

Implemente em linguagem *Haskell*, usando a biblioteca de combinadores monádicos *Parsec*, um analisador sintático para a linguagem definida abaixo, o código intermediário gerado deve ser uma árvore sintática abstrata representada pelos tipos algébricos de dados definidos no final deste documento. A linguagem deve manipular pelo menos três tipos de dados: *int*, *double* e *string*. As produções para expressões (lógicas, relacionais e aritméticas) devem ser definidas.

<Programa>	→ <ListaFuncoes> <BlocoPrincipal>
<ListaFuncoes>	→ <Função> <ListaFuncoes>   $\epsilon$
<Funcao>	→ <TipoRetorno> <b>id</b> (<DeclParametros>) <BlocoPrincipal>
<TipoRetorno>	→ <Tipo>   <b>void</b>
<DeclParametros>	→ <Tipo> <b>id</b> <Parametros>   $\epsilon$
<Parametros>	→ , <DeclParametros>   $\epsilon$
<BlocoPrincipal>	→ {<BlocoPrincipal'>}
<BlocoPrincipal'>	→ <Declaracoes> <ListaCmd>
<Declaracoes>	→ <Tipo> <Listald>; <Declaracoes>   $\epsilon$
<Tipo>	→ <b>int</b>   <b>string</b>   <b>double</b>
<Listald>	→ <b>id</b> <Listald'>
<Listald'>	→ , <Listald>   $\epsilon$
<Bloco>	→ { <ListaCmd> }
<ListaCmd>	→ <Comando> <ListaCmd>   $\epsilon$
<ChamadaFuncao>	→ <b>id</b> (<ListaParametros>)

$\langle \text{ListaParametros} \rangle \rightarrow \langle \text{ListaParametros}' \rangle$   
 $\quad \quad \quad | \epsilon$

$\langle \text{ListaParametros}' \rangle \rightarrow \langle \text{Expressao} \rangle \langle \text{ListaParametros}'' \rangle$

$\langle \text{ListaParametros}'' \rangle \rightarrow , \langle \text{ListaParametros}' \rangle$   
 $\quad \quad \quad | \epsilon$

$\langle \text{Comando} \rangle \rightarrow$  **return**  $\langle \text{Expressao} \rangle$ ;  
 $\quad \quad \quad |$  **if** ( $\langle \text{ExpressaoLogica} \rangle$ )  $\langle \text{Bloco} \rangle$   $\langle \text{Senao} \rangle$   
 $\quad \quad \quad |$  **while** ( $\langle \text{ExpressaoLogica} \rangle$ )  $\langle \text{Bloco} \rangle$   
 $\quad \quad \quad |$  **id** =  $\langle \text{Expressao} \rangle$ ;  
 $\quad \quad \quad |$  **print** ( $\langle \text{Expressao} \rangle$ );  
 $\quad \quad \quad |$  **read** (**id**);  
 $\quad \quad \quad |$   $\langle \text{ChamadaFunção} \rangle$ ;

$\langle \text{Senao} \rangle \rightarrow$  **else**  $\langle \text{Bloco} \rangle$   
 $\quad \quad \quad | \epsilon$

- Uma expressão relacional tem como termos expressões aritméticas e envolve os operadores: <, >, <=, >=, ==, /=.
- Uma expressão lógica tem como termos expressões relacionais e envolve os seguintes operadores: && (conjunção), || (disjunção) e ! (negação). O operador unário ! possui a maior precedência, seguido pelo operador binário && e com menor precedência o operador binário ||. A associatividade dos operadores && e || são da esquerda para a direita.
- Os operadores aritméticos (+, -, \*, /) têm associatividade da esquerda para direita e a precedência usual.
- Uma expressão aritmética tem como termos: identificadores de variáveis, constantes inteiras, constantes com ponto flutuante ou chamadas de funções.
- Nas expressões lógicas ou aritméticas os parênteses alteram a ordem de avaliação.
- Os *tokens* identificador (**id**), constante inteira, constante com ponto flutuante e constante cadeia de caracteres (**literal**) devem ser definidos como ocorrem usualmente em linguagens de programação.

Na segunda fase do trabalho será feita a análise semântica (verificação de tipos) e na terceira a geração de código a partir da representação intermediária.

## Representação intermediária

type Id = String

data Type = TDouble | TInt | TString | TVoid  
deriving Show

data TCons = CDouble Double | CInt Int | CString String  
deriving Show

data Expr = Expr :+: Expr | Expr :-: Expr | Expr :\*: Expr | Expr :/: Expr |  
Neg Expr | Const TCons | IdVar String | Chamada Id [Expr] | Lit String  
deriving Show

data ExprR = Expr :==: Expr | Expr :/=: Expr | Expr :<: Expr |  
Expr :>: Expr | Expr :<=: Expr | Expr :>=: Expr deriving Show

data ExprL = ExprL :&: ExprL | ExprL :|: ExprL | Not ExprL | Rel ExprR  
deriving Show

data Var = Id :#: Type deriving Show

data Funcao = Id :->: ([Var], Type) deriving Show

data Programa = Prog [Funcao] [(Id, [Var], Bloco)] [Var] Bloco deriving Show

type Bloco = [Comando]

data Comando = If ExprL Bloco Bloco  
| While ExprL Bloco  
| Atrib Id Expr  
| Leitura Id  
| Imp Expr  
| Ret Expr  
deriving Show