

# Primeiro parte Trabalho de Compiladores Relatório

Guilherme Borges Oliveira  
guilhermeborges@comp.ufu.br

Tácio Silva Medeiros  
taciomedeiros@comp.ufu.br

Faculdade de Computação  
Universidade Federal de Uberlândia

8 de novembro de 2012

# Sumário

<b>1</b>	<b>Máquina Virtual</b>	<b>3</b>
1.1	Introdução . . . . .	3
1.2	Motivação . . . . .	3
1.3	Arquitetura . . . . .	4
1.4	Instruções . . . . .	6
1.5	Instalação . . . . .	10
<b>2</b>	<b>Traduzindo Programas</b>	<b>12</b>
2.1	MSIL Assembler - Ilasm . . . . .	12
<b>3</b>	<b>Exercícios</b>	<b>13</b>
3.1	Conversão de MiniC para Código da Máquina Virtual CLI-MSLI	13
3.1.1	Ex1.tsc . . . . .	13
3.1.2	Ex2.tsc . . . . .	14
3.1.3	Ex3.tsc . . . . .	15
3.1.4	Ex4.tsc . . . . .	16
3.1.5	Ex5.tsc . . . . .	17
3.1.6	Ex6.tsc . . . . .	18
3.1.7	Ex7.tsc . . . . .	20
3.1.8	Ex8.tsc . . . . .	22
3.1.9	Ex9.tsc . . . . .	24
3.1.10	Ex10.tsc . . . . .	26
3.1.11	Ex11.tsc . . . . .	30
3.1.12	Ex12.tsc . . . . .	34
3.1.13	Ex13.tsc . . . . .	36
3.1.14	Ex14.tsc . . . . .	39
<b>4</b>	<b>Analisador Lexical</b>	<b>43</b>
4.1	fslex . . . . .	43
4.2	Especificação do Analisador Léxico . . . . .	44
4.3	Código para criação do analisador . . . . .	45

<b>5</b>	<b>Teste do Analisador</b>	<b>50</b>
5.1	Relatório de saidas . . . . .	50
5.2	Erros lexicais . . . . .	52
5.2.1	Caracter ilegal . . . . .	52
5.2.2	Comentário nao fechado . . . . .	53
5.2.3	String inacabada . . . . .	54
<b>6</b>	<b>Análise Sintática</b>	<b>55</b>
6.1	Introdução . . . . .	55
6.2	Gramática . . . . .	56
6.2.1	Parser.fsy . . . . .	56
6.3	Erros Sintáticos . . . . .	62
6.4	Árvore - ASA - Ast . . . . .	63
6.4.1	Ast.fs . . . . .	63
6.5	Execução . . . . .	65
<b>7</b>	<b>Análise Semântica</b>	<b>66</b>
7.1	Introdução . . . . .	66
7.2	Implementação . . . . .	66
7.3	Principais problemas enfrentados . . . . .	67
<b>8</b>	<b>Referências Bibliográficas</b>	<b>68</b>

# Capítulo 1

## Máquina Virtual

### 1.1 Introdução

A Common Language Infrastructure - CLI, é uma especificação aberta desenvolvida pela Microsoft e padronizada pela ECMA[1], uma associação internacional responsável por ditar padrões quanto a informação e a tecnologia da informação [2]. Esta especificação permite rodar aplicações escritas em uma linguagem de alto nível em diferentes ambientes e sistemas sem a necessidade de reescrever as aplicações para levar em consideração as características exclusivas de cada um. Desta maneira, cria-se um ambiente o qual permite a "utilização" de várias linguagens de alto nível em diferentes plataformas sem a necessidade de serem reescritas para uma determinada arquitetura[3].

O CLI descreve o código executável e o ambiente de execução que formam o núcleo do Framework .NET da Microsoft, do MONO e do Portable.NET [4], sendo os dois ultimos implementações gratuitas e de código aberto da mesma[5]. A Common Language Runtime é a implementação da CLI pela Microsoft, em outras palavras, é uma máquina virtual que segue um padrão internacional e a base para a criação e execução de ambientes de desenvolvimento em que as linguagens e as bibliotecas trabalham juntos [6].

### 1.2 Motivação

Neste trabalho foi utilizado a Máquina Virtual CLR, inclusa no .NET framework, pois, apesar do MONO e ROTTOR serem projetos de código aberto que "escrevem" uma CLR para outros sistemas operacionais, o que os tornam mais atrativos, não se comparam com o .NET framework, uma ferramenta fechada apenas para o Windows que implementa varias outras bibliotecas

e frameworks que não são englobados nestes dois projetos. Portanto a motivação desta escolha se deve ao maior ponto negativo ser a restrição de plataforma, ser necessariamente o Windows, a qual não é um problema, incluindo assim sua gama de vantagens.

## 1.3 Arquitetura

Para uma melhor compreensão sobre a maquina virtual é necessário alguns conceitos principais sobre a CLI, os quais também são implantados na CLR, descritos a seguir:

### 1. Common Type System (CTS)

O sistema de tipo comum, ou CTS, especifica diretrizes de como os tipos (datatype) são declarados, usados, e gerenciados em tempo de execução, sendo também uma parte importante no suporte à integração de linguagens, sendo estas de diferentes paradigmas ou não. O CTS executa as seguintes funções:

- Estabelece um framework que ajuda a permitir a integração entre linguagens, segurança, e a alta performance da execução do código.
- Disponibiliza um modelo orientado a objetos que suporta a implementação completa de várias linguagens de programação.
- Define regras que as linguagens devem seguir, o que ajuda a garantir que objetos escritos em linguagens diferentes se interagem uns com os outros.[7]

### 2. Common Language Specification (CLS)

Para interagir totalmente com outros objetos, independentemente da linguagem em que foram implementados, os objetos devem expor aos chamadores somente os recursos que são comuns a todas as linguagens que devem interoperar. Por esta razão, a especificação da linguagem comum - CLS, que é um conjunto de características de linguagem básicas necessárias por diversas aplicações, foi definida.

As regras CLS definem um subconjunto do CTS, ou seja, todas as regras que se aplicam ao CTS aplicam-se ao CLS, exceto onde regras mais rígidas são definidas, no CLS, o que ajuda a melhorar e a garantir a interoperabilidade entre linguagens.[8]

### 3. Metadata

Conhecido como dados sobre dados. Um mecanismo entre varias ferramentas, como compiladores e debuggers, e a Virtual Execution System (VES). Define metadados para os tipos de dados do CTS. As informações são armazenadas em METADATA dentro de cada programa no momento da compilação. São descrição dos tipos (classes, estruturas, tipos enumerados, etc) usado na aplicação, podendo esta ter sido gerada em forma de DLL ou executável. Descrição dos membros (propriedades, métodos, eventos etc.) Descrição de cada unidade de código externo (assembly) usada na aplicação e que é requerida para que esta execute adequadamente. Nos metadados há a versão de cada aplicação .NET, harmônicas, que vivem no mesmo ambiente, evitando o conflito entre as aplicações. A CLI, no caso do windows a Common Language Runtime - CLR, procura nos metadados a versão correta da aplicação a ser executada.[9]

### 4. Virtual Execution System (VES)

O Virtual Execution System - VES, ou sistema de execução virtual, é um processo de compilação e onde o compilador just in time - JITTER, converte as instruções da Intermediate Language (o assembly) para instruções específicas da arquitetura do processador onde a aplicação .NET esta sendo executada.

Na CLR ele é ativado quando uma aplicação .NET é chamada. O windows identifica que esta é uma aplicação .NET e uma runtime Win32 passa o controle para a runtime do .NET. Neste momento a compilação do Portátil Executável (PE) é efetuada pela CLR, só então o código assembly próprio da arquitetura do processador é gerado para que a aplicação possa ser executada.

Os conceitos citados acima, e outros, serão melhores explicados analisando o fluxo de um programa executado sobre o .NET framework, o qual contém a maquina virtual CLR, explicado a seguir:

O código fonte, escrito em qualquer uma das linguagens suportadas (C#, F#, entre outras) é compilado para o assembly da plataforma, a Microsoft Intermediate Language (MSIL), estando este já dentro da especificação CLI, contendo metadados, especificações CTS e CLS, entre outras características.[6]

Durante o processo de compilação entrará em ação o montador MSIL Assembler (Ilasm), o qual através do arquivo em MSIL (extensão ".il") resultará em um arquivo PE, com extensão ".exe" ou ".dll". No PE é definida informações de cada método como as suas instruções, manipulações, assinatura, tipo e quantidade de retorno, ordem de parâmetros, tipos dos argumentos, array de tratamento de exceções, tamanho da pilha de execução que o método necessita, tamanho dos arrays locais, entre outras informações importantes.

Ao se executar o PE o mesmo é carregado para a CLR onde, após atender os requisitos de segurança, será executada a compilação just in time (JIT), responsável por converter este arquivo em instruções de máquina, ou seja, por ser responsável pelo carregamento de classes, verificação, compilação JIT e gerenciamento de código a máquina virtual CLR cria um ambiente para execução de código VES. O CLR também oferece serviços relacionados à coleta de lixo, tratamento de exceções e gerenciamento de recursos e tem como seus principais componentes o mecanismo que faz relação com os metadados, um carregador de classe, um verificador (responsável por testar as restrições de métodos, verificar o tamanho da pilha utilizada, entre outras funcionalidades) e o JITTER.[10]

## 1.4 Instruções

O assembly da plataforma, MSIL, é um conjunto de instruções baseado em pilha e orientado a objetos, suas funções em relação à manipulações com a pilha e as demais funcionalidades que não englobam o paradigma de orientações a objetos (como instância de um objeto, etc) estão listadas a seguir:

- add: retira os dois elementos do topo da pilha e coloca o resultado no topo.
- add.ovf: mesma operação que o add porém com uma verificação de overflow(gerando uma exceção de overflow).
- and: retira os dois elementos do topo da pilha, faz a operação and bit a bit e coloca o resultado no topo da pilha.
- arglist: pega argumento da lista (usado em para pegar argumentos de função)
- beq.[length]: branch para label se o dois valores da pilha são iguais

- bge.[length]: branch para label se o primeiro valor é menor que o segundo da pilha
- bge.un.[length]: branch para label se maior ou igual, comparação sem sinal
- bgt.[length]: branch para label quando o segundo valor é maior que o primeiro
- bgt.un.[length]: branch para label quando o topo é maior que o segundo valor, sem sinal
- ble.[length]: branch para label quando o topo é menor ou igual ao segundo valor
- ble.un.[length]: branch para label se topo é menor ou igual ao segundo valor, sem sinal
- blt.[length]: branch para label quando quando o topo é menor que o segundo valor
- blt.un.[length]: branch para label se topo é menor que segundo valor, sem sinal
- bne.un[length]: branch para label quando topo não for igual ou não ordenada
- br.[length]: branch incondicional
- break: instrução breakpoint
- brfalse.[length]: branch para label se falso, nulo, ou zero
- brtrue.[length]: branch para label quando não falseo ou não nulo
- call: chama um método
- calli: chama um método indireto
- ceq: compara se igual
- cgt: compara se maior que
- cgt.un: compara maior que, sem sinal e não ordenado
- ckfinite: Checa se é um número real e finito



- `clt`: compara se menor que
- `clt.un`: compara se menor que, sem sinal e não ordenado
- `conv.[to type]`: conversão de dados
- `conv.ovf.[to type]`: conversão de dados com detecção de overflow
- `conv.ovf.[to type].un`: conversão de dados sem sinal com detecção de overflow
- `cpblk`: copia dados da memória para a memória
- `div`: divide valores
- `div.un`: divide valores inteiros, sem sinal
- `dup`: duplica o valor do topo da pilha
- `endfilter`: fim do filtro da cláusula de SEH
- `endfinally`: finaliza a cláusula do bloco de exceção
- `initblk`: inicializa um bloco de memória para um valores
- `jmp`: pula para um método
- `jmp.i`: pula para um ponteiro de método
- `ldarg.[length]`: carrega um argumento na pilha
- `ldarga.[length]`: carrega um argumento a partir de um endereço
- `ldc.[type]`: carrega uma constante numérica
- `ldftn` : carrega um ponteiro de método
- `ldind.[type]`: carrega um valor indireto na pilha
- `ldloc`: load local variable onto the stack
- `ldloc`: carrega na pilha o valor da variavel local.
- `ldloc.i`: carrega na pilha o valor da variavel local com index.
- `ldnull`: carrega na pilha um ponteiro pra null
- `leave target`: sai de uma região protegida do código

- `localloc`: aloca um espaço na memory pool do tamanho do primeiro elemento da pilha e retorna o endereço da área.
- `mul`: multiplica os dois valores do topo da pilha (retirando-os) e coloca o resultado.
- `mul.ovf.[type]`: multiplica valores inteiros levando em conta o overflow.
- `neg`: retira o valor do topo da pilha, nega ele e põe o resultado no topo, retornando o mesmo tipo de operando.
- `nop`: faz nada :D
- `not`: retira um inteiro e coloca seu complemento (inversão de bits) na pilha.
- `or`: faz o OU bit a bit dos dois valores inteiros no topo da pilha (retirando-os) e coloca o resultado.
- `pop`: remove o elemento do topo da pilha.
- `rem`: computa o resto da divisão do valor abaixo do topo da pilha pelo valor que está no topo (retirando-os) e coloca o resultado no topo.
- `rem.un`: mesmo que o `rem` só que para inteiros unsigned.
- `ret`: retorna para o método corrente, o tipo de retorno do método em questão será o utilizado.
- `shl`: Deslocamento de inteiro para a esquerda...,  
*valor, qntd de deslocamento retira estes dois da pilha e coloca o resultado.*
- `shr`: mesmo que o `shl` porém com deslocamento para a direita.
- `shr.un`: mesmo que o `shr` só que para inteiros unsigned.
- `starg.[length]`: retira o elemento do topo da pilha e o coloca em um argumento (starg num).
- `stind.[type]`: coloca o valor (topo da pilha) no endereço (logo abaixo).
- `stloc`: retira o valor do topo da pilha e o põe na variável (stloc x)
- `sub`: subtrai do segundo valor o primeiro e põe o resultado no topo.
- `sub.ovf.[type]`: subtração de inteiros com overflow

- `tail.`: deve preceder imediatamente instruções de `call`, `calli` ou `callvirt`. Ela indica que o método corrente na pilha deve ser removido antes da chamada da função ser executada.
- `unaligned.` (prefix code): especifica que o endereço na pilha não está alinhado ao tamanho natural.
- `volatile.` (prefix code): especifica que o endereço no topo da pilha é volátil.
- `xor`: executa a operação XOR(bit a bit) entre os dois primeiros elementos da pilha colocando seu resultado na mesma.
- `ldem.[type]`: carrega um elemento do vetor(segunda posição) no index(primeira posição, topo da pilha) na pilha
- `ldelema`: põe o endereço do vetor na posição index na pilha.
- `ldlen`: põe na pilha o tamanho do vetor(topo da pilha)
- `ldstr`: põe a string(ldstr string) no topo da pilha
- `newarr`: cria um array com o tipo definido(`newarr int32`) onde seu tamanho está no topo da pilha.
- `sizeof`: carrega o tamanho em bits do tipo definido(`sizeof int32`) na pilha.
- `stelem.[type]`: coloca no array(terceiro da pilha) no index(segunda da pilha) o valor(topo da pilha).

Um documento com mais detalhes sobre estas funções pode ser encontrado [aqui](#)

## 1.5 Instalação

Microsoft .NET é uma iniciativa da Microsoft em que visa uma plataforma única para desenvolvimento e execução de sistemas e aplicações. Todo e qualquer código gerado para .NET, pode ser executado em qualquer dispositivo ou plataforma que possua o .NET Framework, a "Plataforma .NET".

A plataforma .NET é executada sobre uma máquina virtual, a Common Language Runtime - CLR, um ambiente de execução independente de linguagem, interagindo com uma coleção de bibliotecas unificadas, onde juntas

formam o framework. A CLR é capaz de executar uma grande quantidade de diferentes linguagens de programação, atualmente mais de quarenta[11], as quais se interagem entre si como se fossem uma única linguagem.

Para se obter a CLR é necessário estar em um ambiente windows. Como a CLR está contida no .NET Framework é necessária sua instalação. O framework utilizado neste trabalho é o Microsoft .NET Framework 4 e pode ser encontrado [aqui](#)

feito o download basta executá-lo e prosseguir normalmente com a instalação.

## Capítulo 2

# Traduzindo Programas

### 2.1 MSIL Assembler - Ilasm

O Ilasm é o Assembler da Máquina Virtual CLI na sua execução ele gera um arquivo portátil executável (PE), o qual contém MSIL e os metadados necessários para assegurar a performance esperada.

O Assembler é automaticamente instalado juntamente com o framework e pode ser encontrado no diretório onde foi instalado o Microsoft .NET framework dentro da pasta da versão correspondente. O diretório padrão é :

C:/Windows/Microsoft.NET/Framework64/(pasta versao)

Esté é o diretório do .NET 4.0, utilizado neste trabalho. Nele encontra-se o ilasm.exe. Para utilizá-lo basta seguir o seguinte comando dentro da pasta com o ilasm.exe:

ilasm meuarquivo.il

Este comando gerará o arquivo meuarquivo.exe no diretório em que se encontra o arquivo .IL

Quando compilado o arquivo .IL a saída gerada será um arquivo .exe, é possível nomear este arquivo usando o comando /output= "nome do arquivo".

o comando é similar ao anterior ilasm "nome do arquivo .IL<< /output = "nome da saída» a parte entre "menor que"e "maior que"é opcional caso não seja utilizado o padrão é que o nome seja igual ao do arquivo .IL com a extensão .exe[12]

# Capítulo 3

## Exercícios

### 3.1 Conversão de MiniC para Código da Máquina Virtual CLI-MSLI

#### 3.1.1 Ex1.tsc

MiniC

```
1  
2 int main() {}
```

MSLI

```
1  
2 .assembly extern mscorlib {}  
3  
4 .assembly Vazio  
5 {  
6     .ver 1:0:1:0  
7 }  
8 .module vazio.exe  
9  
10 .method static void main() cil managed  
11 {  
12     .maxstack 1  
13     .entrypoint  
14     ret  
15 }
```

---

### 3.1.2 Ex2.tsc

#### Mini C

```
1
2 int main()
3 {
4   int x;
5   x = 4;
6 }
```

#### MSLI

```
1 .assembly extern mscorlib {}
2
3 .assembly Atribui
4 {
5     .ver 1:0:1:0
6 }
7 .module atribui.exe
8
9 .method static void main() cil managed
10 {
11     .maxstack 1
12     .entrypoint
13
14         ldc.i4 4
15         stloc.0
16         ldloc.0
17         call void [mscorlib]System.Console::WriteLine(
18             int32)
19
20     ret
21 }
```

### 3.1.3 Ex3.tsc

#### Mini C

```
1
2 int main()
3 {
4   int x,y,z;
5
6   x = 2;
7   y = 5;
8   z = x + y;
9 }
10
11
12
13 }
```

#### MSLI

```
1
2 .assembly extern mscorlib {}
3
4 .assembly Soma
5 {
6     .ver 1:0:1:0
7 }
8 .module Soma.exe
9
10 .method static void main() cil managed
11 {
12     .locals init (int32,int32,int32)
13     .maxstack 4 // numero de posicoes da pilha
14     .entrypoint
15
16     ldc.i4 2
17     stloc.0 //tira a variavel do topo da pilha e
18             coloca em var local
19     ldc.i4 5
20     stloc.1
```



```

20
21     ldloc.0
22     ldloc.1
23     add
24     stloc.2
25
26
27     ldloc.2 // le a variavel local
28     //call void [mscorlib]System.Console::WriteLine
        (int32)
29
30
31     ret
32 }

```

### 3.1.4 Ex4.tsc

#### Mini C

```

1
2 int main()
3 {
4     int x,y,z;
5
6     x = 2;
7     y = 5;
8     z = x + y;
9     printf(" \\\%d\\",z);
10 }

```

#### MSLI

```

1 .assembly extern mscorlib {}
2
3 .assembly Soma
4 {
5     .ver 1:0:1:0
6 }
7 .module Soma.exe

```

```

8
9 .method static void main() cil managed
10 {
11     .locals init (int32,int32,int32)
12     .maxstack 4 // numero de posicoes da pilha
13     .entrypoint
14
15     ldc.i4 2
16     stloc.0 //tira a variavel do topo da pilha e
17             coloca em var local
18     ldc.i4 5
19     stloc.1
20
21     ldloc.0
22     ldloc.1
23     add
24     stloc.2
25
26     ldloc.2 // le a variavel local
27     //call void [mscorlib]System.Console::WriteLine
28             (int32)
29
30     ret
31 }

```

### 3.1.5 Ex5.tsc

#### Mini C

```

1
2 #include <string.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 int main ()
6 {
7     char s[50];
8
9     strcpy(s,"Alo_Mundo");

```

```

10 printf("\%s\n",s);
11 system("PAUSE");
12 }

```

## MSLI

```

1  .assembly extern mscorlib {}
2
3  .assembly Alo
4  {
5      .ver 1:0:1:0
6  }
7  .module ALo.exe
8
9  .method static void main() cil managed
10 {
11     .locals init (string)
12     .maxstack 4 // numero de posicoes da pilha
13     .entrypoint
14
15         ldstr "Alo mundo"
16         stloc.0
17
18         ldloc.0
19         call void [mscorlib]System.Console::WriteLine(
20             string)
21
22     ret
23 }

```

### 3.1.6 Ex6.tsc

#### Mini C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int x;

```

```

6 | x = 10 - 3*2;
7 | if (x == 4)
8 |     printf("Certo");
9 | else
10 |     printf("Errado");
11 | }

```

## MSLI

```

1 | .assembly extern mscorlib {}
2 |
3 | .assembly Condicional
4 | {
5 |     .ver 1:0:1:0
6 | }
7 | .module condicional.exe
8 |
9 | .method static void main() cil managed
10 | {
11 |     .locals init (int32)
12 |     .maxstack 4 // numero de posicoes da pilha
13 |     .entrypoint
14 |
15 |         ldc.i4 10
16 |
17 |         ldc.i4 3
18 |         ldc.i4 2
19 |
20 |         mul.ovf //instrução de multiplicação considerando
21 | overflow , gerando exceção
22 |
23 |         sub.ovf //instrução de subtração
24 |
25 |
26 |         stloc.0
27 |         ldloc.0
28 |
29 |         ldc.i4 4
30 |         beq EQUAL
31 |             ldstr "ERRADO"

```

```

32         call void [mscorlib]
33
34 System.Console::WriteLine(string)
35
36
37         br Exit
38 EQUAL:
39         ldstr "CERTO"
40         call void [mscorlib]
41
42 System.Console::WriteLine(string)
43
44 Exit:
45     ret
46 }

```

### 3.1.7 Ex7.tsc

#### Mini C

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main()
5 {
6     int x;
7     x = 0;
8     while (x<5)
9     {
10         x = x+1;
11     }
12
13     printf("x=%d\n",x);
14 }

```

#### MSLI

```

1 .assembly extern mscorlib {}
2

```

```

3  .assembly Repeticao
4  {
5      .ver 1:0:1:0
6  }
7  .module repeticao.exe
8
9  .method static void main() cil managed
10 {
11     .locals init (int32)
12     .maxstack 30
13     .entrypoint
14
15     ldc.i4 0
16     stloc.0
17
18
19     LOOP:
20         ldloc.0 //Coloca a variavel x na pilha
21         ldc.i4 4 //coloca o valor 5 na pilha
22
23         bgt CONTINUE //vai para continue se x > 4 (obs:
                esse safaadinho tira da
24
25 pilha)
26         ldloc.0
27         ldc.i4 1 //coloca o 1 na pilha
28         add //soma o x com o 1
29         stloc.0 //poe o resultado em x
30         br LOOP //volta para o loop
31
32     CONTINUE:
33         ldstr "x = "
34         call void [mscorlib]System.Console::Write(
                string)
35         ldloc.0
36         call void [mscorlib]System.Console::Write(int32
                )
37         ret
38 }

```

### 3.1.8 Ex8.tsc

#### Mini C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int n;
6     int fat;
7     printf(" Digite um numero:");
8     scanf("%d",&n);
9     fat = 1;
10
11     while(n > 0)
12     {
13         fat = fat *n;
14         n = n - 1;
15     }
16     printf("O fatorial de %d e %d",n,fat);
17 }
```

#### MSLI

```
1
2
3 .assembly extern mscorlib {}
4
5 .assembly Fatorial
6 {
7     .ver 1:0:1:0
8 }
9 .module fatorial.exe
10
11 .method static void main() cil managed
12 {
13     .locals init (int32,int32)
14     .maxstack 10
15     .entrypoint
16 }
```

```

17     ldstr "Digite um numero: "
18     call void [mscorlib]System.Console::WriteLine (
19         string)
20     call string [mscorlib]System.Console::ReadLine ()
21     call int32 [mscorlib]System.Int32::Parse(string)
22
23     stloc.0 // atribui valor ao n
24
25     ldc.i4 1 // coloca um na pilha
26     stloc.1 //grava a pilha em fat
27
28 LOOP:
29     ldloc.0 //carrega o valor de n
30
31     brfalse CONTINUE //verifica se n = 0
32     ldloc.0 //n na pilha
33     ldloc.1 //fat na pilha
34
35     mul.ovf // multiplica n pelo fat e poe o resultado
36         no topo
37     stloc.1 // grava em fatorial
38
39     ldloc.0 // poe n na pilha
40     ldc.i4 1 // poe um na pilha
41     sub.ovf // n - 1
42     stloc.0 // poe o resultado em n
43
44     br LOOP
45
46 CONTINUE:
47     ldstr "O fatorial de "
48     call void [mscorlib]System.Console::Write(
49         string)
50     ldloc.0
51     call void [mscorlib]System.Console::Write(int32
52         )
53     ldstr " e "
54     call void [mscorlib]System.Console::Write(
55         string)
56     ldloc.1

```



```

53         call void [mscorlib]System.Console::Write(int32
54             )
55
56     ret
57 }

```

### 3.1.9 Ex9.tsc

#### Mini C

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int main ()
4  {
5      int n;
6      printf("Digite um numero: ");
7      scanf("%d",&n);
8
9      if (n > 0)
10         printf("Numero Positivo");
11     else if (n == 0)
12         printf("Zero");
13     else printf("Numero Negativo");
14
15 }

```

#### MSLI

```

1  .assembly extern mscorlib {}
2
3  .assembly Positivo
4  {
5      .ver 1:0:1:0
6  }
7  .module positivo.exe
8
9  .method static void main() cil managed
10 {

```

```

11     .maxstack 10
12     .entrypoint
13
14
15
16     ldstr "Digite um numero: "
17     call void [mscorlib]System.Console::WriteLine (
18         string)
19     call string [mscorlib]System.Console::ReadLine ()
20     call int32 [mscorlib]System.Int32::Parse(string)
21 stloc.0
22
23     ldloc.0 // carrega o valor de n na pilha
24     ldc.i4 0
25
26     bgt POSITIVO
27     br SENAO
28
29
30 POSITIVO:
31     ldstr "NUMERO POSITIVO "
32     call void [mscorlib]System.Console::WriteLine (
33         string)
34     ret
35
36 SENAO:
37     ldloc.0
38     brtrue NEGATIVO
39     ldstr "ZERO"
40     call void [mscorlib]System.Console::WriteLine (
41         string)
42     ret
43
44 NEGATIVO:
45     ldstr "NUMERO NEGATIVO"
46     call void [mscorlib]System.Console::WriteLine (
47         string)
48
49     ret
50 }

```

### 3.1.10 Ex10.tsc

#### Mini C

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int x,y,z;
6     printf(" Digite tres numeros inteiros positivos: ");
7     scanf("%d\\%d\\%d", &x,&y,&z);
8
9     if (x< y+z && y < x+z && z<x+y)
10    {
11        if (x == y && y == z)
12            printf("Triangulo Equilatero");
13        else if (x == y || x == z || y == z)
14            printf("Triangulo Isoceles");
15        else if ( !(x==y) && !(x == z) && !(y == z))
16            printf("Triangulo Escaleno");
17        else
18            return;
19    }
20    else
21        printf("Essas medidas nao formam um triangulo")
22        ;
23 }
```

#### MSLI

```
1 .assembly extern mscorlib {}
2
3 .assembly Triangulo
4 {
5     .ver 1:0:1:0
6 }
7 .module triangulo.exe
8
9 .method static void main() cil managed
```

```

10 {
11     .maxstack 10
12     .entrypoint
13
14     ldstr "Digite tres numeros inteiros positivos: " //
        numero
15
16 depois enter
17     call void [mscorlib]System.Console::WriteLine (
        string)
18     call string [mscorlib]System.Console::ReadLine ()
19     call int32 [mscorlib]System.Int32::Parse(string)
20
21     stloc.0 // leu x
22
23     call string [mscorlib]System.Console::ReadLine ()
24     call int32 [mscorlib]System.Int32::Parse(string)
25
26 stloc.1 // leu y
27
28     call string [mscorlib]System.Console::ReadLine ()
29     call int32 [mscorlib]System.Int32::Parse(string)
30
31 stloc.2 //leu z
32
33 ldloc.0 //x
34 ldloc.1 // y
35 ldloc.2 // z
36 add // y+z
37 clt // 1 se x < y+z , 0 caso contrario
38 brfalse ELSE1 // pula para else se for 0
39 ldloc.1 //y
40 ldloc.0 //x
41 ldloc.2 //z
42 add // x+z
43 clt // 1 se y < x+z , 0 caso contrario
44 brfalse ELSE1 // pula para else se for 0
45 ldloc.2 //z
46 ldloc.0 // x
47 ldloc.1 //y
48 add // x+y

```

```

49 | clt // 1 se z < x + y 0 caso contrario se isso tudo for
50 |
51 | satisfeito ele entra no primeiro if
52 |
53 |
54 | //comeco segundo if (x == y && y == z)
55 | ldloc.0 //x
56 | ldloc.1 //y
57 | ceq // compara se x e y sao iguais e poe 1 na pilha do
58 |
59 | contrario 0
60 | brfalse ELSE2
61 | ldloc.1 //y
62 | ldloc.2 //z
63 | ceq // compara se y e z sao iguais
64 | brfalse ELSE2
65 | ldstr "Triangulo Equilatero"
66 | call void [mscorlib]System.Console::WriteLine (string)
67 | ret
68 |
69 |
70 |
71 |
72 | //comeco primeiro else if (x == y || x == z || y == z)
73 | ELSE2:
74 | ldloc.0 //x
75 | ldloc.1 //y
76 | ceq // se x e y forem iguais ele retorna 1, 0 caso
    | contrario
77 | brtrue ISOCELES // se for 1 ele e isoceles
78 | ldloc.0 //x
79 | ldloc.2 //z
80 | ceq
81 | brtrue ISOCELES
82 | ldloc.1 //y
83 | ldloc.2 //z
84 | ceq
85 | brtrue ISOCELES
86 |
87 |
88 | //caso nao va para isoceles continua o fluxo

```

```

89 //else if ( !(x==y) && !(x == z) && !(y == z))
90
91 ldloc.0 // x
92 ldloc.1 // y
93 ceq // se x e y forem iguais ele retorna 1, 0 caso
    contrario
94 brtrue ELSEFINAL // caso x == y vai para elsefinal
95 ldloc.0 //x
96 ldloc.2 //z
97 ceq
98 brtrue ELSEFINAL
99 ldloc.1 //y
100 ldloc.2 //z
101 ceq
102 brtrue ELSEFINAL
103
104     ldstr "Triangulo ESCALENO"
105     call void [mscorlib]System.Console::WriteLine (
        string)
106
107     ret
108
109 ELSEFINAL:
110
111     ret
112
113
114
115
116
117 ISOCELES:
118
119
120     ldstr "Triangulo isoceles"
121     call void [mscorlib]System.Console::WriteLine (
        string)
122
123     ret
124
125
126 ELSE1:

```

```

127
128     ldstr "Essas medidas nao formam um triangulo"
129     call void [mscorlib]System.Console::WriteLine (
        string)
130
131     ret
132 }

```

### 3.1.11 Ex11.tsc

#### Mini C

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5      int vet1[10], vet2[10], vet3[20];
6      int i, j;
7
8      j= 1;
9      i = 1;
10     while (i<10)
11     {
12         scanf("%d_", vet1[i]);
13         vet3[j] = vet1[i] ;
14         j = j+1;
15
16         scanf("%d_", vet2[i]);
17         vet3[j] = vet2[i] ;
18         j = j+1;
19         i = i+1;
20     }
21     i = 1;
22     while(i<20)
23     {
24         printf("%d_", vet3[i]);
25         i = i+1;
26     }
27 }

```

## MSLI

```
1  .assembly extern mscorlib {}
2
3  .assembly Intercala
4  {
5      .ver 1:0:1:0
6  }
7  .module Intercala.exe
8
9  .method static void main() cil managed
10 {
11     .locals init (int32 [] vet1,int32 [] vet2,int32 []
12     vet3,int32 i ,int32 j)
13     .maxstack 20 // numero de posicoes da pilha
14     .entrypoint
15
16
17     ldc.i4 0 // coloca 0 na pilha
18     stloc i // grava na variavel i e tira da pilha
19     ldc.i4 0
20     stloc j // grava em j e tira da pilha
21
22
23     //criando um array
24     ldc.i4 10 //tamanho do array
25     newarr int32 // pega o topo da pilha e cria um
        vetor com este tamanho
26     stloc vet1 // vetor 1
27
28     ldc.i4 10 //tamanho do array
29     newarr int32
30     stloc vet2 // vetor 2
31
32     ldc.i4 20 //tamanho do array
33     newarr int32
34     stloc vet3 // vetor 3
35
36
37
38 WHILE: //INICIO DO WHILE
```



```

39         ldloc i
40         ldc.i4 10
41
42         beq CONTINUE
43 // Atribuicao vet 1 e 3
44 //COLOCANDO NO VETOR scanf(" %d ", vet1[i]);
45 ldloc vet1 // carrega o vetor
46 ldloc i
47 call string [mscorlib]System.Console::ReadLine ()
48 call int32 [mscorlib]System.Int32::Parse(string)
49 stelem.i4
50
51
52 // vet3[j] = vet1[i] ;
53 ldloc vet3 // le o vetor 3 bota na pilha
54 ldloc j // j na pilha
55 //pegando o valor de vet1[i]
56 ldloc vet1 // le o vetor 1 e bota na pilha
57 ldloc i
58 ldelem.i4 // le o vetor na posicao e retorna o
        valor (tira da pilha)
59 stelem.i4
60 //J++
61 ldc.i4 1
62 ldloc j
63 add
64 stloc j
65
66
67
68 //atribuicao vet 2 e 3
69
70 //COLOCANDO NO VETOR scanf(" %d ", vet2[i]);
71 ldloc vet2 // carrega o vetor
72 ldloc i
73 call string [mscorlib]System.Console::ReadLine ()
74 call int32 [mscorlib]System.Int32::Parse(string)
75 stelem.i4
76
77
78 // vet3[j] = vet2[i] ;

```

```

79     ldloc vet3 // le o vetor 3 bota na pilha
80     ldloc j    // j    na pilha
81     //pegando o valor de vet1[i]
82         ldloc vet2 // le o vetor 1 e bota na pilha
83         ldloc i
84         ldelem.i4 // le o vetor na posicao e retorna o
            valor (tira da pilha)
85     stelem.i4
86 //J++
87     ldc.i4 1
88     ldloc j
89     add
90     stloc j
91
92     //i++
93     ldc.i4 1
94     ldloc i
95     add
96     stloc i
97
98     br WHILE // retorna while
99
100
101
102 CONTINUE:
103
104     ldc.i4 0
105     stloc i
106
107 WHILE2:
108     ldloc i
109     ldc.i4 20
110     beq FINAL
111
112
113         ldloc vet3
114         ldloc i
115         ldelem.i4
116 call void [mscorlib]System.Console::Write (int32)
117
118 //imprime valor da posicao i do vetor 3

```

```

119
120
121     //i++
122     ldc.i4 1
123     ldloc i
124     add
125     stloc i
126
127     br WHILE2
128
129 FINAL:
130
131     ret
132 }

```

### 3.1.12 Ex12.tsc

#### Mini C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int calculo(int sal);
4 int main()
5 {
6     int sal;
7     int aumento;
8     int novoSal;
9
10    scanf("%d",&sal);
11    aumento = calculo(sal);
12    novoSal = sal + aumento;
13    printf("Novo salário é %d", novoSal);
14 }
15
16 int calculo (int sal)
17 {
18     int perc;
19     int valor;
20
21     scanf("%d",&perc);

```

```

22 | valor = sal*perc/100;
23 | return valor;
24 |
25 | }

```

## MSLI

```

1 | .assembly extern mscorlib {}
2 |
3 | .assembly Salario
4 | {
5 |     .ver 1:0:1:0
6 | }
7 | .module Salario.exe
8 |
9 | .method static void main() cil managed
10 | {
11 |     .locals init (int32 sal,int32 aumento,int32 novoSal
12 |                  )
13 |     .maxstack 10 // numero de posicoes da pilha
14 |     .entrypoint
15 |
16 |     call string [mscorlib]System.Console::ReadLine ()
17 |     call int32 [mscorlib]System.Int32::Parse(string)
18 |
19 |     stloc sal //leitura do salario
20 |
21 |     ldloc sal //poe o salario na pilha para leitura
22 |     na funcao
23 |     call int32 calculo(int32)
24 |     stloc aumento // guardando resultado da funcao
25 |     no aumento
26 |     ldloc aumento // coloca aumento na pilha
27 |     ldloc sal // coloca salario na pilha
28 |     add // aumento
29 |     stloc novoSal //grava a soma em novoSal
30 |
31 |
32 |     ldstr "Novo salario e "
33 |     call void [mscorlib]System.Console::Write (string)

```

```

31
32         ldloc novoSal
33         call void [mscorlib]System.Console::Write(int32
34             )
35     ret
36 }
37
38 .method public int32 calculo(int32) cil managed
39 {
40     .locals init (int32 perc,int32 valor)
41
42
43     call string [mscorlib]System.Console::ReadLine ()
44     call int32 [mscorlib]System.Int32::Parse(string)
45     stloc perc
46
47     ldarg.0 // 1 argumento da chamada da funcao
48     ldloc perc
49     mul.ovf
50
51     ldc.i4 100
52     div
53     stloc valor
54     ldloc valor
55     ret
56
57 }

```

### 3.1.13 Ex13.tsc

#### Mini C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /*
5     Programa que calcula o novo salário
6     /* comentário aninhado 1
7         /* comentário aninhado 2 */

```

```

8          */
9      */
10     int calculo(int sal);
11     int main()
12     {
13         int sal;
14         int aumento;
15         int novoSal; //salario já com o aumento
16
17         scanf(" %d",&sal);
18         aumento = calculo(sal);
19         novoSal = sal + aumento;
20         printf("Novo salário é %d",novoSal);
21     }
22
23     int calculo (int sal)
24     {
25         int perc; //percentual é um número entre 0 e 100
26         int valor;
27
28         scanf("%d",&perc);
29         valor = sal*perc/100;
30         return valor;
31     }
32 }

```

## MSLI

```

1  .assembly extern mscorlib {}
2
3  .assembly Salarior
4  {
5      .ver 1:0:1:0
6  }
7  .module Salarior.exe
8
9  .method static void main() cil managed
10 {
11

```

```

12     .locals init (int32 sal,int32 aumento,int32 novoSal
13         )
14     .maxstack 10 // numero de posicoes da pilha
15     .entrypoint
16     call string [mscorlib]System.Console::ReadLine ()
17     call int32 [mscorlib]System.Int32::Parse(string)
18
19     stloc sal //leitura do salario
20
21     ldloc sal //poe o salario na pilha para leitura
22         na funcao
23     call int32 calculo(int32)
24     stloc aumento // guardando resultado da funcao
25         no aumento
26     ldloc aumento // coloca aumento na pilha
27     ldloc sal // coloca salario na pilha
28     add // aumento
29     stloc novoSal //grava a soma em novoSal
30
31     ldstr "O novo \n"salario \n" e "
32     call void [mscorlib]System.Console::Write (string)
33
34     ldloc novoSal
35     call void [mscorlib]System.Console::Write(int32
36         )
37
38     ret
39 }
40
41 .method public int32 calculo(int32) cil managed
42 {
43     .locals init (int32 perc,int32 valor)
44
45     call string [mscorlib]System.Console::ReadLine ()
46     call int32 [mscorlib]System.Int32::Parse(string)
47     stloc perc
48
49     ldarg.0 // 1 argumento da chamada da funcao

```

```

49         ldloc perc
50         mul.ovf
51
52         ldc.i4 100
53         div
54         stloc valor
55         ldloc valor
56         ret
57     }
58 }

```

### 3.1.14 Ex14.tsc

#### Mini C

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /*
5      Programa que calcula o novo salário
6      /* comentário aninhado 1
7          /* comentário aninhado 2 */
8      */
9  */
10 int calculo(int sal);
11 int soma(int x, int y);
12 int main()
13 {
14     int sal;
15     int aumento;
16     int novoSal; //salario já com o aumento
17
18     scanf("%d",&sal);
19     aumento = calculo(sal);
20     novoSal = soma(sal,aumento)
21     printf("Novo salário é %d",novoSal);
22 }
23
24 int calculo (int sal)
25 {

```



```

26  int perc; //percentual é um número entre 0 e 100
27  int valor;
28
29  scanf("\%d",&perc);
30  valor = sal*perc/100;
31  return valor;
32  }
33  int soma(int x,int y)
34  {
35      return x +y;
36  }

```

## MSLI

```

1
2  .assembly extern mscorlib {}
3
4  .assembly Salarior
5  {
6      .ver 1:0:1:0
7  }
8  .module Salarior.exe
9
10 .method static void main() cil managed
11 {
12
13     .locals init (int32 sal,int32 aumento,int32 novoSal
14                  )
15     .maxstack 10 // numero de posicoes da pilha
16     .entrypoint
17
18     call string [mscorlib]System.Console::ReadLine ()
19     call int32 [mscorlib]System.Int32::Parse(string)
20
21     stloc sal //leitura do salario
22
23     ldloc sal //poe o salario na pilha para leitura
24     na funcao
25     call int32 calculo(int32)

```

```

24         stloc aumento // guardando resultado da funcao
           no aumento
25         ldloc aumento // coloca aumento na pilha
26         ldloc sal // coloca salario na pilha
27         call int32 soma(int32,int32)
28         stloc novoSal //grava a soma em novoSal
29
30
31         ldstr "O novo "salario " e "
32         call void [mscorlib]System.Console::Write (string)
33
34         ldloc novoSal
35         call void [mscorlib]System.Console::Write(int32
           )
36
37         ret
38     }
39
40     .method public int32 calculo(int32) cil managed
41     {
42         .locals init (int32 perc,int32 valor)
43
44
45         call string [mscorlib]System.Console::ReadLine ()
46         call int32 [mscorlib]System.Int32::Parse(string)
47         stloc perc
48
49         ldarg.0 // 1 argumento da chamada da funcao
50         ldloc perc
51         mul.ovf
52
53         ldc.i4 100
54         div
55         stloc valor
56         ldloc valor
57         ret
58
59     }
60     .method public int32 soma(int32 , int32 ) cil managed
61     {
62         ldarg.0

```

```
63     ldarg.1
64     add
65     ret
66
67 }
```

Segunda parte Trabalho de Compiladores  
Relatório

# Capítulo 4

## Analizador Lexical

### 4.1 fslex

O processo de criação do analisador léxico usando a linguagem F# utiliza a ferramenta fslex (o construtor de analisador léxico através de uma especificação em um arquivo .fsl, ou seja, F sharp lexical), uma ferramenta baseada no ocamllex que é presente no pacote **FSharpPowerPack**, o qual pode ser baixado [aqui](#).

Após feita a instalação do pacote podemos encontrar no mesmo duas ferramentas: o fslex.exe, o gerador de um *scanner*, e o fsyacc.exe (o qual utilizaremos na construção de um parser).

Para utiliza-lo basta adicionar o fslex.exe ao path do Windows<sup>1</sup> e chamar a função fslex no Prompt de Comando -CMD, do Windows passando como parâmetro o arquivo contendo as expressões regulares que representam a linguagem.

---

<sup>1</sup>Dessa maneira não é necessário entrar no diretório do fslex.exe toda vez em que precisar utiliza-lo para gerar o analisador léxico.

**Exemplo:** `fslex nome-do-arquivo.fsl -unicode -o nome-do-arquivo-de-saída.fs`

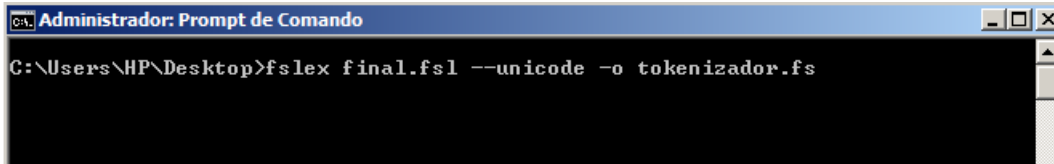


Figura 4.1: Chamada no CMD

## 4.2 Especificação do Analisador Léxico

O processo de geração do analisador léxico, utilizando a linguagem F#, recebe um arquivo com a extensão **.fsl**. O arquivo é organizado em:

- **Início:** qualquer código de usuário necessário para o lexer, assim como abrir módulos, bibliotecas, etc.

{ [Código] }

- **Definições:** padrões identificadores, os quais você pode utilizar nas regras ou outras definições.

let [Ident1] = [Padrão]  
let ...

- **Regras:** ao casar-se com determinado padrão deverá executar tal ação.

rule [Regra1] [arg1... argn] = parse  
— [Padrão 1] [Ação]  
...  
— [Padrão 2] [Ação]  
and [Regra2] [arg1... argn] = parse  
...  
rule [Regra3]

- **Epílogo:** código o qual pode chamar as regras definidas acima.

{ [Código] }

## 4.3 Código para criação do analisador

Para a criação do nosso Analisador Léxico foi utilizado o código abaixo no formato .fls.

As regras `comment` e `string` são as responsáveis pelo tratamento de comentários aninhados e de strings com escape para o caractere `'''`, respectivamente.

```
1 {
2 {
3 (* File MicroC/CLex.lex
4   Lexer specification for micro-C, a small imperative
5     language
6   *)
7 module Lexico
8 open System.Text
9 open Microsoft.FSharp.Text.Lexing
10 open Parser;
11 open System
12
13 let lexeme lexbuf =
14     LexBuffer<char>.LexemeString lexbuf
15
16
17
18 let palavra s =
19     match s with
20     | "char"      -> CHAR
21     | "else"     -> ELSE
22     | "false"    -> CBOOL 0
23     | "if"       -> IF
24     | "int"      -> INT
25     | "float"    -> FLOAT
26     | "bool"     -> BOOL
```

```

27 | "null"    -> NULL
28 | "print"   -> PRINT
29 | "println" -> PRINTLN
30 | "return"  -> RETURN
31 | "true"    -> CBOOL 1
32 | "include" -> INCLUDE
33 | "void"    -> VOID
34 | "while"   -> WHILE
35 | "printint" -> PRINTINT
36 | "printfloat" -> PRINTFLOAT
37 | "printchar" -> PRINTCHAR
38 | "printstr" -> PRINTSTR
39 | "out"     -> OUT
40 | "static"  -> STATIC
41 | -         -> ID s
42
43 let cEscape s =
44   match s with
45   | "\\\\" -> '\\'
46   | "\\\"" -> '\"'
47   | "\\a" -> '\007'
48   | "\\b" -> '\008'
49   | "\\t" -> '\t'
50   | "\\n" -> '\n'
51   | "\\v" -> '\011'
52   | "\\f" -> '\012'
53   | "\\r" -> '\r'
54   | -     -> failwith "Caractere de escape nao
                    reconhecido em C"
55 }
56
57 let biblio = "stdio.h" | "stdlib.h"
58 let num = ['0'-'9']+
59 let intNum = '-'? num
60 let char = '\\'[ 'a'-'z' 'A'-'Z' ] '\\ '
61 let floatNum = '-'? ( num ( '.' num ) | '.' num | num ( 'E' | 'e' ) '-'? num )
62 let ident = '-'? ( [ 'a'-'z' 'A'-'Z' ] | '-' ) ( [ 'a'-'z' 'A'-'Z' ] | '-' | num ) *
63 let whitespace = ' ' | '\\t'
64 let newline = '\\n' | '\\r' '\\n'
65 let string = '"' [ 'a'-'z' 'A'-'Z' ] '"'
66

```

```

67 rule Token = parse
68 | [ ' ' '\t' '\r' ] { Token lexbuf }
69 | '\n' { lexbuf.EndPos <- lexbuf.EndPos.
    NextLine; Token lexbuf }
70 | num { CINT (System.Int32.Parse (lexeme lexbuf)) }
71 | floatNum { CFLOAT (float(lexeme lexbuf)) }
72 | ident { palavra (lexeme lexbuf) }
73 | biblio { BIBLIO (lexeme lexbuf) }
74 | '+' { PLUS }
75 | '-' { MINUS }
76 | '*' { TIMES }
77 | '/' { DIV }
78 | '\%' { MOD }
79 | '=' { ASSIGN }
80 | "==" { COMPARE }
81 | "!=" { DIFFER }
82 | "#" { SHARP }
83 | '>' { MAIORQ }
84 | '<' { MENORQ }
85 | ">=" { MAIOREQ }
86 | "<=" { MENOREQ }
87 | "||" { OR }
88 | "&&" { AND }
89 | "&" { AMP }
90 | "!" { NOT }
91 | '(' { LPAR }
92 | ')' { RPAR }
93 | '{' { LCHAV }
94 | '}' { RCHAV }
95 | '[' { LCOL }
96 | ']' { RCOL }
97 | ';' { FINALI }
98 | ',' { VIRG }
99 | "//" { ComentarioDeLinha lexbuf; Token
    lexbuf }
100 | "/*" { Comentario lexbuf; Token lexbuf }
101 | "\"" { CSTRING (String [] lexbuf) }
102 | eof { EOF }
103 | - { failwith "Simbolo ilegal rapaz" }
104
105 and Comentario = parse
106 | "/*" { Comentario lexbuf; Comentario lexbuf
    }

```



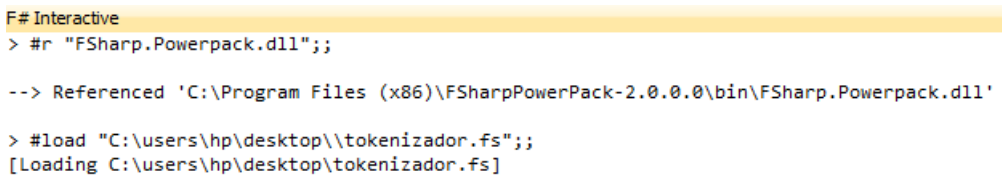
```

107 | "*/"          { () }
108 | '\n'          { lexbuf.EndPos <- lexbuf.EndPos.
      NextLine; Comentario lexbuf }
109 | (eof | '\026') { failwith "Comentario nao finalizado"
      }
110 | -            { Comentario lexbuf }
111
112 and ComentarioDeLinha = parse
113 | '\n'          { lexbuf.EndPos <- lexbuf.EndPos.
      NextLine }
114 | (eof | '\026') { () }
115 | -            { ComentarioDeLinha lexbuf }
116
117 and String chars = parse
118 | ""
119   { Microsoft.FSharp.Core.String.concat "" (List.map
      string (List.rev chars)) }
120 | '\\', ['\\', '"', 'a', 'b', 't', 'n', 'v', 'f', 'r']
121   { String (cEscape (lexeme lexbuf) :: chars) lexbuf }
122 | ""
123   { String ('\\' :: chars) lexbuf }
124 | '\\',
125   { failwith "Erro Lexico: caracter inesperado no
      comentário" }
126 | (eof | '\026')
127   { failwith "Erro Lexico: string nao finalizada" }
128 | ['\n', '\r']
129   { failwith "Lexer error: newline in string" }
130 | ['\000' - '\031', '\127', '\255']
131   { failwith "Erro Lexico: caracter invalido na string"
      }
132 | -
133   { String (char (lexbuf.LexemeChar 0) :: chars) lexbuf
      }

```

Após gerada o analisador léxico através do arquivo usado no fslex podemos executar uma chamada da funcao existente dentro do arquivo que ao receber um buffer como entrada consegue transformar o conteúdo da arquivo em tokens especificos da linguagem MINIC, isso pode ser feito através do F# interactive que pode ser invocado usando o atalho Ctrl+Alt+F.

Com o F# Interactive aberto siga os passos:



```

F# Interactive
> #r "FSharp.Powerpack.dll";;

--> Referenced 'C:\Program Files (x86)\FSharpPowerPack-2.0.0.0\bin\FSharp.Powerpack.dll'

> #load "C:\users\hp\desktop\tokenizador.fs";;
[Loading C:\users\hp\desktop\tokenizador.fs]

```

parte.png

Figura 4.2: Chamada do Power Pack

1. Faz a chamada do pacote Power Pack o qual contem as bibliotecas do lex (primeira linha da Figura 1.2).
2. Lê o arquivo gerado pela saída do fslex(linha correspondente ao load na Figura 1.2).
3. `let lexbuf = Lexing.LexBuffer<_>.FromString "3.4 x 34 xyx";;`  
Cria um buffer da String "3.4 x 34 xyx".
4. `Lexico.token lexbuf;;`

Chamada da função token, sendo sua saída:

1	0 0
2	
3	Lexico.token = FLOAT 3.4

Essa função deve ser chamada até que se encontre o fim do arquivo.

# Capítulo 5

## Teste do Analisador

### 5.1 Relatório de saídas

Para os três primeiros exercícios do trabalho anterior foi aplicado o analisador lexico gerado a partir da especificação do capítulo 1.

Além disso, os mesmos exercícios foram alterados para gerarem erros, tais como: caracter inválido, string não fechada e comentário não fechado.

Segue os exercícios e suas respectivas saídas:

#### Exercício 1

Código em C:

```
1      int main()  
2      {  
3  
4      }
```

Saída do analisador léxico:

```
1      0 0 INT  
2      0 4 MAIN  
3      0 8 LPAR  
4      0 9 RPAR  
5      0 10 LCHAV  
6      0 11 RCHAV
```

## Exercício 2

Código em C:

```
1      int main()  
2      {  
3          int x;  
4          x = 4;  
5      }
```

Saída do analisador léxico:

```
1      0 0 INT  
2      0 3 MAIN  
3      0 8 LPAR  
4      0 9 RPAR  
5      1 0 LCHAV  
6      2 0 INT  
7      2 3 ID "X"  
8      2 4 FINALI  
9      3 0 ID "X"  
10     3 3 ATRIB  
11     3 5 INTEGER 4  
12     3 6 FINALI  
13     4 0 RCHAV
```

## Exercício 3

Código em C:

```
1      int main()  
2      {  
3          int x,y,z;  
4  
5          x = 2;  
6          y = 5;  
7          z = x + y;  
8      }
```

Saída do analisador léxico:

1	0 0 INT
2	0 3 MAIN
3	0 8 LPAR
4	0 9 RPAR
5	1 0 LCHAV
6	2 0 INT
7	2 4 ID "X"
8	2 6 ID "Y"
9	2 7 ID "Z"
10	2 8 FINALI
11	4 0 ID "X"
12	4 2 ATRIB
13	4 4 INTEGER 2
14	4 5 FINALI
15	5 0 ID "Y"
16	5 2 ATRIB
17	5 4 INTEGER 5
18	5 5 FINALI
19	6 0 ID "Z"
20	6 2 ATRIB
21	6 4 ID "X"
22	6 6 PLUS
23	6 8 ID "Y"
24	6 9 FINALI
25	7 0 RCHAV

## 5.2 Erros lexicais

### 5.2.1 Caracter ilegal

Código em C:

```

1      int main()
2      {
3          @
4      }
```

Saída do analisador léxico: **unrecognized input: '@'**

```

1      System.Exception: unrecognized input: '@'
2      at FSI_0002.Tokenizador._fslex_token@88-72.Invoke(
      String message) in C:\Users\HP\AppData\Local\Temp
      \final.fsl:line 88
3      at Microsoft.FSharp.Core.PrintfImpl.go@512-3[b,c,d](
      String fmt, Int32 len, FSharpFunc'2 outputChar,
      FSharpFunc'2 outa, b os, FSharpFunc'2 finalize,
      FSharpList'1 args, Int32 i)
4      at Microsoft.FSharp.Core.PrintfImpl.run@510[b,c,d](
      FSharpFunc'2 initialize, String fmt, Int32 len,
      FSharpList'1 args)
5      at Microsoft.FSharp.Core.PrintfImpl.capture@529[b,c,
      d](FSharpFunc'2 initialize, String fmt, Int32 len
      , FSharpList'1 args, Type ty, Int32 i)
6      at <StartupCode$FSharp-Core>.$Reflect.Invoke@617-4.
      Invoke(T1 inp)
7      at <StartupCode$FSI_0004>.$FSI_0004.main@()
8      Stopped due to error

```

### 5.2.2 Comentário nao fechado

Código em C:

```

1      int main()
2      {
3          /*
4          int x;
5          x = 4;
6      }

```

Saída do analisador léxico: **Unterminated comment**

```

1      System.Exception: Unterminated comment
2      at FSI_0002.Tokenizador._fslex_comment
3      Stopped due to error

```

### 5.2.3 String inacabada

Código em C:

```
1      int main()  
2      {  
3          "asdasd  
4          int x,y,z;  
5          x = 2;  
6          y = 5;  
7          z = x + y;  
8      }
```

Saída do analisador léxico: **end of file in string started at  
or near {pos\_fname = "; ...}**

```
1 2 0  
2 System.Exception: end of file in string started at or  
   near {pos_fname = "; ...}
```

Terceira parte Trabalho de Compiladores  
Relatório

# Capítulo 6

## Análise Sintática

### 6.1 Introdução

A análise sintática é a fase responsável por pegar os tokens gerados pelo léxico e verificar se esta cadeia pode ser gerada por uma gramática livre de contexto determinada. Para realização desta fase é necessário criar uma ASA (Árvore Sintática Abstrata)/"Ast.fs" seu objetivo é transformar o código original em uma representação conveniente para continuação do processo na fase de análise semântica.

Para esta fase temos a construção do Parser e para no ajudar nesta tarefa assim como na fase anterior o pacote PowerPack nos oferece uma ferramenta para gerar o código de um parser. No arquivo utilizado para gerar o Parser temos a descrição de terminais e não-terminais, os operadores associados as suas definições de precedência, além das definições das regras da gramática.

A ferramenta fsyacc.exe funciona com arquivos de texto com extensão .fsy. Estes arquivos possuem três partes distintas. Primeiramente o cabeçalho, uma seção com F puro cercada por símbolos de percentagem e chaves % { para abertura e % } para fechar). Esta seção tipicamente é utilizada para iniciar o módulo da AST (Abstract Syntax Tree) e definir os terminais da linguagem. Um terminal (Token) é algo concreto na sua gramática algo como um identificador ou um símbolo. Tipicamente estes são encontrados pelo lexer.

A ferramenta denominada fsyacc é chamada a partir do pacote PowerPack através do Cmd.exe com o seguinte comando:

```
fsyacc -module "Nome ModuloNome do arquivo"
```



Declarações	Descrição
%token	Declara um símbolo dado como um token na linguagem
%token tipo	Declara símbolos como token, com argumentos com tipo dado
%start	Esta declara a regra na qual o parser deve iniciar
%type type	Declara o tipo de uma regra em particular
%left	Declara um token com associação a esquerda
%nonassoc	Declara um token sem associação

Tabela 6.1: Tabela 11-2 retirada livro Apress Foundations of F Sharp -2007  
.

## 6.2 Gramática

Abaixo está representada a descrição das regras usadas para gerar a gramática do MiniC, as definições de tokens e todas as associações de precedências que os operadores necessitam (Em sua grande maioria se não declaradas geram conflitos shift/reduce )

### 6.2.1 Parser.fsy

**Parser**

```

1
2
3 %{
4
5
6 open System.Collections.Generic
7
8 let compose1 f (g, s) = ((fun x -> g(f(x))), s)
9 open Ast
10 %}
11
12 %token <int> CINT CBOOL
13 %token <string> CSTRING ID
14 %token <string> BIBLIO
15 %token VOID
16 %token <float> CFLOAT
17

```

```

18 %token CHAR ELSE IF INT FLOAT NULL BOOL MAIN PRINT
    PRINTLN RETURN WHILE SHARP INCLUDE PRINTINT
    PRINTFLOAT PRINTCHAR PRINTSTR OUT STATIC
19 %token PLUS MINUS TIMES DIV MOD
20 %token COMPARE DIFFER MAIORQ MENORQ MAIOREQ MENOREQ
21 %token NOT OR AND
22 %token LPAR RPAR LCHAV RCHAV LCOL RCOL FINALI VIRG
    ASSIGN AMP
23 %token EOF
24
25 %right ASSIGN /* Menor precedencia */
26 %nonassoc PRINT PRINTINT PRINTFLOAT PRINTCHAR PRINTSTR
27 %left OR
28 %left AND
29 %left COMPARE DIFFER
30 %left MAIORQ MENORQ MAIOREQ MENOREQ
31 %left PLUS MINUS
32 %left TIMES DIV MOD
33 %nonassoc IFX
34 %nonassoc ELSE
35 %nonassoc NOT AMP
36 %nonassoc LCOL /* Maior precedencia */
37
38 %start Programa
39 %type < Ast.programa > Programa /*Tipo de retorno da
    funcao Programa (obrigatorio apenas para a primeira
    funcao)*/
40
41 %%
42
43 /*Clausula inicial: retorna Programa( [Bibliotecas], [
    Funcoes] )*/
44
45 Programa:
46     Prologo DeclFuncs EOF {
47         Prog ( Biblio ( $1 ) , $2 ) }
48 ;
49 /*Retorna uma lista com as bibliotecas inclusas*/
50
51 Prologo:

```

```

52 | /*vazio*/ { [] }
53 | SHARP INCLUDE MENORQ BIBLIO MAIORQ Prologo { $4::
    $6 }
54 ;
55
56 Tipo:
57     INT { TipoI }
58 | CHAR { TipoC }
59 | FLOAT { TipoF }
60 | BOOL { TipoB }
61 ;
62
63 /*Array de declaracoes de funcoes*/
64
65 DeclFuncs:
66     /*vazio */ { [] }
67 | DeclFunc DeclFuncs { $1::$2 }
68 ;
69
70 /*
71 Retorna as funcoes. No caso, a funcao principal sera do
    tipo DeclFunc(int, main, [], [Variaveis], Comandos)
72 as demais:
73 DeclFunc(Tipo, Nome, [argumentos], [Variaveis], [
    Comandos])
74 */
75
76 DeclFunc:
77     VOID ID LPAR ListaDeArgs RPAR Bloco { DeclFunc(
    TipoVoid, $2, $4, $6) }
78 | Tipo ID LPAR ListaDeArgs RPAR Bloco { DeclFunc($1,
    $2, $4, $6) }
79 | error { failwith("Erro na declaração de função") }
80 ;
81
82 DeclVars:
83     /*vazio*/ { [] }
84 | DeclVar FINALI DeclVars{ Vardec(fst $1, snd $1):: $3
    }
85 ;
86

```

```

87
88 DeclVar :
89     Tipo DescVar { ((fst $2) $1, snd $2 ) } /*Vardec(
          tipo(tipo), ID) ou Vardec(TipoV (tamano) ID)*/
90 ;
91
92 DescVar :
93     ID { ((fun t -> t), $1) }
94     | DescVar LCOL CINT RCOL { compose1 (fun t -> TipoV(t
          , $3)) $1 }
95     | error { failwith("Erro_na_descrição_de_variáveis")
          }
96 ;
97
98 ListaDeArgs :
99     /* vazio */ { [] }
100    | ListaDeArgs1 { $1 }
101 ;
102
103 ListaDeArgs1 :
104     DeclVar { [$1] }
105    | DeclVar VIRG ListaDeArgs1 { $1::$3 }
106 ;
107
108
109 Bloco :
110    | LCHAV ComandoOuDecl RCHAV { Bloco $2 }
111 ;
112
113
114 ComandoOuDecl :
115 /*vazio*/ { [] }
116    | Comando ComandoOuDecl { Comando $1::$2 }
117    | DeclVar FINALI ComandoOuDecl { Dec(fst $1, snd $1)
          ::$3 }
118 ;
119
120
121 Comando :
122     Expressao FINALI { Expr($1) }
123    | RETURN FINALI { Retorno None }

```

```

124 | RETURN Expressao FINALI {Retorno (Some ($2)) }
125 | Bloco {$1}
126 | IF LPAR Expressao RPAR Comando %prec IFX { If($3,$5
    ,Bloco[]) }
127 | IF LPAR Expressao RPAR Comando ELSE Comando { If($3
    ,$5,$7) }
128 | WHILE LPAR Expressao RPAR Comando { While($3,$5) }
129 ;
130
131 /*
132     GAmbarra MASTER!
133 %prec é uma definição de precedência de operadores; %
    nonassoc indica que um operador não é associativo.
    Estas directivas permitem ao parser tomar decisões
    relativamente a situações em que a gramática seria
    ambígua. Quando a gramática não é ambígua, as
    directivas são simplesmente ignoradas. IFX é um
    token utilizado pelo parser para resolução de uma
    potencial ambiguidade na gramática.
134
135 Fontes :http://stackoverflow.com/questions/1737460/how-
    to-find-shift-reduce-conflict-in-this-yacc-file
136 https://fenix.ist.utl.pt/disciplinas/com56
    /2011-2012/2-semester/faq/compact
137 */
138
139
140 Elses:
141 /*vazio muchacho*/ { Bloco [] }
142 | ELSE Comando { $2 }
143 ;
144
145
146 Expressao:
147     Variavel { Acesso $1 }
148 | Expr { $1 }
149 | error { failwith("Erro na expressão") }
150 ;
151
152 Expr:

```

153	CINT	{
	ConstInt \$1 }	
154	CBOOL	{
	ConstBool \$1 }	
155	CFLOAT	{
	ConstFloat \$1 }	
156	/*  CSTRING	{ \$1 }
	*/	
157	MINUS CINT	{
	ConstInt -\$2 }	
158	NULL	{
	ConstNull -1 }	
159	ID LPAR ListaExpr RPAR	{ Call
	(\$1,\$3) } /*————— Duvidas	
	—————*/	
160	NOT Expressao	{ Op("
	!", \$2) }	
161	Variavel ASSIGN Expressao	{
	Atrib(\$1,\$3) }	
162	Expressao PLUS Expressao	{
	Binop("+", \$1,\$3) }	
163	Expressao MINUS Expressao	{
	Binop("-", \$1,\$3) }	
164	Expressao TIMES Expressao	{
	Binop("*", \$1,\$3) }	
165	Expressao DIV Expressao	{
	Binop("/", \$1,\$3) }	
166	Expressao MOD Expressao	{
	Binop("%", \$1,\$3) }	
167	Expressao COMPARE Expressao	{
	Binop("=", \$1,\$3) }	
168	Expressao DIFFER Expressao	{
	Binop("!=", \$1,\$3) }	
169	Expressao MAIORQ Expressao	{
	Binop(">", \$1,\$3) }	
170	Expressao MENORQ Expressao	{
	Binop("<", \$1,\$3) }	
171	Expressao MAIOREQ Expressao	{
	Binop(">=", \$1,\$3) }	
172	Expressao MENOREQ Expressao	{
	Binop("<=", \$1,\$3) }	

```

173 | Expressao AND Expressao { And
    ($1,$3) }
174 | Expressao OR Expressao { Or(
    $1,$3) }
175 | PRINTINT LPAR Expressao RPAR { Op(
    "printint",$3) }
176 | PRINTFLOAT LPAR Expressao RPAR { Op(
    "printfloat",$3) }
177 | PRINTCHAR LPAR Expressao RPAR { Op(
    "printchar",$3) }
178 | PRINTSTR LPAR Expressao RPAR { Op(
    "printstr",$3) }
179 ;
180
181 Variavel:
182     ID { AccVar $1 }
183 | ID LCOL Expressao RCOL { AccIndex($1, $3)
    }
184 ;
185
186 ListaExpr:
187     /*Vazio*/ { [] }
188 | Expressoes { $1 }
189 ;
190
191 Expressoes:
192     Expressao { [$1] }
193 | Expressao VIRG Expressoes { $1::$3 }
194 ;

```

Como saída das regras da gramática temos a geração dos nós da Ast que casam neste exemplo uma operação que verifica a diferença entre duas expressões:

Expressao DIFFER Expressao Binop("!=",1,3)

Gerando o nó da árvore Binop("!=",Expressao1,Expressao2)

## 6.3 Erros Sintáticos

Produções que não consigam casar com a gramática são consideradas como erro e retornam o tipo de erro a linha e a coluna em que ele pode ser encon-

trado .

Exemplo: `int x ;;`

(Sendo apenas um ponto e vírgula esta produção seria aceita, como existem duas o parser avisa o erro)

## 6.4 Árvore - ASA - Ast

A árvore começa a ser gerada enquanto o parser casa os padrões da gramática. Note que a primeira regra Programa é do tipo árvore AST. Os padrões do tipo option podem receber parâmetros `Some("conteúdo")` ou `None`. Padrões do tipo list recebem uma lista como parâmetro.

### 6.4.1 Ast.fs

Ast

```
1 module Ast
2
3
4 type tipo =
5   | TipoI  (* tipo inteiro *)
6   | TipoC  (* tipo character *)
7   | TipoV  of tipo * int (* tipo vetor *)
8   | TipoF  (* tipo float *)
9   | TipoVoid of void (* tipo void *)
10
11 and expressao =
12   | Acesso of variavel (* constante *)
13   | Atrib  of variavel * expressao (* atribuicao variavel *)
14   | ConstExp of int (* constante *)
15   | Op  of string * expressao (* operacao unaria *)
16   | Binop of string * expressao * expressao (* operacao dois operadores *)
17   | And  of expressao * expressao (* e logico *)
```



```

18 | Or of expressao * expressao (*
    |   ou logico *)
19 | Call of string * expressao list
    |   (* chamada de funcao *)
20
21 and variavel =
22 |   AccVar of string
23 |   AccIndex of string * expressao
24
25
26 and comando =
27 |   If of expressao * comando * comando (* if -
    |   ultimo é option pois pode ser balanceado ou nao*)
28 |   While of expressao * comando (*
    |   while *)
29 |   Retorno of expressao option (*
    |   retorno da expresao*)
30 |   Expr of expressao
31 |   Bloco of comandoOuDecl list (*
    |   Bloco de comandos *)
32 (* | Comando of comando (* Bloco
    |   de comandos *) *)
33
34 and comandoOuDecl =
35 |   Dec of tipo * string (*
    |   Declaracao de variaveis locais *)
36 |   Comando of comando (* Bloco de
    |   comandos *)
37
38 and declaraVar =
39 |   Vardec of tipo * string
40
41 and declaraFunc =
42 |   DeclFunc of tipo * string * (tipo * string) list *
    |   comando (* Declaracao de funcao*)
43
44 and bibliotecas =
45 |   Biblio of string list (* Bibliotecas*)
46
47 and programa =

```

48	Prog of bibliotecas * declaraFunc list (* Programa completo*)
----	--

## 6.5 Execução

Para executar o código do Léxico e Parser criamos um script que faz as devidas chamadas de funções, onde passamos apenas um arquivo, a função lê o arquivo cria um buffer e aplica a função Token do Léxico neste buffer em seguida a função Programa do Parser é chamada a fim de criar a Árvore.

```

1 #r "FSharp.PowerPack.dll"
2 open System
3 open System.IO
4 open Microsoft.FSharp.Text.Lexing
5 #light
6
7
8 #load "Ast.fs"
9 #load "Parser.fs"
10 #load "Lexico.fs"
11
12 let fromFile (filename : string) =
13     use reader = new StreamReader(filename)
14     let lexbuf = Lexing.LexBuffer<char>.FromTextReader
15         reader
16     in try
17         Parser.Programa Lexico.Token lexbuf
18     with
19         | exn -> let pos = lexbuf.EndPos
20                 in failwithf "%s No programa %s na %s %d, %d\n"
21                     (exn.Message) filename (pos.Line
22                     +1) pos.Column
23
24 let buf = fromFile("caminho\ex1.c")
25 let _ = System.Console.WriteLine(buf)

```

# Capítulo 7

## Análise Semântica

### 7.1 Introdução

A etapa de análise semântica, terceira fase do Front-End de um compilador, pode ser dividida em três fases principais, sendo elas a criação da tabela de símbolos, a análise de escopos e tipos e a validação das expressões e comandos. Para estas etapas usamos a tabela de símbolos para nos auxiliar no processo.

A tabela de símbolos é comumente implementada como uma hash, devido à quantidade de acesso e buscas pelo valor de uma certa variável e seu tipo, ou até a funções, buscando parâmetros, tipo de retorno, entre outras especificações pertencente às regras, gramática, da linguagem analisada.

Utilizamos uma tabela hash para representar o escopo global do programa, todas as suas funções, tendo esta tabela como chave(index) o nome da função (já que em MiniC não pode haver duas funções com o mesmo nome). Cada linha da tabela de símbolos global representa uma função, onde cada função terá sua própria tabela de símbolos (hash) a validação interna de suas variáveis.

### 7.2 Implementação

Na implementação do código utilizamos para representar a tabela de símbolos um Mapa, uma estrutura pertencente a uma das bibliotecas nativas do F, o que nada mais é do que uma tabela hash do tipo `Map<chave,valor>`. Em nossa implementação temos a tabela de símbolos para as variáveis como sendo uma tabela hash onde a chave é o nome da variável e o valor na tabela é uma tupla contendo o tipo da variável e seu valor (em um primeiro momento não é utilizado, apenas para verificação se aquela variável tem ou não um valor). A tabela de símbolos global é uma hash contendo, como chave, o nome

da função e seu valor: uma tabela de símbolos para as variáveis da função, seu tipo de retorno e o valor de seu retorno. Iniciamos a análise semântica percorrendo a árvore e populando o ambiente global com as funções encontradas e suas respectivas tabelas de símbolos de variáveis com os parâmetros. Tendo em mãos o ambiente global, analisamos, separadamente, o corpo de cada função, a fim de validar os comandos e declarações encontradas. Para isto forma-se uma estrutura de funções, associando a cada elemento, tipo, da nossa árvore sintática abstrata (não terminal) uma função para tratá-lo. Por exemplo, o tipo comando em nossa árvore pode ser um If, While, Retorno, Expr ou Bloco; para validá-los cria-se uma função, validaComando, onde recebe os ambientes populados e uma variável do tipo comando e casa ela com o que receber (If, While,...), tratando cada uma de uma maneira específica. Com a criação de um conjunto de funções recursivas, cada uma associada ao tipo que podemos receber da árvore, fica fácil percorrer a árvore e validar onde cada parte da árvore tem sua implementação porém a idéia geral é:

- Receber os comandos, para eles basta apenas validar as operações, caso seja um condicional, e chamar recursivamente com os comandos dos condicionais.
- Nas expressões retornamos o tipo da expressão (pertencente ao tipo tipo da árvore: TipoI, TipoF, TipoS ...). Por exemplo, caso seja um Binop(+,a,b) validamos a expressão do lado direito (a) e do lado esquerdo (b) e o retorno de cada uma deve ser compatível com a operação em questão (para o + pode ser TipoI + TipoI retornando um TipoI, TipoF + TipoI -, TipoF, e assim por diante). As demais validações seguem o mesmo raciocínio e não entraremos tão afundo, já que o código está disponibilizado.

## 7.3 Principais problemas enfrentados

Um dos principais problemas que enfrentamos foi quanto a tipagem forte do F. Ao criarmos um mapa atribuímos a ele um valor, porém se declararmos o mapa como uma variável normal durante a execução do programa não conseguimos alterar seu valor em diferentes funções impossibilitando a inserção de vários elementos no mapa. Isso acontece devido a função Map.Add() devolver uma nova estrutura, devemos então declarar a variável como mutable e atribuímos o MapAntigo.Add() ao novo Map .

Exemplo: `let mutable novoMapa = MapAntigo.Add()`

# Capítulo 8

## Referências Bibliográficas

- 1 <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf> Acessado em 12 de Março de 2012
- 2 <http://www.ecma-international.org/memento/index.html> Acessado em 12 de Março de 2012
- 3 <http://community.grapecity.co.in/2011/10/net-architecture.html> Acessado em 12 de Março de 2012
- 4 [http://pt.wikipedia.org/wiki/Common\\_Language\\_Infrastructure](http://pt.wikipedia.org/wiki/Common_Language_Infrastructure) Acessado em 12 de Março de 2012
- 5 <http://www.yoda.arachsys.com/csharp/faq/> Acessado em 12 de Março de 2012
- 6 <http://msdn.microsoft.com/pt-br/library/z1zx9t92.aspx> Acessado em 12 de Março de 2012
- 7 [http://msdn.microsoft.com/en-us/library/zcx1eb1e\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/zcx1eb1e(v=vs.71).aspx) Acessado em 12 de Março de 2012
- 8 [http://msdn.microsoft.com/pt-br/library/12a7a7h3\(v=vs.90\).aspx](http://msdn.microsoft.com/pt-br/library/12a7a7h3(v=vs.90).aspx) Acessado em 12 de Março de 2012
- 9 [http://www.baboo.com.br/conteudo/modelos/Arquitetura-NET\\_a11283\\_z0.aspx](http://www.baboo.com.br/conteudo/modelos/Arquitetura-NET_a11283_z0.aspx) Acessado em 12 de Março de 2012
- 10 <http://www.ic.unicamp.br/~rodolfo/Cursos/mo401/2s2005/Trabalho/002092-net.pdf> Acessado em 12 de Março de 2012
- 11 <http://www.dotnetlanguages.net/DNL/Resources.aspx> Acessado em 06 de Novembro de 2012

12 <http://msdn.microsoft.com/pt-br/library/496e4ekx.aspx> Acessado em 08 de Novembro de 2012