Compilador 2 – Documentação Externa

Fase 2

Compiladores
Prof. Dra. Tiemi C. Sakata

Alunos:

Bruno Villas Bôas da Costa	**317527**
Renato Jensen Filho	**317136**

Sumário

Introdução

Este documento trata-se das funcionalidades e características do compilador em questão. Aqui, iremos definir e descrever a gramática de nossa linguagem, explicar o funcionamento do compilador, apresentar casos de testes e analisar os resultados.

Gramática

Houve alterações entre as gramáticas da primeira para a segunda fase do projeto. Para esta fase, precisamos adicionar procedimentos e funções, tratamento correto de variáveis locais e globais e apresentar os vários erros que podem existir no código-fonte.

A gramática aplicada à primeira fase deste projeto está definida abaixo:

```
Program ::= [VarDecList] CompositeStatement
CompositeStatement ::= "inicio" StatementList "fim"
StatementList ::= | Statement ";" StatementList
Statement ::= AssignmentStatement | IfStatement | ReadStatement | WriteStatement | WhileStatement
AssignmentStatement ::= Variable "<-" OrExpr
IfStatement ::= "se" OrExpr "entao" StatementList ["senao" StatementList] "fimse;"
ReadStatement ::= "leia" '(' Variable ')'
WriteStatement ::= "escreva" '(' OrExpr ')'
WhileStatement ::= "enquanto" OrExpr "faca" StatementList "fimenquanto;"
VarDecList ::= VarDecList2 {VarDecList2}
VarDecList2 ::= Type Variable {',' Variable} ';'
Variable ::= Letter {Letter}
Type ::= "inteiro" | "carac" | "boleano"
OrExpr ::= AndExpr [ "||" AndExpr]
AndExpr ::= RelExpr [ "&&" RelExpr]
RelExpr ::= AddExpr [ RelOp AddExpr]
AddExpr ::= MultExpr {AddOp MultExpr}
MultExpr ::= SimprExpr {MultOp SimpleExpr}
SimpleExpr ::= Number | Character | '(' OrExpr ')' | "!" SimpleExpr | Variable | '(' OrExpr ')' | "verdadeiro" | "falso"
RelOp ::= '<' | '<=' | '>' | '>=' | '==' | '!='
AddOp ::= '+' | '-'
Number ::= ['+'|'-'] Digit {Digit}
Digit ::= '0' | '1' | ... | '9'
Letter ::= 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'
```

Um exemplo de código desta linguagem é:

```
Inteiro i;
boleano b;
carac c;
inicio
     read(i);
     c <- ‘a’;
     if (b == verdadeiro) então
          i <- i + 5;
     senão
          enquanto (b == falso) faca
               i <- i + 500;
               b <- verdadeiro;
          fimenquanto;
     fimse;
fim
```

A gramática aplicada à segunda fase está apresentada a seguir:

```
Program ::= ProcFunc { ProcFunc }
ProcFunc ::= Procedure | Function
Procedure ::= "procedimento" Ident '(' ParamList ')'
[VarDecList] CompositeStatement "fimprocedimento;"
Function ::= "funcao" Type Ident '(' ParamList ')' [
VarDecList ] CompositeStatement "fimfuncao;"
ParamList ::= | ParamDec {';' ParamDec}
ParamDec ::= Type Ident
CompositeStatement ::= "inicio" StatementList "fim"
StatementList ::= | Statement ";" StatementList
Statement ::= AssignmentStatement | IfStatement |
ReadStatement | WriteStatement | WhileStatement |
ProcedureCall | ReturnStatement
AssignmentStatement ::= Variable "<-" OrExpr
IfStatement ::= "se" OrExpr "entao" StatementList ["senao"
StatementList] "fimse;"
ReadStatement ::= "leia" '(' Variable ')'
WriteStatement ::= "escreva" '(' OrExpr ')'
WhileStatement ::= "enquanto" OrExpr "faca" StatementList
"fimenquanto;"
ProcedureCall ::=  Ident '(' ExprList ')'
ExprList ::= | OrExpr {';' OrExpr}
ReturnStatement ::= "retorna" OrExpr
VarDecList ::= VarDecList2 {VarDecList2}
VarDecList2 ::= Type Variable {',' Variable} ';'
Variable ::= Letter {Letter}
Type ::= "inteiro" | "carac" | "boleano"
OrExpr ::= AndExpr [ "||" AndExpr]
AndExpr ::= RelExpr [ "&&" RelExpr]
RelExpr ::= AddExpr [ RelOp AddExpr]
AddExpr ::= MultExpr {AddOp MultExpr}
MultExpr ::= SimprExpr {MultOp SimpleExpr}
```

```
SimpleExpr ::= Number | Character | '(' OrExpr ')' | "!"
SimpleExpr | Variable | '(' OrExpr ')' | "verdadeiro" |
"falso"
RelOp ::= '<' | '<=' | '>' | '>=' | '==' | '!='
AddOp ::= '+' | '-'
Number ::= ['+'|'-'] Digit {Digit}
Digit ::= '0' | '1' | ... | '9'
Letter ::= 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'
```

Um exemplo de código desta linguagem é:

```
procedimento principal()
   inteiro i ,j;
   boleano d,e,f;
   inicio
   leia(i);
   se(i > 10)
   entao
     escreva(i);
   senao
     escreva('<');
   fimse;
   enquanto (i < 10) faca
     escreva('<');
     i <- i+1;
   fimenquanto;
fim
```

Descrição do Programa

O programa possui a funcionalidade de compilar um código da nossa linguagem especificada para a linguagem C, realizando análises sintática e léxica.

Nosso compilador foi feito baseado no ‘compilador 9’ desta matéria e, modificado para a segunda fase, baseado em algumas funções do ‘compilador 10’. Portanto suas classes estão divididas da seguinte forma: (1) um pacote padrão contendo as classes ‘`compiler.java`’ e ‘`main.java`’; (2) um pacote AST contendo as classes necessárias para a analise sintática do código; (3) um pacote Lexer contendo as classes ‘`lexer.java`’ e ‘`Symbol.java`’, responsáveis pela análise léxica e definição da gramática utilizada; e (4) um pacote AuxComp contendo as classes ‘`CompilerError.java`’, ‘`StatementException.java`’ e ‘`SymbolTable.java`’, responsáveis pelo tratamento de erro sem finalizar a execução do programa.

Para o programa ser executado, deve ser especificado um arquivo de entrada contendo código da linguagem definida para este compilador e outro

arquivo de saída (.c) onde o programa irá inserir o código em linguagem C. Estes arquivos devem ser passados por argumentos ao executar o programa. Este é executado da seguinte forma:

```
Java –jar "nomeDoPrograma.jar" entrada.i saída.c
```

Descrição dos Algoritmos Utilizados

O algoritmo criado para a resolução deste problema é baseado sumariamente na análise do arquivo de entrada linha-a-linha. Cada linha é analisada sintaticamente e lexicamente de acordo com a gramática apresentada e deve seguir uma ordem especificada pela linguagem.

As linhas são analisadas palavra por palavra através de um '`token`' que identifica o que a palavra poderia representar na linguagem, por exemplo, um comando, uma variável, uma função, palavras reservadas, entre outros símbolos (definidas na classe '`Symbol.java`').

Conforme o arquivo de entrada está sendo analisado, o código em linguagem C vai sendo gerado, através da função abstrata '`genC()`' da classe abstrata '`Expr.java`'. O código gerado é armazenado para ser impresso na ordem correta ao final da execução do programa no arquivo de saída. Este procedimento é realizado pela classe '`PW.java`'.

A partir da segunda fase, o tratamento de erros passou a analisar o código completo sem a interrupção da execução ao encontrar o primeiro erro, como era feito anteriormente. Esta função é desempenhada pelo pacote AuxComp do compilador.

Condições de Contorno

Para o programa funcionar corretamente, deve ser especificado um arquivo de entrada contendo um código válido da linguagem definida pela gramática proposta neste documento.

Caso haja algum comando que não seja identificado pela gramática, ou não siga a ordem correta da gramática, o programa irá detectar a linha onde o erro foi encontrado e imprimirá uma mensagem de erro identificando-o no arquivo de saída.

No nosso compilador, temos três tipos de variáveis (inteiro, carac e boleano), porém, na linguagem C, não há um equivalente para o tipo 'boleano'. Para sanar este problema, criamos um novo tipo de variável em C através das funções '`typedef`' e '`enum`'. Assim, tornou-se possível utilizar variáveis booleanas na linguagem C.

Se o usuário quiser inserir comentários na linguagem, basta acrescentar os caracteres ‘- -’ seguidos do comentário. Desta forma, a linha não será interpretada pelo compilador.

Ao chamar uma função, ou procedimento, os tipos e a quantidade dos parâmetros passados devem ser idênticos aos definidos na declaração dos mesmos. O procedimento ‘principal’ não pode conter parâmetros.

Descrição dos Testes Realizados

Foi realizada uma série de testes utilizando inúmeras variações de comandos na linguagem proposta, mesclados com comandos errados propositalmente para analisar a resposta do programa. Estes testes formam um total de 12 (doze) como solicitado na especificação do trabalho.

Os testes realizados com erros foram identificados corretamente e impressos no arquivo de saída. Nos testes realizados com os códigos corretos, o compilador se comportou da forma esperada, imprimindo o código equivalente em C no arquivo de saída.

Em anexo ao compilador, estamos enviando arquivos para testes, na extensão ‘`.i`’ e seu equivalente em linguagem C (‘`.c`’). Estes arquivos estão na pasta ‘`dist`’, junto ao arquivo ‘`.jar`’.

Dificuldades Encontradas

A maior dificuldade encontrada foi entender o uso do compilador devido à alta complexidade do mesmo.