Departamento de Ciência da Computação
Compiladores
Trabalho – 1ª parte – Análise Sintática

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 um 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>
| ε

<Funcao> → <TipoRetorno> **id** (<DeclParametros>) <BlocoPrincipal>

<TipoRetorno> → <Tipo>
| **void**

<DeclParametros> → <Tipo> **id** <Parametros>
| ε

<Parametros> → **,** <DeclParametros>
| ε

<BlocoPrincipal> → **{**<BlocoPrincipal'>**}**

<BlocoPrincipal'> → <Declaracoes> <ListaCmd>

<Declaracoes> → <Tipo> <ListaId>**;** <Declaracoes>
| ε

<Tipo> → **int**
| **string**
| double

<ListaId> → **id** <ListaId'>

<ListaId'> → , <ListaId>
| ε

<Bloco> → **{** <ListaCmd> **}**

<ListaCmd> → <Comando> <ListaCmd>
| ε

<ChamadaFuncao> → **id (**<ListaParametros>**)**

<ListaParametros> → <ListaParametros'>
| ε

<ListaParametros'> → <Expressao> <ListaParametros''>

<ListaParametros''> → **,** <ListaParametros'>
| ε

<Comando> → **return** <TvzExpressao>**;**
| **if (**<ExpressaoLogica>**)** <Bloco> <Senao>
| **while** (<ExpressaoLogica>) <Bloco>
| **id =** <Expressao>**;**
| **print (**<Expressao>**);**
| **read (id);**
| <ChamadaFunção>**;**

<TvzExpressao> → <Expressao>
| ε

<Senao> → **else** <Bloco>
| ε

- 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 term 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 Tipo = TDouble | TInt | TString | TVoid
                                    deriving Show

data TCons = CDouble Double | CInt Int deriving Show

data Expr = Expr :+: Expr | Expr :-: Expr | Expr :*: Expr | Expr :/: Expr |
            Neg Expr | Const TCons | IdVar Id | 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 :#: Tipo deriving Show

data Funcao = Id :->: ([Var], Tipo) 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 (Maybe Expr)
             | Proc Id [Expr]
                   deriving Show
```