Relatório - Compiladores

Luan Bodner do Rosário

27 de Abril de 2016

Introdução

Este documento tem como objetivo especificar o processo de implementação da primeira parte de um compilador(análise léxica e análise sintática) para a disciplina de Compiladores. Para a implementação do programa, foi utilizada a linguagem de programação C++ e a biblioteca *boost* para auxíliar na varredura do arquivo.
As saídas e entradas desse programa estão armazenadas em arquivos dados prévia à execução do compilador. O restante deste documento está organizado da seguinte forma:

- Análise Léxica : Funcionamento da varredura do arquivo e descrição das bibliotecas utilizadas e criadas para a captura das marcas do arquivo T++.
- Análise Sintática: Descrição do método utilizado para criação da AST ,análise da estrutura dos *tokens* e métodos de tratamento de erros.
- Gramática : Definição da gramática BNF utilizada para a criação da análise sintática e descrição da classe de tratamento de erros do compilador.

Análise Léxica

Para o processo de varredura, não foi utilizada nenhuma ferramenta auxiliar para a construção da lista de *tokens*. Para que as marcas fossem reconhecidas no arquivo, foi utilizada a biblioteca *boost/regex*.

Os tipos de marcas foram armazenadas em um *enum* na biblioteca *Token.h*. Essa lista de marcas são definidas dessa forma:

```
typedef enum {

    IF, THEN, OTHERWISE, END, REPEAT,
    FLOAT, VOID, UNTIL, READ, WRITE,
    INTEGER, COMMENTS, RETURN, SUM, SUBTRACTION,
```

```
        MULTIPLICATION, DIVISION, EQUAL, COMMA, ATTRIBUTION,
        LESS_THAN, BIGGER_THAN, LESS_EQUAL, BIGGER_EQUAL, OPEN,
        CLOSE, NUMBER_INTEGER, NUMBER_FLOAT, DOUBLE_POINT, IDENTIFIER
    } TokenType;
```

Quando dado *token* é analisado, o seu conteúdo é armazenado em um vetor de estruturas *tokens*

Os *tokens* são analisados de acordo com expressões regulares. As mais importantes listadas a seguir:

- ```
  ^\\b[A-Za-z_][A-Za-z_0-9_]*
  ```

```
    static std::vector<Token::Token> tokens;
```

A classe *Token.h* é dada pela seguinte estrutura:

```
    std::string tokenName;
    TokenType token;
    int tokenColumn;
    int tokenLine;
```

O arquivo em que o código fonte é armazenado é lido linha por linha até que todos os caracteres sejam analisados e aceitos pela varredura, caso algum erro aconteça, a execução é finalizada. A varredura do arquivo é feita pela classe *LexicalAnalyzer*.

Análise Sintática

A análise sintática deste projeto foi feita utilizando algoritmos *top-down* de análise recursiva descendente sem auxílio de ferramentas para a verificação da estrutura do programa.
A gramática BNF foi transformada para se adequar à analise descendente (retirada da recursão à esquerda).
Utilizando o algoritmo de descida recursiva, a gramática e a estrutura do programa ficaram bastante semelhantes, já que cada não terminal da gramática se tornou um método dentro do programa que avançava na lista de *tokens* conforme a leitura avançava.

A biblioteca responsável pela criação da árvore e análise é chamada de *SyntaxAnalyzer.h* e conta com apenas dois atributos:

```
    Lex::LexicalAnalyzer lexer;
    Tree::Tree syntaxTree;
```

A análise léxica está portanto diretamente ligada a análise sintática e não exitirá sem uma instância da análise sintática, assim como a classe *Tree.h*, que é definida pela seguinte estrutura:

```
    std::string exp;
    Token::Token token;
    int active;
    std::vector<Tree*> children;
```
```

Conforme os *tokens* são analisados e se encaixam na gramática, eles também são inseridos na árvore de acordo com a sua colocação dentro do programa e sua necessidade. Desta forma, montamos uma árvore abstrata onde apenas as folhas representam valores diretamente do programa.

Os erros gerados durante a execução do programa foram especificados na classe *CompilerErrors.h* (tanto erros léxicos quanto erros sintáticos), onde cada método mostra uma mensagem adequada quanto a coluna e a linha do programa onde determinado erro aconteceu. Como por exemplo:

```
void CompilerErrors::unidentifiedTokenError(int Token, Token::Token token) {

        std::cout << "Unidentified Token Error; "
                                << "Received : "
                << intToString(Token)
                << "; Expected : "
                << token.tokenTypeToString()
                << "; Name : " << token.getTokenName();

        exit(EXIT_FAILURE);
}
```

Portanto, a cada erro encontrado, o programa mostra a mensagem de erro e é finalizado, o que gera o problema de não conseguir mostrar mais de um erro por compilação.

Gramática

<type> ::= 'vazio'
| 'inteiro'
| 'flutuante'

<variableDec> ::= <type> ':' 'id'

<operationsExp> ::= <equalityExp>

<equalityExp> ::= <relationalExp> <equalityExpTransformed>

<equalityExpTransformed> ::= NULL
| '=' <relationalExp> <equalityExpTransformed>

<relationalExp> ::= <additiveExp> <relationalExpTransformed>

<relationalExpTransformed> ::= NULL
| < <additiveExp> <relationalExpTransformed>
| > <additiveExp> <relationalExpTransformed>
| <= <additiveExp> <relationalExpTransformed>

| >= <additiveExp> <relationalExpTransformed>

<additiveExp> ::= <multiplicativeExp> <additiveExpTransformed>

<additiveExpTransformed> ::= NULL
| + <multiplicativeExp> <additiveExpTransformed>
| - <multiplicativeExp> <additiveExpTransformed>

<multiplicativeExp> ::= <factor> <multiplicationExpTransformed>

<multiplicationExpTransformed> ::= NULL
| * <factor> <multiplicationExpTransformed>
| / <factor> <multiplicationExpTransformed>

<factor> ::= '(' <operationsExp> ')'
| 'numberFloat'
| 'numberInt'
| 'id'

<prototypeDef> ::= '(' <paramFunction> ')'

<paramFunction> ::= NULL
| <variableDec> ',' <paramFunction>
| <variableDec>

<functionDec> ::= <type> 'id' <prototypeDef> <compoundStmt> 'fim'

<prototypeCall> ::= NULL
| <operationsExp> ',' <protypeCall>
| <operationsExp>

<functionCall> ::= 'id' '(' <paramCall> ')'

<iterationExp> ::= 'repita' <compoundStmt> 'até' <operationsExp> 'fim'

<selectionExp> ::= 'se' <operationsExp> 'então' <compoundStmt> 'fim'
| 'se' <operationsExp> 'então' <compoundStmt> 'senão'
<compoundStmt> 'fim'

<ioTypes> ::= 'id'
| 'numberInt'
| 'numberFloat'

| <functionCall>

<attributionExp> ::= 'id' ':=' <operationsExp>

<returnCom> ::= 'retorna' '(' <operationsExp> ')'

<readCom> ::= 'leia' '(' 'id' ')'

<writeCom> ::= 'escreve' '(' <operationsExp> ')'

<compoundStmt> ::= <expression> <compoundStmt>
| <expression>

<expression> ::= <selectionExp>
| <iterationExp>
| <functionCall>
| <readCom>
| <writeCom>
| <returnCom>
| <variableDec>
| <attributionExp>