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

```
<programa>    ::= #100 pr_begin <lista_instruções> pr_end #101 ;

<lista_instruçoes> ::= <instrucao> ";" | < instrucao> ";" <lista_instrucoes> ;
<instrucao>   ::= <declaracao_variavel> | <comando> ;

<declaracao_variavel> ::= <tipo> <lista_identificadores> #119 ;
<tipo>        ::= <simples> | <lista> ;
<simples>     ::= pr_bool #120 | pr_int #120 | pr_float #120 | pr_string #120 ;
<lista>       ::= pr_list "(" <simples> "," cte_int ")" ;
              // não será implementado nesse semestre
<lista_identificadores> ::= id #121 | id #121 "," <lista_identificadores> ;

<comando>     ::= id #121 <atribuição-add-delete> | <entrada> | <saida> | <selecao> | <repeticao> ;
<atribuição-add-delete> ::= <atribuicao> | <add-delete> ;
<atribuicao>  ::= "=" <expressao> #122 | "<-" <expressao> #122 ;
<add-delete>  ::= pr_add "(" <expressao> "," <expressao> ")"  | pr_delete "(" <expressao> ")" ;
              // não será implementado nesse semestre

<entrada>     ::= pr_read "(" <lista_entrada> ")" ;
<lista_entrada>::= <opcional> id #123 | <opcional> id #123 "," <lista_entrada>;
<opcional>    ::= cte_string #124 "," | î ;

<saida>       ::= pr_print "(" <lista_expressoes> ")" #118 ;
<lista_expressoes> ::= <expressao> #102 | <expressao> #102 "," <lista_expressoes> ;

<selecao>     ::= pr_if <expressao> #125 <lista_comandos> <else> pr_end #126 ;
<else>        ::= î | pr_else #127 <lista comandos> ;

<repeticao> ::= pr_do #128 <lista_comandos> pr_until <expressao> #129 ;

<lista_comandos>::= <comando> ";" | <comando> ";" <lista_comandos> ;

<expressao>   ::= <valor> <expressao_>  ;
<expressao_>  ::= î | pr_and <valor> #113 <expressao_> | pr_or <valor> #114 <expressao_> ;
<valor>       ::= <relacional> | pr_true #115 | pr_false #116 | pr_not <valor> #117 ;
<relacional>  ::= <aritmetica> <relacional_> ;
<relacional_>::= î | <operador_relacional> #111 <aritmetica> #112 ;
<operador_relacional>::= "==" | "~=" | "<" | ">" ;
<aritmetica>  ::= <termo> <aritmetica_>  ;
<aritmetica_>::= î | "+" <termo> #106 <aritmetica_> | "-" <termo> #107 <aritmetica_> ;
<termo>       ::= <fator> <termo_> ;
<termo_>      ::= î | "*" <fator> #108 <termo_> | "/" <fator> #109 <termo_> ;
<fator>       ::= id #130 <fator_> |
                  cte_int #103 |
                  cte_float #104 |
                  cte_string #105 |
                  "(" <expressao> ")" |
                  "+" <fator> |
                  "-" <fator> #110 ;
<fator_>      ::= î | pr_count | pr_size | pr_elementOf "(" <expressao> ")" ;
              // não será implementado nesse semestre
```

DESCRIÇÃO DOS REGISTROS SEMÂNTICOS: para executar a análise semântica e a geração de código é necessário fazer uso de registros semânticos (outros podem 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 **#111**, para uso posterior na ação **#112**
- **`tipo`** (inicialmente igual a `""`): usado para armazenar o tipo reconhecido pela ação **#120**, para uso posterior na ação **#119**
- **`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_identificadores`** (inicialmente vazia): usada para armazenar os identificadores reconhecidos pela ação **#121**, para uso posterior nas ações **#119** e **#122**
- **`tabela_simbolos`** (inicialmente vazia): usada para armazenar os identificadores declarados (ação **#119**). Cada linha da tabela tem dois campos: o identificador da variável declarada e o tipo correspondente.

identificador	tipo
de variável do tipo **bool**	**bool**
de variável do tipo **int**	**int64**
de variável do tipo **float**	**float64**
de variável do tipo **string**	**string**

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

operando$_1$	operando$_2$	operador	tipo resultante
id			de acordo com declaração de variáveis
cte_int			int64
cte_float			float64
cte_string			string
true			bool
false			bool
int64		operadores unários: + -	int64
float64		operadores unários: + -	float64
int64	int64	operadores binários: + - *	int64
int64 float64 float64	float64 int64 float64	operadores binários: + - *	float64
int64 ou float64	int64 ou float64	operadores binários: /	float64
int64 ou float64	int64 ou float64	== ~= < >	bool
string	string	== ~= < >	bool
bool		operador unário: **not**	bool
bool	bool	operadores binários: **and** **or**	bool

A verificação da compatibilidade de tipos não será implementada, ou seja, deve-se considerar que os operadores das expressões são de tipos compatíveis (é responsabilidade do programador escrever o programa corretamente). Mas, é necessário determinar o tipo de uma expressão, conforme indicado na tabela acima e descrito nas ações semânticas a seguir.

DESCRIÇÃO DA SEMÂNTICA:

(1) A ação #100 deve gerar código objeto na linguagem intermediária (IL) com o cabeçalho do programa objeto.

(2) A ação #101 deve gerar código objeto com as instruções para finalizar o programa objeto.

(3) A semântica do comando <saida> é a seguinte:

- ação #102:
  - desempilhar um tipo da **pilha_tipos**;
  - valores do tipo int64 da linguagem fonte são tratados como float64 em IL, portanto devem ser convertidos para int64 (código: conv.i8);
  - gerar código objeto para escrever o valor conforme o tipo desempilhado (código: call void [mscorlib]System.Console::Write(<tipo>), onde <tipo> pode ser int64, float64, string ou bool.

- ação #118:
  - gerar código objeto para escrever quebra de linha na saída padrão.

(4) A semântica para constantes em uma <expressao> é a seguinte:

- para cte_int (ação #103):
  - empilhar na **pilha_tipos** o tipo correspondente conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para carregar o valor da constante (código: ldc.i8 token.getLexeme), observando que valores do tipo int64 da linguagem fonte são tratados como float64 em IL, portanto devem ser convertidos para float64 (código: conv.r8) .

- para cte_float (ação #104):
  - empilhar na **pilha_tipos** o tipo correspondente conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para carregar o valor da constante (código: ldc.r8 token.getLexeme) .

- para cte_string (ação #105):
  - empilhar na **pilha_tipos** o tipo correspondente conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para carregar o valor da constante.

- para true (ação #115):
  - empilhar na **pilha_tipos** o tipo correspondente conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para carregar o valor da constante.

- para `false` (ação #116):
  - empilhar na **`pilha_tipos`** o tipo correspondente conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para carregar o valor da constante.

(5) A semântica para operadores aritméticos em uma `<expressao>` é a seguinte:
- para o operador aritmético unário `"-"` (ação #110):
  - gerar código objeto para efetuar a operação correspondente.

- para os operadores aritméticos binários (ações #106, #107, #108, #109):
  - desempilhar dois tipos da **`pilha_tipos`**, empilhar o tipo resultante da operação conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para efetuar a operação correspondente (código: `add`, `sub`, `mul` ou `div`, respectivamente).

(6) A semântica para os operadores relacionais em uma `<expressao>` é a seguinte:
- ação #111:
  - guardar em **`operador_relacional`** o operador relacional reconhecido.

- ação #112:
  - desempilhar dois tipos da **`pilha_tipos`**, empilhar o tipo resultante da operação conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para efetuar a operação correspondente ao operador relacional armazenado em **`operador_relacional`**.

(7) A semântica para os operadores lógicos em uma `<expressao>` é a seguinte:
- para o operador lógico unário **`not`** (ação #117):
  - gerar código objeto para efetuar a operação correspondente.

- para os operadores lógicos binários (ações #113, #114):
  - desempilhar dois tipos da **`pilha_tipos`**, empilhar o tipo resultante da operação conforme indicado na TABELA DE TIPOS;
  - gerar código objeto para efetuar a operação correspondente.

(8) A semântica da declaração de variáveis é a seguinte:
- ação #120:
  - guardar `<tipo>` (`token.getLexeme`) em **`tipo`** para uso posterior.

- ação #121:
  - guardar `id` (`token.getLexeme`) na **`lista_identificadores`** para uso posterior.

- ação #119: para cada `id` da **`lista_identificadores`**:
  - inserir o `id` na **`tabela_simbolos`** com o **`tipo`** guardado na ação #120, sendo que identificadores do tipo `int` na linguagem fonte são declarados como `int64` em IL; do tipo `float` na linguagem fonte são declarados como `float64` na IL; do tipo `string` na linguagem fonte são declarados como `string` na IL; do tipo `bool` na linguagem fonte são declarados como `bool` na IL;
  - gerar código objeto para declarar o `id` (código: `.locals (tipo id)`), onde o tipo do `id` pode ser: `int64`, `float64`, `string` ou `bool`;
  - limpar a **`lista_identificadores`**, após o processamento.

(9) A semântica do comando `<atribuição>` é a seguinte:
- ação #121:
  - guardar `id` (`token.getLexeme`) na **`lista_identificadores`** para uso posterior.

- ação #122:
  - desempilhar o tipo da `<expressao>` da **`pilha_tipos`**;
  - se `<expressao>` for do tipo `int64`, primeiramente deve ser convertida para `int64` (código: `conv.i8`);
  - recuperar `id` armazenado na **`lista_identificadores`**;
  - gerar código objeto para armazenar o valor da `<expressao>` em `id` (código: `stloc id`);
  - limpar a **`lista_identificadores`**, após o processamento.

(10) A semântica do comando `<entrada>` é a seguinte:
- a ação #123:
  - verificar se `id` (`token.getLexeme`) é do tipo `bool`:
    (a) em caso positivo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `id inválido para comando de entrada` (por exemplo: `logico inválido para comando de entrada`);

(b) em caso negativo:
(i) gerar código objeto para ler (da entrada padrão) um valor do tipo de `id`, onde o tipo do `id` é determinado quando da declaração de variáveis, podendo ser: `int64`, `float64` ou `string`;
(ii) gerar código objeto para armazenar o valor lido em `id` (código: `stloc id`).

- a ação #124:
– gerar código objeto para carregar o valor da `cte_string`;
– gerar código objeto para escrever a constante (código: `call void [mscorlib]System.Console::Write(string)`).

(11)A semântica de `id` (ação #130) em `<expressao>` é a seguinte:

- a ação #130:
– empilhar na **`pilha_tipos`** o tipo correspondente do `id` (`token.getLexeme`) conforme indicado na TABELA DE TIPOS, onde o tipo do `id` é determinado quando da declaração de variáveis, podendo ser: `bool`, `int64`, `float64` ou `string`;
– gerar código objeto para carregar o valor armazenado em `id`;
– se `id` for do tipo `int64`, gerar código objeto para converter o valor para `float64` (código: `conv.r8`).

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

- ação #125:
– desempilhar o tipo da `<expressao>` da **`pilha_tipos`** e verificar se o tipo é `bool`;
(a) em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `expressão incompatível em comando de seleção`;
(b) 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 `<expressao>` for `false` (código: `brfalse novo_rotulo1`);
(iii) empilhar o rótulo (`novo_rotulo1`) na **`pilha_rotulos`** para resolução posterior.

- ação #127:
– criar um rótulo (`novo_rotulo2`);
– gerar código objeto para desviar incondicionalmente para o primeiro comando após o comando `<seleção>` (código: `br novo_rotulo2`);
– desempilhar `novo_rotulo1` da **`pilha_rotulos`**;
– rotular o primeiro comando da `<lista_comandos>` associada à cláusula `else` com o rótulo desempilhado (código: `novo_rotulo1:`);
– empilhar o rótulo (`novo_rotulo2`) na **`pilha_rotulos`** para resolução posterior.

- ação #126:
– desempilhar um rótulo da **`pilha_rotulos`** (`rotulo_desempilhado`);
– rotular a próxima instrução do código objeto com o rótulo desempilhado (código: `rotulo_desempilhado:`).

(13)A semântica do comando `<repetição>` é a seguinte:

- ação #128:
– criar um rótulo (`novo_rotulo`);
– rotular a próxima instrução do código objeto com o rótulo criado (código: `novo_rotulo:`).
– empilhar o rótulo (`novo_rotulo`) na **`pilha_rotulos`** para resolução posterior.

- ação #129:
– desempilhar o tipo da `<expressao>` da **`pilha_tipos`** e verificar se o tipo é `bool`;
(a) em caso negativo, encerrar a execução e apontar erro semântico, indicando a linha e apresentando a mensagem `expressão incompatível em comando de repetição`;
(b) em caso positivo:
(i) desempilhar um rótulo da **`pilha_rotulos`** (`rotulo_desempilhado`);
(ii) gerar código objeto para desviar para o primeiro comando do comando `<repetição>` caso o resultado da avaliação da `<expressao>` seja `false` (código: `brfalse rotulo_desempilhado`).

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, lembrando 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.