Relatório - Compiladores

Luan Bodner do Rosário

28 de Junho 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:

```
        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>

Mudança na Linguagem

Foram feitas algumas modificações na sintátixe da linguagem, como a remoção de múltiplos retornos(é possível criar apenas um retorno por função) e as expressões numéricas e chamadas de funções dentro de expressões deve estrar entre parenteses, pois isso facilita a analise da expressão. Outra mudança feita, devido à problemas durante a converção de valores durante a geração de código, foi deixar a tipagem forte, ou seja, valores de ponto flutuante só podem estar em variáveis flutuantes, caso contrário um erro é mostrado.

Análise Semântica

Para fazer a análise Semântica da linguagem, sendo essa a "terceira"passada pelo código foi criada a classe *SemanticAnalysis*. Essa classe por sua vez passa pela AST e gera uma tabela de símbolos, como exemplificada a seguir:

```
0, principal / vazio / Function / 0
1, ret / inteiro / Used / Initialized
1, f / inteiro / Used / Initialized
0, fatorial / vazio / Function / 1 / inteiro
0, n / inteiro / Global / Unused / Not Initialized
```

Essa tabela é representada no programa como uma *hashtable*, cuja estrutura é definida da seguinte maneira:

```
typedef std::pair<int, std::string> ScopeName;
typedef std::vector<std::string> vectorString;

boost::unordered_map<ScopeName, vectorString> symbolTable;
```

As colunas armazenadas nessa tabela são:

- Escopo(chave)
- Nome(chave)
- Tipo

A partir daqui, a tabela se diferencia entre funções e variaveis, sendo que para funções, as próximas colunas são:

- *Flag* definindo se é uma função
- Número de parâmetros
- Tipo de cada um dos parâmetros

Para variáveis:

- *Flag* definindo se é global
- *Flag* definindo se foi ou não utilizada
- *Flag* definindo se foi ou não inicializada

Não foi utilizada nenhuma técnica em especial para geração dessa tabela, sendo que ela foi preenchida conforme os dados foram descobertos na AST.

Além disso, a adição da análise semântica levou à modificação da classe de Erros, que foi atualizada agora para gerar os *warnings* e erros da anpalise semântica, o cabeçalho das novas funções adicionadas são:

```
void declarationScopeError(Token::Token);
void variableNotDeclaredError(Token::Token);
void functionCallError(Token::Token);
void functionCallScopeError(Token::Token);
void variableNotDefinedError(Token::Token);
void variableVoidError(Token::Token);
void expressionTypeError(Token::Token);
void returnIgnoredWarning(Token::Token);
void voidAttributionError(Token::Token);
void functionWithoutReturnWarning(std::string);
void returnMayBeIgnoredWarning(Token::Token);
void variableNotUsedWarning(std::string, std::string);
void mainDeclarationError();
void tooManyReturnsError(Token::Token);
```

Além dessas adições, o esquema de mensagem de erro e *warnings* também foi modificada de forma que elas sejam visualmente mais simples de perceber e produzam mais informações para o programador. Entre as melhorias citadas estão:

- Mensagens com cores
- printar os *tokens* onde o erro se encontra

A estrutura da AST não foi modificada, e a geração de código é feita em cima da tabela de simbolos gerada e da própria árvore.

Geração de Código

Para a geração de código, inicialmente foi utilizada a classe *CodeGeneration*, que foi refatorada em uma nova classe *llvmCodeGeneration*, devido aos diversos erros gerados. A classe inicial então foi reutilizada quando adequada.

Como váriaveis da classe, foram armazenadas as referencias dos módulos responsáveis pela adição de instruções e contexto dos módulos.

```
/* Current Builder */
llvm::IRBuilder<> * builder;

/* tiny Module */
llvm::Module * module;

/* Current function */
llvm::Function * function;

/* Current basic block */
llvm::BasicBlock * block;
```

Conforme as variáveis são definidas ou declaradas, sejam elas globais ou locais, elas são armazenadas em duas *hashtables* separadas, para que a origem da variável seja distinguível.

```
typedef boost::unordered_map<ScopeName, vectorString> SymbolTable;

/* Alloca Instruction table */
boost::unordered_map<ScopeName, llvm::AllocaInst*> variablesHash;

/* Function Parameters table */
boost::unordered_map<ScopeName, llvm::AllocaInst*> paramHash;
```

A geração de código não foi concluída, portanto não foi possível aplicar nenhuma forma de otimização de código. Foram concluidas apenas as seguintes operações na linguagem :

- Atribuição
- Declaração de globais
- Declaração de funções
- Chamada de funções
- Alocação de variáveis locais
- Operações de soma, subtração, multiplicação e divisão com valores inteiros ou flutuantes
- Retorno nas funções

Execução

Para executar o compilador, deve-se executar o seguinte comando na linha de comando após a compilação:

```
./compilador <nome da entrada> <lista de tokens> <AST> <tabela de simbolos>
```