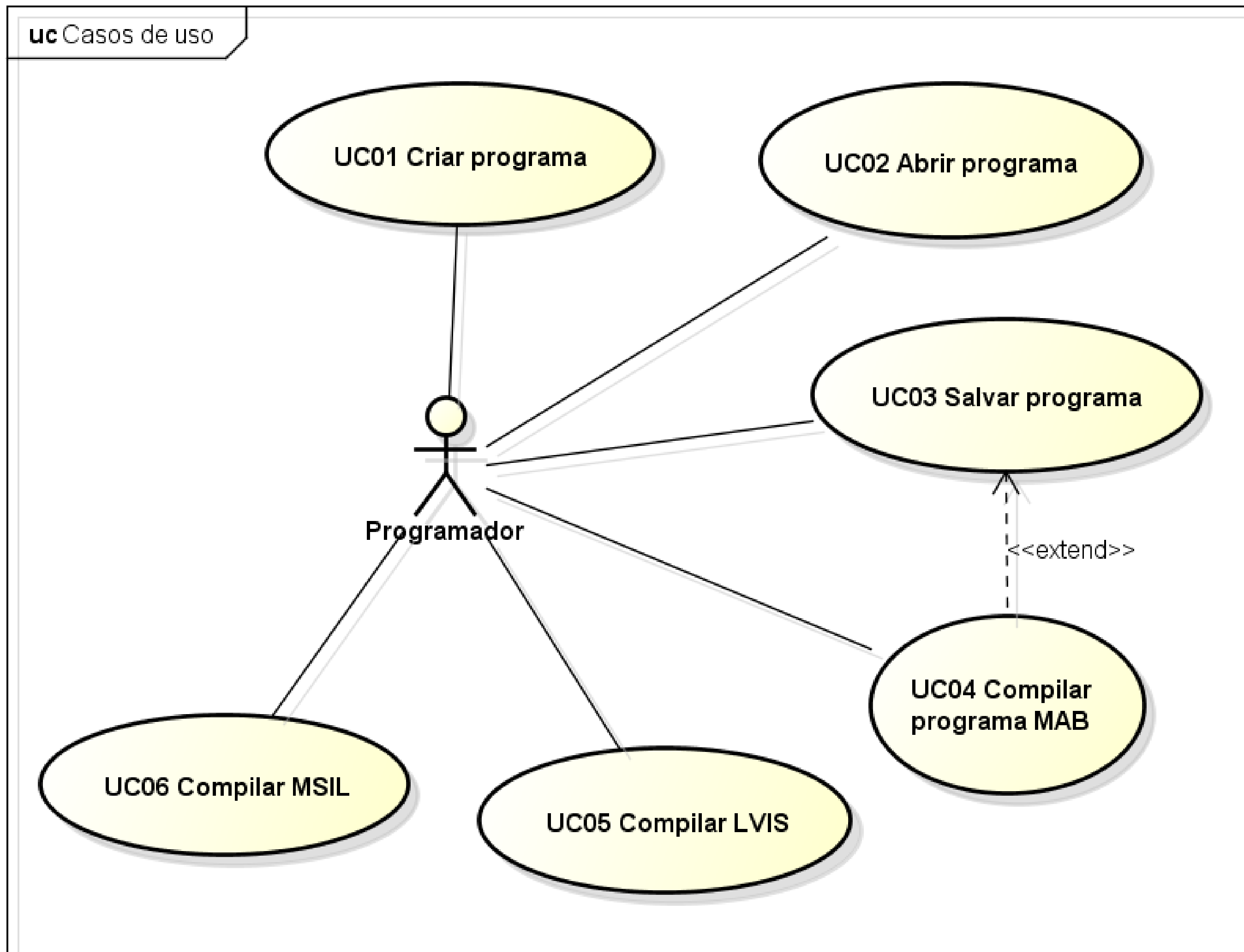
Desenvolvimento

Especificação da linguagem MAB – Verificações semânticas

- Nenhum identificador pode ser declarado mais de uma vez.
- Não é permitida atribuição de valores a uma variável constante, apenas quando da declaração.
- O tipo e a quantidade de argumentos quando da chamada de uma função deve ser igual ao tipo e à quantidade de parâmetros declarados (na mesma ordem).
- O valor de retorno de uma função deve ser do mesmo tipo da função.
- Não é possível converter valores do tipo lógico em valores dos tipos `str`, `float` e `int`.

Especificação

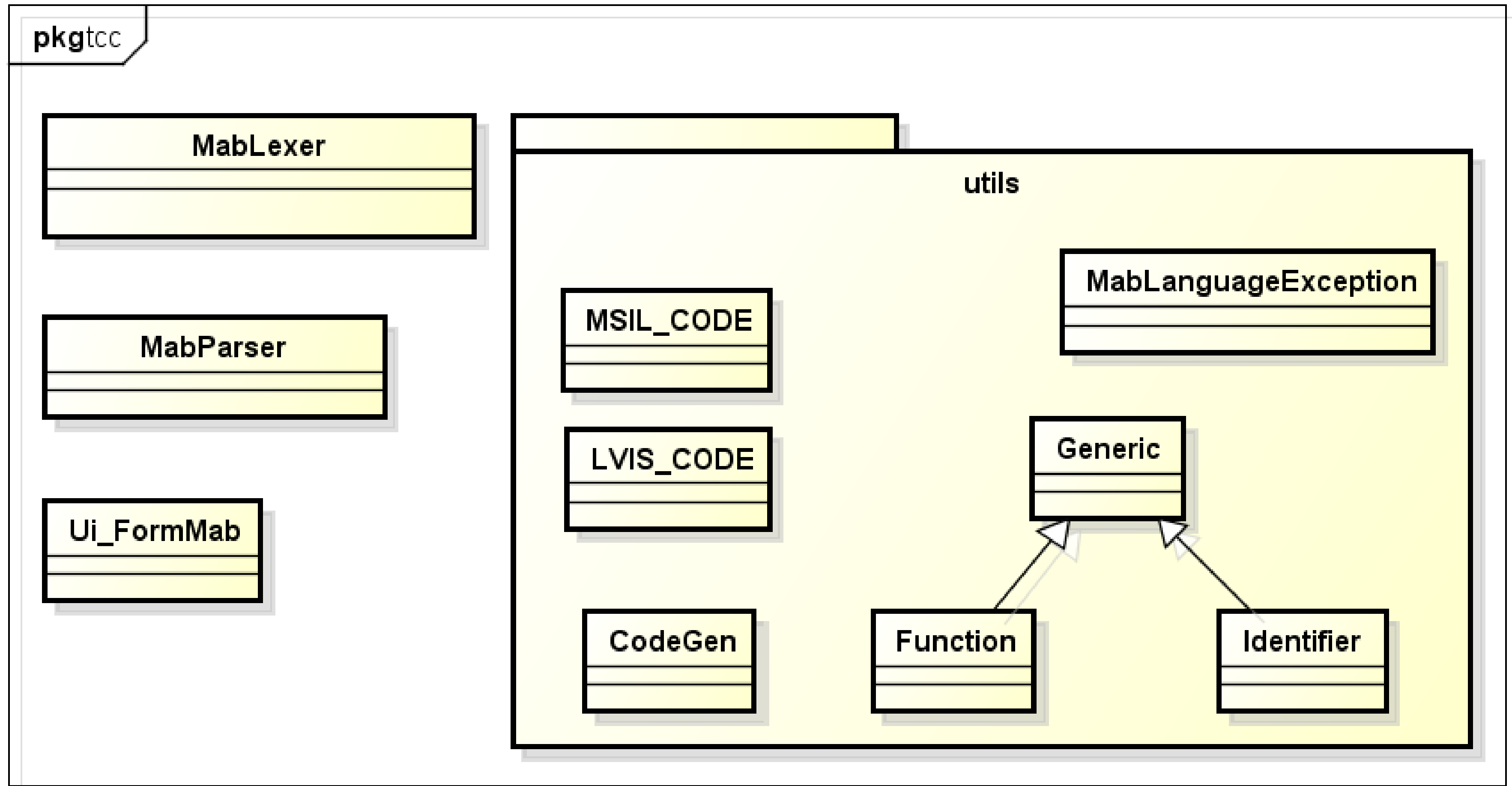
Diagrama de Casos de Uso



powered by Astah

Especificação

Diagrama de Classes



Implementação

Analisador Léxico

```
class MabLexer:
    ...
    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
    ...
```

Implementação

Analisador Sintático

```
class MabParser:
…
    def p_read(self, p):
    '''read : READ PAR_OPEN IDENTIFIER PAR_CLOSE'''
        if self.is_first_parse():
            pass
        else:
            # geração de código
…
```

Implementação

Analisador Semântico

```
...
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()) +")")
...
```

Implementação

Classes e tratamento de erro

```
class MabParser:

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

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

Implementação

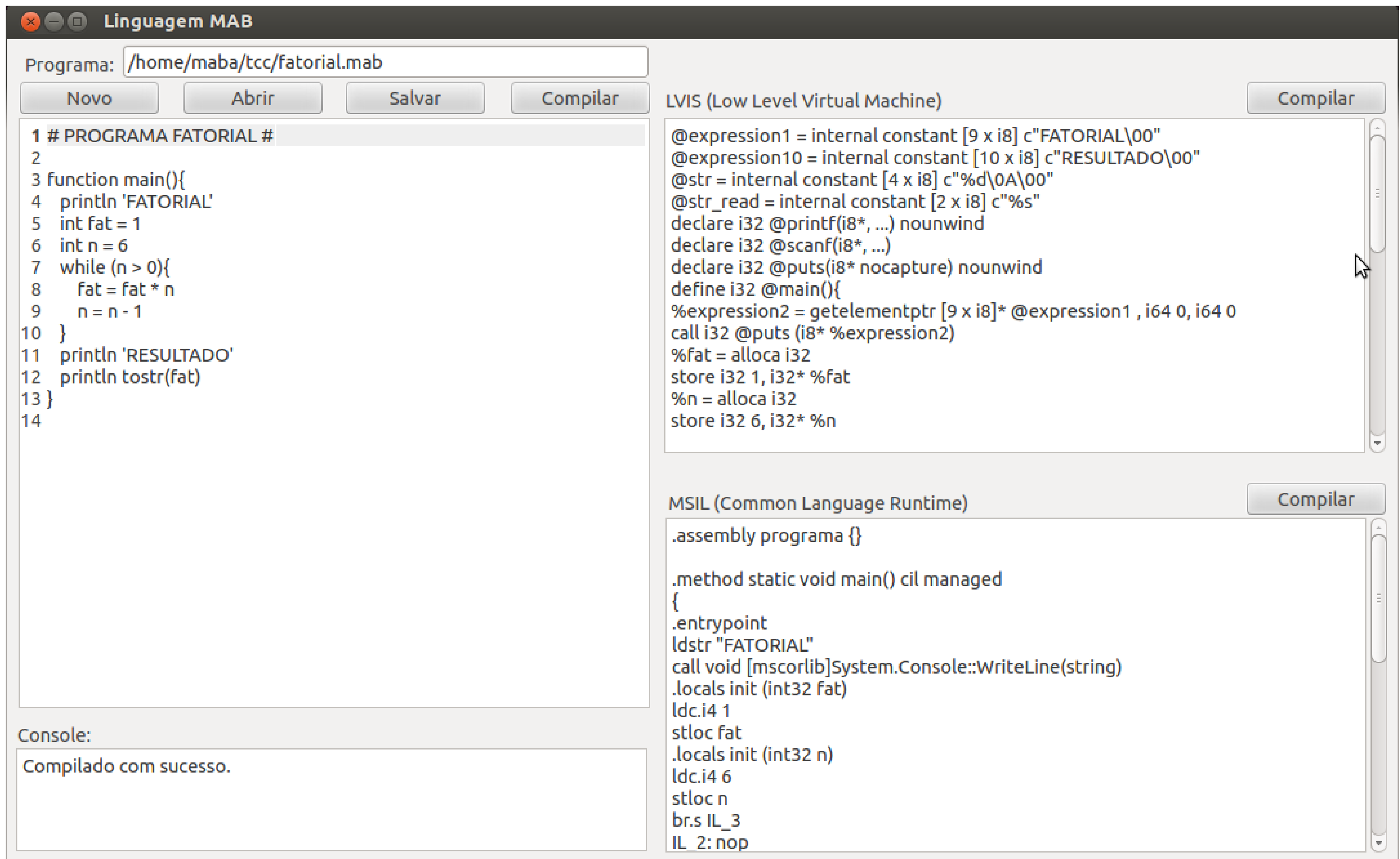
Geração de código intermediário

```
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)

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

Operacionalidade

Compilador MAB



Resultados e discussões

- Códigos gerados: MSIL e LVIS.

```
%expression6 = load i32* %fat
%expression7 = load i32* %n
%expression5 = mul i32 %expression6, %expression7
```

```
ldloc fat
ldloc n
mul
```

Resultados e discussões

```
%fat = alloca i32
store i32 1, i32* %fat

%expression6 = load i32* %fat
```

```
.locals init (int32 fat)

ldc.i4 1
stloc fat
ldloc fat
```

- Semelhanças: função `main` e chamada de função (`call`).

Resultados e discussões

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	

Conclusões

- Códigos intermediários em LVIS e em MSIL equivalentes.
- Estudo detalhado da fase de geração de código intermediário do compilador.
- Compilador para as duas máquinas.
  - Três endereços (LVIS)
  - Pós-fixada (MSIL)
- Bibliotecas importantes: PLY, PyQt e QT Designer.

Extensões

- Estender a linguagem MAB adicionando estruturas de dados homogêneas e suporte à programação orientado a objetos.
- Homologar o compilador para a plataforma Windows.
- Possibilitar a utilização de funções nativas das máquinas virtuais LLVM e CLR.
- Permitir importar funções de outro(s) arquivo(s) da linguagem MAB.

Apresentação do Aplicativo