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

```
<programa>            ::= #15 main <declarações> <módulos> #16 begin <corpo> end #17

<declarações>         ::= ... // não será implementado, manter as regras sintáticas
<módulos>             ::= ... // não será implementado, manter as regras sintáticas

<corpo>               ::= <declaração_variáveis> <lista_comandos>
<declaração_variáveis>::= î | <variável> <declaração_variáveis>
<variável>            ::= <tipo> #30 : <lista_identificadores> #31
<tipo>                ::= int | float | bool | char | string
<lista_identificadores>::= identificador #32 | identificador #32 , <lista_identificadores>
<lista_comandos>      ::= <comando> . <lista_comandos> | <comando> .

<comando>             ::= <atribuição> | <entrada> | <saída> | <seleção> | <repetição> | <retorno>

<atribuição>          ::= identificador #32 <operador> #36 <expressão> #34
<operador>            ::= = | += | -=

<entrada>             ::= read ( <lista_identificadores> #35 )

<saída>               ::= write ( <lista_expressões> )
<lista_expressões>    ::= <expressão> #14 , <lista_expressoes> | <expressão> #14

<seleção>             ::= #37 ( <expressão> ) ifTrueDo #38 <lista_comandos> <ifFalseDo> end #39
<ifFalseDo>           ::= î | #40 ifFalseDo <lista_comandos>

<repetição>           ::= #37 ( <expressão> ) whileTrueDo  #41 <lista_comandos> end #42 |
                          #37 ( <expressão> ) whileFalseDo #41 <lista_comandos> end #42

<retorno>             ::= ... // não será implementado, manter as regras sintáticas

<expressao>           ::= <expressão> and <elemento> #18 | <expressão> or <elemento> #19 | <elemento>
<elemento>            ::= <relacional> | true #11 | false #12 | not <elemento> #13
<relacional>          ::= <aritmética> <operador_relacional> #9 <aritmética> #10 | <aritmética>
<operador_relacional>::= == | != | < | <= | > | >=
<aritmética>          ::= <aritmética> + <termo> #1 | <aritmética> - <termo> #2 | <termo>
<termo>               ::= <termo> * <fator> #3 | <termo> / <fator> #4 | <fator>
<fator>               ::= identificador #33 <fator_> |
                          cte_int      #5     |
                          cte_float    #6     |
                          cte_caracter #20    |
                          cte_string   #21    |
                          ( <expressao> )     |
                          + <fator>    #7     |
                          - <fator>    #8
<fator_>              ::= ... // não será implementado, manter as regras sintáticas
```

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 #9`**, para uso posterior na **`ação #10`**.
- ✓ • **`código`**: usado para armazenar o código objeto gerado.
- ✓ • **`pilha_de_tipos`** (inicialmente vazia): usada para determinar o tipo de uma expressão durante a compilação do programa.
- ✓ • **`pilha_de_rótulos`** (inicialmente vazia): usada na análise dos comandos de seleção e de repetição.
- ✓ • **`lista_de_identificadores`** (inicialmente vazia): usada para armazenar os identificadores reconhecidos pela **`ação #32`**, para uso posterior nas **`ações #31`**, **`#34`**, **`#35`** e **`#36`**.
- ✓ • **`tabela_símbolos`** (inicialmente vazia): usada para armazenar informações sobre os identificadores declarados. Cada linha da tabela tem dois campos:

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

DESCRIÇÃO DAS VERIFICAÇÕES SEMÂNTICAS:

- ✓ A linguagem é *case sensitive*.
- ✓ Qualquer `identificador` só pode ser declarado uma vez (a partir do não terminal `<declaração_variáveis>`).
- ✓ Qualquer `identificador` só pode ser usado (a partir do não terminal `<lista_comandos>`) se for declarado.
- ✓ O tipo de uma `<expressão>` deve ser determinado da seguinte forma:

operando$_1$	operando$_2$	operador	tipo da expressão resultante
identificador			conforme a declaração da variável, podendo ser **int64, float64, bool, string**
cte_int			**int64**
cte_float			**float64**
cte_caracter			**string**
cte_string			**string**
true			**bool**
false			**bool**
int64		operadores unários : **+ -**	**int64**
int64	**int64**	operadores binários: **+ - ***	**int64**
int64	**int64**	operador binário: **/**	**float64**
float64		operadores unários : **+ -**	**float64**
int64 ou **float64**	**int64** ou **float64**	operadores binários: **+ - * /** pelo menos um operando do tipo **float64**	**float64**
int64 ou **float64**	**int64** ou **float64**	**== != < <= > >=**	**bool**
string	**string**	**== != < <= > >=**	**bool**
bool		**not**	**bool**
bool	**bool**	**and or**	**bool**

Operadores e tipos não previstos na tabela anterior indicam que a operação correspondente não pode ser executada com os tipos em questão. Assim, por exemplo, **`true and "teste"`** deve gerar um erro semântico (*tipos incompatíveis em expressão lógica*); **`true == true`** deve gerar um erro semântico (*tipos incompatíveis em expressão relacional*); **`true`** `+ 10` deve gerar um erro semântico (*tipos incompatíveis em expressão aritmética)*.

- ✓ Só serão implementadas algumas verificações semânticas, tais como compatibilidade de tipos (em expressões), declaração e uso de identificadores. As mensagens de erro e as ações que devem fazer as validações são:

✓ 1. tipo(s) incompatível(is) em expressão aritmética: **ações #1**, **#2**, **#3**, **#4**, **#7**, **#8**;

✓ 2. tipos incompatíveis em expressão relacional: **ação #10**;

✓ 3. tipo(s) incompatível(is) em expressão lógica: **ações #13**, **#18**, **#19**;

✓ 4. identificador não declarado: **ações #33**, **#34**, **#35**, **#36**;

✓ 5. identificador já declarado: **ação #31**.

- ✓ Quanto ao uso de identificadores em comandos de `<atribuição>`, deve-se considerar que os identificadores serão corretamente usados, ou seja, **não** é necessário implementar verificação de compatibilidade de tipos para comando de atribuição. Também não é necessário validar o tipo de uma `<expressão>` de um comando de seleção ou de repetição.

DESCRIÇÃO DA SEMÂNTICA:

- ✓ A semântica de uma expressão (`<expressão>`) é a seguinte:
  - para `identificador` (**ação #33**): (1) gerar código para carregar o valor armazenado no `identificador` (`ldloc`); (2) empilhar o tipo da variável na **`pilha_de_tipos`**;
  - para constantes (`cte_int` - **ação #5**, `cte_float` - **ação #6**, `cte_string` - **ação #21**, **`true`** - **ação #11**, **`false`** - **ação #12**): (1) gerar código para carregar o valor da constante; (2) empilhar tipo da constante na **`pilha_de_tipos`**;
  - para `cte_caracter` (`\s`, `\n`, `\t`) – **ação #20**: (1) gerar código para carregar o valor correspondente da constante (`" "`, `"\n"`, `"\t"`, respectivamente); (2) empilhar tipo da constante na **`pilha_de_tipos`**;
  - para os operadores (lógicos – **ação #13**, **#18**, **#19**, relacionais – **ação #10**, aritméticos – **ação #1**, **#2**, **#3**, **#4**, **#7**, **#8**): (1) gerar código para efetuar a operação correspondente; (2) empilhar tipo resultante da operação na **`pilha_de_tipos`**.
- ✓ A semântica do comando `<saída>` (**ação #14**) é a seguinte: (1) gerar código para escrever (na saída padrão) o resultado da avaliação da `<expressão>` conforme seu tipo.

- ✓ A semântica da <declaração_variáveis> é a seguinte (**ação #31**): (1) incluir cada identificador da <lista_identificadores> na **tabela_símbolos** com o tipo correspondente (em MSIL: int64, float64, bool ou string); (2) gerar código para alocar memória para o(s) identificador(es) declarado(s).

- ✓ A semântica do comando <atribuição> é a seguinte (**ação #36**, **#34**): (1) para os operadores **+=** e **-=** gerar código para carregar o valor armazenado no identificador (ldloc); (2) para o operador **=**, gerar código para atribuir o resultado da avaliação da <expressão> ao identificador (stloc); (3) para o operador **+=**, gerar código para somar (add) o valor armazenado no identificador ao resultado da avaliação da <expressão> e atribuir o resultado ao identificador (stloc); (4) para o operador **-=**, gerar código para subtrair (sub) do valor armazenado no identificador o resultado da avaliação da <expressão> e atribuir o resultado ao identificador (stloc).

- ✓ A semântica do comando <entrada> é a seguinte (**ação #35**): (1) para cada identificador da <lista_identificadores>, gerar código para ler (da entrada padrão) um valor; (2) gerar código para armazenar o valor lido no identificador correspondente.

- ✓ A semântica do comando <seleção> é a seguinte (**ação #38**, **#39**, **#40**): (1) gerar código para verificar se o resultado da avaliação da <expressão> é false e desviar para o primeiro comando da <lista_comandos> associada à cláusula **ifFalseDo**, se existir, ou para o primeiro comando após o **end**; (2) se cláusula **ifFalseDo** existir, gerar código para desviar para o primeiro comando após o **end** e rotular apropriadamente o primeiro comando da <lista_comandos> associada à cláusula **ifFalseDo**; (3) rotular apropriadamente o primeiro comando após o **end**.

- ✓ A semântica do comando <repetição> **whileTrueDo** é a seguinte (**ação #37**, **#41**, **#42**): (1) rotular o primeiro comando da <expressão>; (2) gerar código para verificar se o resultado da avaliação da <expressão> é false e desviar para o primeiro comando após o **end**; (3) gerar código para desviar incondicionalmente para o primeiro comando da <expressão> e rotular apropriadamente o primeiro comando após o **end**.

- ✓ A semântica do comando <repetição> **whileFalse** é a seguinte (**ação #37**, **#41**, **#42**): (1) rotular o primeiro comando da <expressão>; (2) gerar código para verificar se o resultado da avaliação da <expressão> é true e desviar para o primeiro comando após o **end**; (3) gerar código para desviar incondicionalmente para o primeiro comando da <expressão> e rotular apropriadamente o primeiro comando após o **end**.

  DICA: cada vez que um rótulo (label) é gerado, pode ser colocado na **pilha_de_rótulos** 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 labels (os rótulos são sequenciais) diferentes para cada comando.

EXEMPLO DE PROGRAMA FONTE / OBJETO

programa fonte: **teste_03.txt**

```
main begin
 int: lado, area.
 read (lado).
 area = 0.
 (lado > 0) ifTrueDo area = lado * lado. end.
 write (area).
end
```

programa objeto: **teste_03.il**

```
.assembly extern mscorlib {}
.assembly _codigo_objeto{}
.module   _codigo_objeto.exe

.class public _UNICA{
.method static public void _principal() {
  .entrypoint
  .locals (int64 lado, int64 area)
  call string [mscorlib]System.Console::ReadLine()
  call int64 [mscorlib]System.Int64::Parse(string)
  stloc lado
  ldc.i8 0
  conv.r8
  conv.i8
  stloc area
//início do código gerado pela ação #37
  label1:
//fim
  ldloc lado
  conv.r8
  ldc.i8 0
  conv.r8
  cgt
//início do código gerado pela ação #38
  brfalse label2
//fim
  ldloc lado
  conv.r8
  ldloc lado
  conv.r8
  mul
  conv.i8
  stloc area
//início do código gerado pela ação #39
  label2:
//fim
  ldloc area
  call void [mscorlib]System.Console::Write(int64)
  ret
}
}
```

programa fonte: **teste_04.txt**

```
main begin
 float: valor.
 read (valor).
 (valor > 0.0) ifTrueDo  write ("maior").
               ifFalseDo write ("menor ou igual"). end.
end
```

programa objeto: **teste_04.il**

```
.assembly extern mscorlib {}
.assembly _codigo_objeto{}
.module   _codigo_objeto.exe

.class public _UNICA{
.method static public void _principal() {
  .entrypoint
  .locals (float64 valor)
  call string [mscorlib]System.Console::ReadLine()
  call float64[mscorlib]System.Double::Parse(string)
  stloc valor
//início do código gerado pela ação #37
  label1:
//fim
  ldloc valor
  ldc.r8 0.0
  cgt
//início do código gerado pela ação #38
  brfalse label2
//fim
  ldstr "maior"
  call void [mscorlib]System.Console::Write(string)
//início do código gerado pela ação #40
  br label3
  label2:
//fim
  ldstr "menor ou igual"
  call void [mscorlib]System.Console::Write(string)
//início do código gerado pela ação #39
  label3:
//fim
  ret
}
}
```

programa fonte: **teste_05.txt**

```
main begin
  int: valor.
  read (valor).
  (valor < 0)  whileTrueDo   read (valor).      end.
  (valor == 0) whileFalseDo  write (valor, \n).
                             valor -= 1.        end.
end
```

programa objeto: **teste_05.il**

```
.assembly extern mscorlib {}
.assembly _codigo_objeto{}
.module   _codigo_objeto.exe

.class public _UNICA{
.method static public void _principal() {
  .entrypoint
  .locals (int64 valor)
  call string [mscorlib]System.Console::ReadLine()
  call int64  [mscorlib]System.Int64::Parse(string)
  stloc valor
//início do código gerado pela ação #37
  label1:
//fim
  ldloc valor
  conv.r8
  ldc.i8 0
  conv.r8
  clt
//início do código gerado pela ação #41
  brfalse label2
//fim
  call string [mscorlib]System.Console::ReadLine()
  call int64  [mscorlib]System.Int64::Parse(string)
  stloc valor
//início do código gerado pela ação #42
  br label1
  label2:
//fim
//início do código gerado pela ação #37
  label3:
//fim
  ldloc valor
  conv.r8
  ldc.i8 0
  conv.r8
  ceq
//início do código gerado pela ação #41
  brtrue label4
//fim
  ldloc valor
  conv.r8
  conv.i8
  call void [mscorlib]System.Console::Write(int64)
  ldstr "\n"
  call void [mscorlib]System.Console::Write(string)
  ldloc valor
  conv.r8
  ldc.i8 1
  conv.r8
  sub
  conv.i8
  stloc valor
//início do código gerado pela ação #42
  br label3
  label4:
//fim
  ret
}
}
```