

ESQUEMA DE TRADUÇÃO – completo (para implementação do analisador semântico e gerador de código)

```
<programa> ::= #100 fun main "{" <lista_instrucoes> "}" #101 ;

<lista_instrucoes> ::= <instrucao> ";" <lista_instrucoes_> ;
<lista_instrucoes_> ::= ̑ | <lista_instrucoes> ;

<instrucao> ::= <lista_identificadores> <instrucao_> | <entrada> | <saida> | <selecao> | <repeticao> ;
<instrucao_> ::= ":" <valor> #126 | #127 ̑ | "=" <expressao> #128 ;

<valor> ::= constante_int | constante_float | constante_string | true | false ;

<lista_identificadores> ::= identificador #125 <lista_identificadores_> ;
<lista_identificadores_> ::= ̑ | "," <lista_identificadores> ;

<comando> ::= <atribuicao> | <entrada> | <saida> | <selecao> | <repeticao> ;

<atribuicao> ::= <lista_identificadores> "=" <expressao> #128 ;

<entrada> ::= in "(" <lista_entrada> ")" ;
<lista_entrada> ::= <opcional> <lista_identificadores> #129 <lista_entrada_> ;
<lista_entrada_> ::= ̑ | ";" <lista_entrada> ;
<opcional> ::= ̑ | constante_string #130 "," ;

<saida> ::= out "(" <lista_expressoes> ")" ;
<lista_expressoes> ::= <expressao> #102 <lista_expressoes_> ;
<lista_expressoes_> ::= ̑ | "," <lista_expressoes> ;

<selecao> ::= if "(" <expressao> ")" #118 "{" <lista_comandos> "}" <else> #119 ;
<else> ::= ̑ | #120 else "{" <lista_comandos> "}" ;
<lista_comandos> ::= <comando> ";" <lista_comandos_> ;
<lista_comandos_> ::= ̑ | <lista_comandos> ;

<repeticao> ::= #121 while "(" <expressao> ")" #122 do "{" <lista_comandos> "}" #123
| #121 repeat "{" <lista_comandos> "}" while "(" <expressao> ")" #124 ;

<expressao> ::= <elemento> <expressao_> ;
<expressao_> ::= ̑ | "&" <elemento> #103 <expressao_> | "|" <elemento> #104 <expressao_> ;
<elemento> ::= <relacional> | true #105 | false #106 | "!" <elemento> #107 ;

<relacional> ::= <aritmetica> <relacional_> ;
<relacional_> ::= ̑ | <operador_relacional> #108 <aritmetica> #109 ;
<operador_relacional> ::= "==" | "!=" | "<" | ">" ;

<aritmetica> ::= <termo> <aritmetica_> ;
<aritmetica_> ::= ̑ | "+" <termo> #110 <aritmetica_> | "-" <termo> #111 <aritmetica_> ;

<termo> ::= <fator> <termo_> ;
<termo_> ::= ̑ | "*" <fator> #112 <termo_> | "/" <fator> #113 <termo_> ;

<fator> ::= identificador #131 |
constante_int #114 |
constante_float #115 |
constante_string #116 |
 "(" <expressao> ")" |
 "+" <fator> |
 "-" <fator> #117 ;
```

DESCRIÇÃO DOS REGISTROS SEMÂNTICOS: para executar a análise semântica e a geração de código é necessário fazer uso de registros semânticos (outros podem e devem ser definidos, bem como os descritos abaixo podem ser alterados, conforme a implementação das ações semânticas):

- operador_relacional** (inicialmente igual a ""): usado para armazenar o operador relacional reconhecido pela ação **#108**, para uso posterior na ação **#109**
- código_objeto**: usado para armazenar o código objeto gerado
- pilha_tipos** (inicialmente vazia): usada para determinar o tipo de uma expressão durante a compilação do programa
- pilha_rotulos** (inicialmente vazia): usada na análise dos comandos de seleção e de repetição
- lista_id** (inicialmente vazia): usada para armazenar os identificadores reconhecidos pela ação **#125**, para uso posterior nas ações **#126 a #129**
- tabela_simbolos** (inicialmente vazia): usada para armazenar informações sobre os identificadores declarados (constantes e variáveis, ações **#126 e #127**). Cada linha da tabela tem três campos:

identificador	tipo	valor
de variável ou de constante do tipo int	int64	
de variável ou de constante do tipo float	float64	
de variável ou de constante do tipo string	string	
de variável ou de constante do tipo bool	bool	

Para constantes, o campo `valor` deve ser preenchido com o valor da constante conforme declarado no programa. Para variáveis, o campo `valor` deve ficar sem conteúdo.

Assim, por exemplo, para o programa

```
fun main {  
    _fnotamaxima: 10.0;  
    _icontador, _fnota;  
}
```

A `tabela_simbolos` deverá ser preenchida da seguinte forma:

identificador	tipo	valor
<code>_fnotamaxima</code>	<code>float64</code>	<code>10.0</code>
<code>_icontador</code>	<code>int64</code>	
<code>_fnota</code>	<code>float64</code>	

TABELA DE TIPOS: o tipo de uma `<expressão>` deve ser determinado da seguinte forma:

operando ₁	operando ₂	operador	tipo resultante
identificador			conforme <code>tabela_simbolos</code>
constante_int			<code>int64</code>
constante_float			<code>float64</code>
constante_string			<code>string</code>
<code>true</code>			<code>bool</code>
<code>false</code>			<code>bool</code>
<code>int64</code>	<code>int64</code>	operadores binários: <code>+</code> <code>-</code> <code>*</code> <code>/</code>	<code>int64</code>
<code>int64</code> <code>float64</code> <code>float64</code>	<code>float64</code> <code>int64</code> <code>float64</code>	operadores binários: <code>+</code> <code>-</code> <code>*</code> <code>/</code>	<code>float64</code>
<code>int64</code> <code>int64</code> <code>float64</code> <code>float64</code> <code>string</code>	<code>int64</code> <code>float64</code> <code>int64</code> <code>float64</code> <code>string</code>	<code>==</code> <code>!=</code> <code><</code> <code>></code>	<code>bool</code>
<code>bool</code>	<code>bool</code>	operadores binários: <code>&</code> (and) <code> </code> (or)	<code>bool</code>

A verificação da compatibilidade de tipos não será implementada. Mas, é necessário determinar o tipo de uma expressão, conforme indicado na tabela acima.

DESCRIÇÃO DA SEMÂNTICA:

- A ação **#100** deve gerar código com o cabeçalho do programa objeto (verificar no `exemplo.il`).
- A ação **#101** deve gerar código com as instruções para finalizar o programa objeto (verificar no `exemplo.il`).
- A semântica de uma `<expressão>` é a seguinte:
 - para identificador (ação **#131**):
 - verificar se o identificador (`token.getLexeme`) foi declarado, ou seja, se está na `tabela_simbolos`;
 - em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `token.getLexeme` não declarado (por exemplo: `_iarea` não declarado);
 - em caso positivo e é identificador de constante:
 - gerar código objeto para carregar o valor da constante, sendo que o `valor` deve ser recuperado da `tabela_simbolos` e que o código objeto a ser gerado depende do tipo da constante (código pode ser: `ldc.i8, ldc.r8, ldstr, ldc.i4.1, ldc.i4.0`);
 - se a constante for do tipo `int64`, gerar código objeto para converter o valor para `float64` (código: `conv.r8`);
 - empilhar o tipo da constante na `pilha_tipos`, sendo que o tipo deve ser recuperado da `tabela_simbolos`.
 - em caso positivo e é identificador de variável:
 - gerar código objeto para carregar o valor armazenado em identificador (código: `ldloc token.getLexeme`);
 - se a variável for do tipo `int64`, gerar código objeto para converter o valor para `float64` (código: `conv.r8`);
 - empilhar o tipo do identificador na `pilha_tipos`, sendo que o tipo deve ser recuperado da `tabela_simbolos`.

- para `constante_int` (ação #114):
 - (a) empilhar na `pilha_tipos` o tipo correspondente, conforme TABELA DE TIPOS;
 - (b) gerar código objeto para carregar o valor da constante (código: `ldc.i8 token.getLexeme`), observando que a `constante_int` da linguagem fonte deve ser tratada como `float64` em IL, portanto deve ser convertida para `float64` (código: `conv.r8`).
 - para `constante_float` (ação #115):
 - (a) empilhar na `pilha_tipos` o tipo correspondente, conforme TABELA DE TIPOS;
 - (b) gerar código objeto para carregar o valor da constante (código: `ldc.r8 token.getLexeme`).
 - para `true` (ação #105 – CORREÇÃO):
 - (a) empilhar na `pilha_tipos` o tipo correspondente, conforme TABELA DE TIPOS;
 - (b) gerar código objeto para carregar o valor da constante (código: `ldc.i4.1`).
 - para `false` (ação #106 – CORREÇÃO):
 - (a) empilhar na `pilha_tipos` o tipo correspondente, conforme TABELA DE TIPOS;
 - (b) gerar código objeto para carregar o valor da constante (código: `ldc.i4.0`).
 - para `constante_string` (ação #116):
 - (a) empilhar na `pilha_tipos` o tipo correspondente, conforme TABELA DE TIPOS;
 - (b) gerar código objeto para carregar o valor da constante em IL (verificar no anexo: instruções MSIL).
 - para o operador aritmético unário `"-"` (ação #117):
 - (a) gerar código objeto para efetuar a operação correspondente em IL (verificar na lista 7).
 - para os operadores aritméticos binários (ações #110, #111, #112, #113):
 - (a) desempilhar dois tipos da `pilha_tipos`, empilhar o tipo resultante da operação conforme indicado na TABELA DE TIPOS;
 - (b) gerar código objeto para efetuar a operação correspondente em IL (código: `add`, `sub`, `mul` ou `div`, respectivamente).
 - para os operadores relacionais (ações #108, #109 – CORREÇÃO):
 - (a) ação #108: guardar o operador relacional reconhecido em `operador_relacional`;
 - (b) ação #109: desempilhar dois tipos da `pilha_tipos`, empilhar o tipo resultante da operação conforme indicado na TABELA DE TIPOS;
 - (c) ação #109: gerar código objeto para efetuar a operação correspondente em IL conforme o operador relacional armazenado em `operador_relacional` (verificar no anexo: instruções MSIL, verificar em AVA > Aulas > Links > TOMAZELLI, Giancarlo. ...).
 - para o operador lógico unário `"!"` (ação #107):
 - (a) gerar código objeto para efetuar a operação correspondente em IL (verificar na lista 7).
 - para os operadores lógicos binários (ações #103, #104):
 - (a) desempilhar dois tipos da `pilha_tipos`, empilhar o tipo resultante da operação conforme indicado na TABELA DE TIPOS;
 - (b) gerar código objeto para efetuar a operação correspondente em IL (verificar no anexo: instruções MSIL).
- (4) A semântica do comando `<saída>` é a seguinte (ação #102):
- desempilhar um tipo da `pilha_tipos`;
 - valores do tipo `int` da linguagem fonte são tratados como `float64` em MSIL, portanto devem ser primeiramente convertidos para `int64` (código: `conv.i8`).
 - gerar código objeto para escrever o valor conforme o tipo desempilhado (código: `call void [mscorlib]System.Console.WriteLine(<tipo>)`, onde `<tipo>` pode se `int64`, `float64`, `string` ou `bool`).

(5) A semântica da declaração de constantes é a seguinte:

- ação #125: guardar `identificador (token.getLexeme)` na `lista_id` para uso posterior;
- ação #126: para cada `identificador` da `lista_id`:
 - (a) verificar se o `identificador` foi declarado, ou seja, se está na `tabela_simbolos`;
 - (b) em caso positivo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `token.getLexeme` já declarado (por exemplo: `_iarea` já declarado);
 - (c) em caso negativo, inserir o `identificador` com tipo e valor (`token.getLexeme`) correspondentes na `tabela_simbolos`, sendo que identificadores do tipo `int64` tem prefixo `_i`, do tipo `float64` tem prefixo `_f`, do tipo `string` tem prefixo `_s`, do tipo `bool` tem prefixo `_b`. Assumir que o valor é de tipo compatível com o `identificador`;
 - (d) limpar a `lista_id`, após o processamento.

(6) A semântica da declaração de variáveis é a seguinte:

- ação #125: guardar `identificador (token.getLexeme)` na `lista_id` para uso posterior;
- ação #127: para cada `identificador` da `lista_id`:
 - (a) verificar se o `identificador` foi declarado, ou seja, se está na `tabela_simbolos`;
 - (b) em caso positivo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `token.getLexeme` já declarado (por exemplo: `_iarea` já declarado);
 - (c) em caso negativo:
 - (i) inserir o `identificador` com tipo correspondente na `tabela_simbolos`, sendo que identificadores do tipo `int64` tem prefixo `_i`, do tipo `float64` tem prefixo `_f`, do tipo `string` tem prefixo `_s`, do tipo `bool` tem prefixo `_b`;
 - (ii) gerar código objeto para declarar o `identificador` (código: `.locals (tipo identificador)`);
 - (d) limpar a `lista_id`, após o processamento.

(7) A semântica do comando `<atribuição>` é a seguinte:

- ação #125: guardar `identificador (token.getLexeme)` na `lista_id` para uso posterior;
- ação #128:
 - (a) desempilhar o tipo da `<expressão>` da `pilha_tipos`;
 - (b) gerar o código objeto `dup n` vezes, onde `n` é igual a quantidade de identificadores da `lista_id` menos 1;
 - (c) para cada `identificador` da `lista_id`:
 - (i) verificar se o `identificador` foi declarado, ou seja, se está na `tabela_simbolos`;
 - (ii) em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `identificador não declarado` (por exemplo: `_iarea não declarado`);
 - (iii) em caso positivo: gerar código objeto para armazenar o valor da `<expressão>` em `identificador` (código: `stloc identificador`), lembrando que se `<expressão>` for do tipo `int64`, primeiramente deve ser convertida para `int64` (código: `conv.i8`). Assumir que o valor da `<expressão>` é de tipo compatível com o `identificador` e o `identificador` não é `identificador de constante`;
 - (d) limpar a `lista_id`, após o processamento.

(8) A semântica do comando `<entrada>` é a seguinte:

- a ação #130:
 - (a) gerar código objeto para carregar o valor da `constante_string` (verificar no anexo: instruções MSIL);
 - (b) gerar código objeto para escrever a constante (código: `call void [mscorlib]System.Console::Write (string)`);
- a ação #125: guardar `identificador (token.getLexeme)` na `lista_id` para uso posterior;
- a ação #129: para cada `identificador` da `lista_id`:
 - (a) verificar se o `identificador` foi declarado, ou seja, se está na `tabela_simbolos`;
 - (b) em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `identificador não declarado` (por exemplo: `_iarea não declarado`);
 - (c) em caso positivo:
 - (i) gerar código objeto para ler (da entrada padrão) um valor do tipo de `identificador`, sendo o tipo recuperado da `tabela_simbolos` (verificar no anexo: instruções MSIL);
 - (ii) gerar código objeto para armazenar o valor lido em `identificador` (código: `stloc identificador`). Assumir que o `identificador` não é `identificador de constante`;
 - (d) limpar a `lista_id`, após o processamento.

(9) A semântica do comando <seleção> é a seguinte:

- ação #118 (após <expressão>) deve:
 - (a) desempilhar o tipo da <expressão> e verificar se o tipo é bool;
 - (b) em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem expressão incompatível em comando de seleção;
 - (c) em caso positivo:
 - (i) criar um rótulo (novo_rotulo1);
 - (ii) gerar código objeto para desviar os comandos da cláusula if caso o resultado da avaliação da <expressão> for false (código: brfalse novo_rotulo1);
 - (iii) empilhar o rótulo (novo_rotulo1) na pilha_rotulos para resolução posterior.
- ação #120 (antes da cláusula else) deve:
 - (a) criar um rótulo (novo_rotulo2);
 - (b) gerar código objeto para desviar para o primeiro comando após o comando <seleção> (código: br novo_rotulo2);
 - (c) desempilhar novo_rotulo1;
 - (d) rotular o primeiro comando da <lista_comandos> associada à cláusula else (código: novo_rotulo1:);
 - (e) empilhar o rótulo (novo_rotulo2) na pilha_rotulos para resolução posterior.
- ação #119 (após o comando <seleção>) deve:
 - (a) desempilhar novo_rotulo2 (ou novo_rotulo1, para comando sem a cláusula else);
 - (b) rotular o primeiro comando após o comando <seleção> com rótulo desempilhado (conforme o caso, código: novo_rotulo1: ou novo_rotulo2:).

(10) A semântica do comando <repetição> while-do é a seguinte:

- ação #121 (antes do comando de <repetição>) deve:
 - (a) criar um rótulo (novo_rotulo1);
 - (b) rotular o primeiro comando da <expressão> código: novo_rotulo1:);
 - (c) empilhar o rótulo (novo_rotulo1) na pilha_rotulos para resolução posterior.
- ação #122 (após <expressão>) deve:
 - (a) desempilhar o tipo da <expressão> e verificar se o tipo é bool;
 - (b) em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem expressão incompatível em comando de repetição;
 - (c) em caso positivo:
 - (i) criar um rótulo (novo_rotulo2);
 - (ii) gerar código objeto para desviar para o primeiro comando após o comando <repetição> caso o resultado da avaliação da <expressão> for false (código: brfalse novo_rotulo2);
 - (iii) empilhar o rótulo (novo_rotulo2) na pilha_rotulos para resolução posterior.
- ação #123 (após o comando <repetição>) deve:
 - (a) desempilhar novo_rotulo2;
 - (b) desempilhar novo_rotulo1;
 - (c) gerar código objeto para desviar para o primeiro comando da <expressão> (código: br novo_rotulo1);
 - (d) rotular o primeiro comando após o comando <repetição> (código: novo_rotulo2:).

(11) A semântica do comando <repetição> repeat-while é a seguinte:

- ação #121 (antes do comando de <repetição>) deve:
 - (a) criar um rótulo (novo_rotulo1);
 - (b) rotular o primeiro comando da <expressão> (código: novo_rotulo1:);
 - (c) empilhar o rótulo (novo_rotulo1) na pilha_rotulos para resolução posterior.
- ação #124 (após <expressão>) deve:
 - (a) desempilhar o tipo da <expressão> e verificar se o tipo é bool;
 - (b) em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem expressão incompatível em comando de repetição;
 - (c) em caso positivo:
 - (i) desempilhar novo_rotulo1;
 - (ii) gerar código objeto para desviar para o primeiro comando do comando <repetição> caso o resultado da avaliação da <expressão> for true (código: brtrue novo_rotulo1).

OBSERVAÇÃO:

Para os comandos de <seleção> ou <repetição>, cada vez que um rótulo (novo_rótulo) é criado, deve ser colocado na pilha_rotulos para ser “resolvido” posteriormente.

Lembre-se que um programa pode possuir vários comandos de <seleção> ou <repetição>, aninhados ou não. Isto significa que devem ser criados rótulos (os rótulos são sequenciais) diferentes para cada comando.