Especificação da Fase 3

Bacharelado em Ciência da Computação
Universidade Federal de São Carlos, Campus Sorocaba
Compiladores

1 Terceira Fase

Nesta terceira fase vocês realizarão a geração de código C para a linguagem descrita pela gramática da Seção 3.

2 Entrega

Data: 05/07/19.

Entregue um arquivo compactado .zip com o formato de nome: F01_Nome1_Nome2_Nome3_Nome4.zip, contendo apenas os arquivos de extensão .java e testes realizados, respeitando a seguinte estrutura:

- F03_Nome1_Nome2_Nome3_Nome4/AST: Classes da árvore sintática.
- F03_Nome1_Nome2_Nome3_Nome4/Lexer: Classes do analisador léxico.
- F03_Nome1_Nome2_Nome3_Nome4/Testes: Arquivos de teste (Opcional).
- F03_Nome1_Nome2_Nome3_Nome4/Compiler.java: Arquivo principal e analisador sintático.
- F03_Nome1_Nome2_Nome3_Nome4/Main.java: Arquivo que contém a função void main().

Em nome poder ser NomeSobrenome, para diferenciar alunos com nomes iguais.
Seu Compiler.java deve implementar um método compile(), chamado em Main.java.
Adicione um cabeçalho com nome e RA dos integrantes do grupo em todos os arquivos *.java que forem entregues.
Deve ser possível compilar esta estrutura via linha de comando com javac.

3 Gramática

Na gramática, são terminais:

- Id, um identificador;
- LiteralInt, um número inteiro entre 0 e 2147483647;
- LiteralString, uma sequência de caracteres entre " e ".

Há duas funções pré-definidas, "write" e "writeln", cuja sintaxe é dada pela regra da gramática FuncCall. Elas tomam uma ou mais expressões como argumentos e as imprimem na saída padrão. Apenas expressões dos tipos "Int" e "String" podem ser impressas. "writeln" imprime "\r\n" ao final de todos os argumentos.
Há algumas produções não fatoradas, como `AssignExprStat` (não se sabe se teremos uma atribuição ou apenas uma expressão) e `ExprPrimary` (não se sabe se teremos apenas um Id, uma variável, ou Id seguido de "(", uma chamada de função).

```
Program ::= Func {Func}
Func ::= "function" Id [ "(" ParamList ")" ] ["->" Type ] StatList
ParamList ::= ParamDec {","ParamDec}
ParamDec ::= Id ":" Type
```

```
Type ::= "Int" | "Boolean" | "String"
StatList ::= "{" {Stat} "}"
Stat ::= AssignExprStat | ReturnStat | VarDecStat | IfStat | WhileStat
AssignExprStat ::= Expr [ "=" Expr ] ";"
ReturnStat ::= "return" Expr ";"
VarDecStat ::= "var" Id ":" Type ";"
IfStat ::= "if" Expr StatList [ "else" StatList ]
WhileStat ::= "while" Expr StatList
Expr ::= ExprAnd {"or" ExprAnd}
ExprAnd ::= ExprRel {"and" ExprRel}
ExprRel ::= ExprAdd [ RelOp ExprAdd ]
RelOp ::= "<" | "<=" | ">" | ">=" | "==" | "!="
ExprAdd ::= ExprMult {("+" | "-") ExprMult}
ExprMult ::= ExprUnary {("*" | "/") ExprUnary}
ExprUnary ::= [ ( "+" | "-" ) ] ExprPrimary
ExprPrimary ::= Id | FuncCall | ExprLiteral
ExprLiteral ::= LiteralInt | LiteralBoolean | LiteralString
LiteralBoolean ::= "true" | "false"
FuncCall ::= Id "(" [ Expr {"," Expr} ] ")"
```

4 Exemplo de código

Um programa nesta linguagem, que seu compilador deve aceitar, está exemplificado na figura 6 abaixo:

```
function fatorial(n:Int) -> Int{
  if n <= 0{
    return 1;
  }
  else{
    return n*fatorial(n-1);
  }
}

function imprima(before: Int, valor: Int, after: String){
  var i:Int;

  i = 0;

  while i < before {
    write("*");
    i = i + 1;
  }
  writeln("");
  writeln(valor);
  writeln(after);
}

function main {
  imprima(50, fatorial(5)*2*fatorial(3), "        fim");

  writeln("teste sem erro");
}
```

5 Mensagens de Erro

Nesta fase as mensagens devem ser significativas, principalmente para informar precisamente o erro semântico identificado.

Use o formato o formato a seguir para as mensagens de erro:

\n<nome do arquivo>:<número da linha de erro>:<mensagem de erro>\n<linha do código com erro>

Em <nome do arquivo> não é para conter o caminho do arquivo!

Em <linha do código com erro> pode manter o padrão do compilador, de imprimir o código próximo ao ponto onde o token estava no momento que o erro ocorreu.

O compilador pode continuar lançando exceção java.lang.RuntimeException, mas a mensagem de erro deve estar no formato citado.

6 Geração de código C

Para a geração de código em C, utilize o arquivo Main.java que recebe o arquivo de entrada e o arquivo de saída via linha de comando.

Implemente obrigatoriamente o método genC(), chamado em sua classe principal da AST. Porém algumas classes da AST, mais simples, podem ficar sem o método genC().

Se o código não possuir erros, escreva o arquivo correspondente em linguagem C para o arquivo de saída definido via linha de comando. Caso haja erros, imprima-os na tela no formato pedido na seção 5.

A geração de código C estará correta se o arquivo gerado com extensão .c compilar no compilador gcc. Isso implica que não serão avaliados espaçamentos e indentações em seu código.

Os testes para geração de código C quando compilados e executados produzirão saídas que serão comparadas com diff, ignorando diferenças de espaçamentos, mas é importante deixar ao menos um espaço simples entre qualquer elemento impresso para tela ou arquivo.

O código abaixo é uma possível tradução para o exemplo da seção 4.

```c
#include <stdio.h>

int fatorial(int n){
  if (n <= 0){
    return 1;
  }
  else{
    return n*fatorial(n-1);
  }
}

void imprima(int before, int valor, char after[]){
  int i;
  i = 0;
  while (i < before){
    printf("*");
    i = i + 1;
  }
  printf("\r\n");
  printf("%d \r\n", valor);
  printf("%s \r\n", after);
}
void main(){
  imprima(50, fatorial(5)*2*fatorial(3), "        fim");
}
```