

O objetivo final desse trabalho é a implementação de um compilador que gere *bytecodes Java* para a linguagem definida pela gramática abaixo, a saída deve ser um arquivo texto com mnemônicos que representem as instruções. O arquivo gerado deve ser montado pelo *Jasmin*. A linguagem deve manipular apenas três tipos de dados: *int*, *float* e *string*. A sintaxe para expressões (lógicas, relacionais e aritméticas) deve ser definidas.

Na primeira etapa devem ser implementados os **analísadores léxico e sintático** usando o *alex* e *happy*. O analisador sintático deve retornar, como representação intermediária, um tipo algébrico de dados que representa o código fonte por meio de uma árvore sintática abstrata (AST).

Gramática de linguagem fonte:

<Programa>	→ <ListaFuncoes> <BlocoPrincipal> <BlocoPrincipal>
<ListaFuncoes>	→ <ListaFuncoes> <Função> <Funcao>
<Funcao>	→ <TipoRetorno> id (<DeclParametros>) <BlocoPrincipal> <TipoRetorno> id () <BlocoPrincipal>
<TipoRetorno>	→ <Tipo> void
<DeclParametros>	→ <DeclParametros>, <Parametro> <Parametro>
<Parametro>	→ <Tipo> id
<BlocoPrincipal>	→ {<Declaracoes> <ListaCmd>} {<ListaCmd>}
<Declaracoes>	→ <Declaracoes> <Declaracao> <Declaração>
<Declaracao>	→ <Tipo> <Listald>;
<Tipo>	→ int string float
<Listald>	→ <Listald>, id id
<Bloco>	→ { <ListaCmd> }
<ListaCmd>	→ <ListaCmd> <Comando> <Comando>

<Comando>	→ <CmdSe> <CmdEnquanto> <CmdAtrib> <CmdEscrita> <CmdLeitura> <ChamadaProc> <Retorno>
<Retorno>	→ return <ExpressaoAritmetica>; return literal ; return ;
<CmdSe>	→ if (<ExpressaoLogica>) <Bloco> if (<ExpressaoLogica>) <Bloco> else <Bloco>
<CmdEquanto>	→ while (<ExpressaoLogica>) <Bloco>
<CmdAtrib>	→ id = <ExpressaoAritmetica>; id = literal ;
<CmdEscrita>	→ print (<ExpressaoAritmetica>; print (literal) ;
<CmdLeitura>	→ read (id) ;
<ChamadaProc>	→ <ChamaFunção>;
<ChamadaFuncao>	→ id (<ListaParametros>) id ()
<ListaParametros>	→ <ListaParametros>, <ExpressaoAritmetica> <ListaParametros>, literal <ExpressaoAritmetica> literal

- Uma expressão relacional tem como termos expressões aritméticas e envolve um dos operadores: <, >, <=, >=, ==, /=.
- Uma expressão lógica tem como termos expressões relacionais e envolve os seguintes operadores: && (conjunção), || (disjunção), ! (negação). Os operadores binários && e || têm a mesma precedência e a associatividade é da esquerda para a direita, o operador ! é um operador unário e possui a maior precedência.
- 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.

Árvore sintática Abstrata:

type Id = String

data Tipo = TDouble | TInt | TString | TVoid
deriving (Show, Eq)

data TCons = CDouble Double | CInt Int deriving Show

data Expr = Add Expr Expr | Sub Expr Expr | Mul Expr Expr | Div Expr Expr | Neg Expr
| Const TCons | IdVar String | Chamada Id [Expr] | Lit String
| IntDouble Expr | DoubleInt Expr deriving Show

data ExprR = Req Expr Expr | Rdif Expr Expr | Rlt Expr Expr
| Rgt Expr Expr | Rle Expr Expr | Rge Expr Expr deriving Show

data ExprL = And ExprL ExprL | Or ExprL ExprL | Not ExprL | Rel ExprR
deriving Show

data Var = Id :#: (Tipo, Int) 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