ESQUEMA DE TRADUÇÃO nº2 - completo
(para implementação do analisador semântico e gerador de código)

1º passo: abra o arquivo que contém as especificações dos *tokens*, das regras sintáticas da linguagem e a numeração das ações semânticas do esquema de tradução nº1.

2º passo: coloque a numeração das ações semânticas na gramática, considerando o esquema de tradução abaixo, que possui não determinismo à esquerda.

```
<programa>::= #15 def <definição_tipos> <definição_constantes> <declaração_variáveis>
              execute <lista_comandos> #16

<definição_tipos>::= ... não será implementada nesse semestre ...

<definição_constantes>::= î | consts <lista_id> = <valor> #32 <lista_constantes>
<lista_constantes>::= î | <lista_id> = <valor> #32  <lista_constantes>
<lista_id>::= identificador #22 | identificador #22 , <lista_id>
<valor>::= cte_int | cte_float | cte_str | true | false

<declaração_variáveis>::= î | var <lista_id> : <tipo> #21 #23  <lista_variáveis>
<lista_variáveis>::= î | <lista_id> : <tipo> #21 #23 <lista_variáveis>
<tipo>::= int | float |  str | bool | identificador

<lista comandos>    ::= <comando> | <comando> <lista_comandos>
<comando>           ::= <atribuição> | <manipulação> | <entrada> | <saída> | <seleção> | <repetição>

<atribuição>        ::= identificador #22 := <expressão> #26

<manipulação>       ::= ... não será implementado nesse semestre ...

<entrada>           ::= input ( <lista_id> #24 )

<saída>             ::= print ( <lista_expressões> ) | println ( <lista_expressões> ) #17
<lista_expressões>  ::= <expressão> #14 | <expressão> #14 , <lista_expressões>

<seleção>           ::= #27 ( <expressão> ) ifTrue #28 : <lista_comandos> end #29     |
                        #27 ( <expressão> ) ifTrue #28 : <lista_comandos>
                                           #30 ifFalse: <lista_comandos> end #29     |
                        #27 ( <expressão> ) ifFalse #28: <lista_comandos> end #29


<repetição>         ::= #27 ( <expressão> ) whileTrue #28: <lista_comandos> end #31   |
                        #27 ( <expressão> ) whileFalse #28: <lista_comandos> end #31

<expressão>          ::= <elemento> <expressão_>
<expressão_>         ::= ε | && <elemento> #18 <expressão_> | || <elemento> #19 <expressão_>
<elemento>           ::= <relacional> | true #11 | false #12 | ! <elemento> #13
<relacional>         ::= <aritmetica> <relacional_>
<relacional_>        ::= ε | <operador_relacional> #9 <aritmetica> #10
<operador_relacional>::= = | != | < | <= | > | >=
<aritmética>         ::= <termo> <aritmética_>
<aritmética_>        ::= ε | + <termo> #1 <aritmética_> | - <termo> #2 <aritmética_>
<termo>              ::= <fator> <termo_> ;
<termo_>             ::= ε | * <fator> #3 <termo_> | / <fator> #4 <termo_>
<fator>              ::= identificador #25  |
                         identificador get ... não será implementado nesse semestre ... |
                         cte_int   #5    |
                         cte_float #6    |
                         cte_str   #20   |
                         ( <expressão> ) |
                         + <fator> #7    |
                         - <fator> #8
```

3º passo: uma vez que a gramática esteja alterada e as ações semânticas corretamente colocadas, gere novamente os analisadores léxico, sintático e semântico para refletir na implementação as alterações feitas. Observa-se que, em geral, o único código alterado pelo GALS é o das constantes (em Java - `ScannerConstants.java`, `ParserConstants.java`, `Constants.java`).

4º passo: implemente os registros semânticos e as ações semânticas que constituem o analisador semântico e o gerador de código, conforme explicado em aula. Algumas das ações semânticas especificadas em sala, deverão ser alteradas para atender a semântica da linguagem 2018.1, conforme descrito a seguir.

5º passo: valide o código objeto gerado. Para tanto, utilize o `ilasm` para gerar o executável a partir do código objeto gerado e, em seguida, execute o arquivo executável.

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). Tem-se:

- **operador**: 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_tipos**: usada para determinar o tipo de uma <expressão>.
- **lista_de_identificadores** (inicialmente vazia): usada para armazenar os identificadores reconhecidos pela **ação #22**, para uso posterior em outras ações semânticas.
- **pilha_rótulos** (inicialmente vazia): usada na análise dos comandos de seleção e de repetição.
- **tabela_símbolos** (inicialmente vazia): usada para armazenar informações sobre os identificadores declarados:

identificador	classe	tipo (em MSIL)	valor
da constante **false**	**c**	**bool**	valor
da constante **true**	**c**	**bool**	valor
de cte_int	**c**	**int64**	valor
de cte_float	**c**	**float64**	valor
de cte_str	**c**	**string**	valor
de variável do tipo **bool**	**v**	**bool**	-
de variável do tipo **int**	**v**	**int64**	-
de variável do tipo **float**	**v**	**float64**	-
de variável do tipo **string**	**v**	**string**	-

onde:

- **identificador** é o identificador declarado na <definição_constantes> ou na <declaração_variáveis>
- **classe** indica se o identificador é de constante (**c**) ou de variável (**v**);
- **tipo** indica o tipo (em MSIL) da constante ou da variável
- **valor** é o valor da constante quando da <definição_constantes>

DESCRIÇÃO DAS VERIFICAÇÕES SEMÂNTICAS: tem-se:

✓ O tipo de uma <expressão> deve ser determinado da seguinte forma:

$operando_1$	$operando_2$	operador	tipo da expressão resultante
cte_int			**int64**
cte_float			**float64**
cte_str			**string**
true			**bool**
false			**bool**
identificador **(na ação #25)**			**int64** ou **float64** ou **string** ou **bool,** conforme declaração de variáveis ou a definição de constantes
int64		operadores unários : **+ -**	**int64**
int64	**int64**	operadores binários: **+ - * /**	**int64**
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		**!** (não)	**bool**
bool	**bool**	**&&** (e) **||** (ou)	**bool**

Operadores e tipos não previstos na tabela anterior indicam que a operação correspondente não pode ser executada. Assim, por exemplo, `10 = "oi"` deve gerar um erro semântico (*tipos incompatíveis em expressão relacional)*.

✓ A linguagem é *case sensitive*.

✓ Qualquer identificador só pode ser declarado uma vez.

✓ Qualquer identificador só pode ser usado se for declarado.

✓ Só serão implementadas algumas verificações semânticas, sendo as mensagens de erro e a ação que deve validar indicadas na sequência. As mensagens de erro para compatibilidade de tipos (em expressões) devem ser: tipo incompatível em operação aritmética unária (**ação #7**, **#8**); tipos incompatíveis em operação aritmética binária (**ação #1**, **#2**, **#3**, **#4**); tipos incompatíveis em operação relacional (**ação #10**); tipo incompatível em operação lógica unária (**ação #13**); tipos incompatíveis em operação lógica binária (**ação #18**, **#19**). As mensagens de erro para identificadores devem ser: identificador não declarado (**ação #24**, **#25**, **#26**); identificador já declarado (**ação #23**, **#32**); tipo incompatível em comando de atribuição (**ação #26**), caso o tipo do identificador seja diferente do tipo da <expressão>.

- ✓ Quanto ao uso de identificadores, deve-se considerar que os identificadores de variáveis e constantes serão corretamente usados, ou seja, **não** é necessário implementar: verificação do uso correto de identificadores em comando de atribuição, entrada ou saída.

DESCRIÇÃO DA SEMÂNTICA: tem-se:

- ✓ A semântica de uma expressão (`<expressão>`) é a seguinte:
  - para as constantes (`cte_int` – ação **#5**, `cte_float` – ação **#6**, `cte_str` – ação **#20**, **`true`** – ação **#11**, **`false`** – ação **#12**): (1) empilhar o tipo da constante na **`pilha_tipos`**; (2) gerar código para carregar o valor da constante,
  - para os operadores (lógicos, relacionais, aritméticos), (1) efetuar a verificação de tipos conforme descrito na tabela anterior; (2) gerar código para efetuar a operação correspondente;
  - para `identificador` (**ação #25** – sugestão: adaptar a ação #25 especificada em aula): SE for identificador de VARIÁVEL: (1) efetuar a verificação semântica descrita anteriormente; (2) empilhar o tipo da variável na **`pilha_tipos`**; (3) gerar código para carregar o valor armazenado na variável; SE for identificador de CONSTANTE: (1) efetuar a verificação semântica descrita anteriormente; (2) empilhar o tipo da constante na **`pilha_tipos`**; (3) gerar código para carregar o valor da constante já armazenado na **`tabela_simbolos`.**
- ✓ A semântica da `<definição_constantes>` é a seguinte (**ação #32**): (1) efetuar a verificação semântica descrita anteriormente; (2) incluir cada `identificador` da `<lista_id>` na **`tabela_símbolos`** com a classe `c` (para indicar que é constante), o tipo correspondente (em MSIL) e o valor, conforme declarado.
- ✓ A semântica da `<declaração_variáveis>` é a seguinte (**ação #23** – sugestão: adaptar a ação #23 especificada em aula): (1) efetuar a verificação semântica descrita anteriormente; (2) incluir cada `identificador` da `<lista_id>` na **`tabela_símbolos`** com a classe `v` (para indicar que é variável) e o tipo correspondente (em MSIL), conforme declarado; (3) gerar código para alocar memória para o(s) `identificador`(es) declarado(s).
- ✓ A semântica do comando `<atribuição>` é a seguinte (**ação #26**): (1) efetuar a verificação semântica descrita anteriormente; (2) gerar código para atribuir o resultado da avaliação da `<expressão>` ao `identificador`.
- ✓ A semântica do comando `<entrada>` é a seguinte (**ação #24**): (1) efetuar a verificação semântica descrita anteriormente; (2) para cada `identificador` da `<lista_id>`, gerar código para ler (da entrada padrão) um valor; (3) gerar código para armazenar o valor lido no `identificador` correspondente.
- ✓ A semântica do comando **`print`** é a seguinte: gerar código para escrever (na saída padrão) o resultado da avaliação de cada `<expressão>` da `<lista_expressões>`.
- ✓ A semântica do comando **`println`** é a seguinte (ou seja, a ação **#17** deve): (1) gerar código para escrever (na saída padrão) o resultado da avaliação de cada `<expressão>` da `<lista_expressões>`, (2) gerar código para escrever `\n` (na saída padrão).
- ✓ A **ação #27** deve ser gerar um rótulo.
- ✓ A semântica do comando `<seleção>` é a seguinte (**ação #28**, **#29**, **#30**): SE for o comando `ifTrue`: gerar código para verificar se o resultado da avaliação da `<expressão>` é falso e, em caso negativo, executar apenas os comandos da `<lista_comandos>` associada à cláusula **`ifTrue`**; em caso positivo, gerar código para executar apenas os comandos da `<lista_comandos>` da cláusula **`ifFalse`**, se existir. SE for o comando `ifFalse`: gerar código para verificar se o resultado da avaliação da `<expressão>` é verdadeiro e, em caso negativo, executar apenas os comandos da `<lista_comandos>` associada à cláusula **`ifFalse`**.
- ✓ A semântica do comando `<repetição>` é a seguinte (**ação #28**, **#31**): SE for o comando `whileTrue`: gerar código para verificar se o resultado da avaliação da `<expressão>` é falso e, em caso negativo, executar apenas os comandos da `<lista_comandos>` associada à cláusula **`whileTrue`**; gerar código para executar novamente a avaliação da `<expressão>`; SE for o comando `whileFalse`: gerar código para verificar se o resultado da avaliação da `<expressão>` é verdadeiro e, em caso negativo, executar apenas os comandos da `<lista_comandos>` associada à cláusula **`whileFalse`**; gerar código para executar novamente a avaliação da `<expressão>`.

EXEMPLOS DE PROGRAMA FONTE / OBJETO

programa fonte: **teste_02.txt**

```
def
var lado, area: float
execute
  print("digite um valor para lado: ")
  input(lado)
  area:= lado * lado
  print(area)
```

programa objeto: **teste_02.il**

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

.class public _UNICA{
.method static public void _principal() {
  .entrypoint
  .locals (float64 lado, float64 area)
  ldstr "digite um valor para lado: "
  call void
[mscorlib]System.Console::Write(string)
  call string [mscorlib]System.Console::ReadLine()
  call float64
[mscorlib]System.Double::Parse(string)
  stloc lado
  ldloc lado
  ldloc lado
  mul
  stloc area
  ldloc area
  call void
[mscorlib]System.Console::Write(float64)
  ret
  }
}
```

programa fonte: **teste_03.txt**

```
def
var lado, area: float
execute
  input(lado)
  (lado > 0) ifTrue:
    area:= lado * lado
  ifFalse:
    print("erro: valor inválido para lado - ")
    area:= 0,0
  end
  print("área: ", area)
```

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 (float64 lado, float64 area)
  call string [mscorlib]System.Console::ReadLine()
  call float64
[mscorlib]System.Double::Parse(string)
  stloc lado
//início do código gerado pela ação #27
  label1:
//fim
  ldloc lado
  ldc.i8 0
  conv.r8
  cgt
//início do código gerado pela ação #28
  brfalse label2
//fim
  ldloc lado
  ldloc lado
  mul
  stloc area
//início do código gerado pela ação #30
  br label3
  label2:
//fim
  ldstr "erro: valor inválido para lado - "
  call void
[mscorlib]System.Console::Write(string)
  ldc.r8 0.0
  stloc area
//início do código gerado pela ação #29
  label3:
//fim
  ldstr "área: "
  call void
[mscorlib]System.Console::Write(string)
  ldloc area
  call void
[mscorlib]System.Console::Write(float64)
  ret
  }
}
```

programa fonte: **teste_04.txt**

```
def
var CH: int
execute
  print("qual a CH de compiladores? ")
  input(CH)
  (CH < 18) whileTrue:
    print("qual a CH de compiladores? ")
    input(CH)
  end
  print("total de créditos: ", CH / 18)
```

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 (int64 CH)
  ldstr "qual a CH de compiladores? "
  call void
[mscorlib]System.Console::Write(string)
  call string [mscorlib]System.Console::ReadLine()
  call int64 [mscorlib]System.Int64::Parse(string)
  stloc CH
//início do código gerado pela ação #27
  label1:
//fim
  ldloc CH
  conv.r8
  ldc.i8 18
  conv.r8
  clt
//início do código gerado pela ação #28
  brfalse label2
//fim
  ldstr "qual a CH de compiladores? "
  call void
[mscorlib]System.Console::Write(string)
  call string [mscorlib]System.Console::ReadLine()
  call int64 [mscorlib]System.Int64::Parse(string)
  stloc CH
//início do código gerado pela ação #31
  br label1
  label2:
//fim
  ldstr "total de créditos:  "
  call void
[mscorlib]System.Console::Write(string)
  ldloc CH
  conv.r8
  ldc.i8 18
  conv.r8
  div
  conv.i8
  call void [mscorlib]System.Console::Write(int64)
  ret
  }
}
```