Tradutor De C Modificado

Artur Henrique Brandão de Souza - 15/0118783

Universidade de Brasília
150118783@aluno.unb.br

1 Motivação

Para uma formação completa para a área de Bacharelado de Ciência da Computação, o entendimento de como funciona a tradução de um programa é essencial. Assim, a disciplina Tradutores, ministrada na Universidade de Brasília, proporciona os estudantes a entenderem como funciona todo o processo de compilação de um programa. Para esse processo, foi divido pela docente em 4 partes do trabalho prático total, em que essa divisão se resume a análise léxica, análise sintática, análise semântica e por fim um código intermediário para haver a execução de programas. Com isso, este trabalho demonstra uma versão simplificada da analise léxica para a linguagem C. No entanto, há uma variação quanto a linguagem que se trata da introdução de operações para conjuntos, em que há dois novos tipos *elem* e *set* e operações referentes a esses novos tipos.

2 Desenvolvimento

2.1 Analisador Léxico

Como primeira parte do trabalho, a analise léxica foi desenvolvida através da biblioteca flex [1] e com o auxílio dos livros [2], [3] e [4] em busca de verificar se uma cadeia de caracteres, também chamado de *tokens*, pertencem a forma léxica descrita na linguagem relatada na Seção 5 em que se encontra a gramática da linguagem.

Para a identificação da posição de leitura da linha e posicionamento da coluna das palavras verificadas pelo código léxico, os *tokens*, foi utilizado dois contadores para todo o processo. Assim, para cada *token* encontrado, é acrescentado o contador até chegar no *token* $\backslash n$ que significa que a linha acabou e é verificado a linha abaixo zerando o contador das colunas até chegar ao fim do arquivo.

Por fim, os *tokens* que forem coletados pelo analisador léxico, os de maior relevância para os demais trabalhos a serem desenvolvidos serão os referentes ao ID(identificador). Para que esse identificador seja analisado, será desenvolvido uma tabela de símbolos em que será verificado se este identificador já existe na tabela de símbolos e se pertence a uma função ou variável. Assim, será acrescentado na tabela todos os novos casos que forem aparecendo conforme o código é percorrido. E, além de armazenar esses identificadores, será necessário observar em qual nível de bloco que eles se encontram, visto que pode haver identificadores similares, porém em funções distintas.

Para tratamento de erro no léxico foi desenvolvido uma forma que, em caso do léxico não encontrar um *token* que não esteja identificado na linguagem, ele será identificado pela expressão regular "." e, posteriormente, será mostrado uma mensagem de erro identificando a posição de linha e coluna do *token* não reconhecido.

2.2 Análise Sintática

Na segunda parte do trabalho, foi desenvolvido a parte sintática utilizando o *Bison* [5]. O analisador sintático utiliza os *tokens* identificados pelo léxico para criar uma árvore baseada na estrutura da gramática da Seção 5.

Para a implementação, a "main" do código foi passado para o arquivo "sintatico.y" para que esse pudesse chamar tanto a parte léxica quanto o parse com *yyparse()*. Após a compilação do arquivo sintático, é gerado pelo *Bison* um arquivo chamado "sintatico.tab.h" que será importado pelo arquivo léxico. Esse novo arquivo é o que fará a ligação entre os dois arquivos(sintático e léxico) junto com todos os campos declarados pela diretiva *%tokens* no analisador sintático. Após garantir essa conexão, foi desenvolvido uma construção de regras de transição da gramática para o analisador sintático. Assim, as variáveis da gramática são desdobradas até chegarem a seus respectivos terminais *tokens*.

Com as regras de transições construidas, em seguida vem a implementação da árvore sintática, em que, até então, é a única estrutura de dados desenvolvida, tendo em vista que será nela que a gramática será percorrida e seus nós sendo os *tokens*. Assim, a chamada da árvore é *bottom-up*, ou seja, ocorre uma chamada recursiva do parser ao chamar as regras. Devido a isso, para cada nó contido na *struct*, ele saberá quem são os nós que estão à sua esquerda e direita. A ideia é que o *Bison* reconheça os nós como string e, com isso, conseguirá distinguir qual será *token* ou variável da gramática.

Para tratamento de erro no sintático, É mostrado qual regra da gramática deveria ter aparecido ao invés do que foi passado no código exemplo.

3 Arquivos Testes

3.1 Analisador Léxico

Para os arquivos *analisadorLexico_certo1.txt* e *analisadorLexico_certo2.txt* contidos no arquivo enviado junto a esse documento, há os casos em que os testes estão corretos. Foi pensado em uma forma de conter a maior parte da nova variação da linguagem.

Por outro lado, para os dois programas abaixo(*analisadorLexico_errado1.txt* e *analisadorLexico_errado2.txt*), ambos contêm erros que podem ser detectados pela analise léxica, em que no primeiro há um abre aspas simples sem fechar na linha 4.

```
1 int main() {
2     set zero; set one; set two; set three;
```

```
3
4       zero = 'EMPTY;
5       one = succ(zero);
6       two = succ(one);
7       three = succ(two);
8   }
```

Para o segundo programa, há dois erros referentes a um caractere especial que não foi descrito na linguagem, o que faz com que as linhas em que se encontram erros e suas posições exatas sejam mostradas na saída da execução do programa. Além disso, demonstra que o analisador desenvolvido consegue capturar mais de um erro sem que o analisador pare após encontrar o primeiro erro.

```
1 set copy_set(set s) {
2     set %ans;
3     ans = EMPTY;
4     forall(x in$ s) add(x in ans);
5     return ans;
6 }
```

3.2 Analisador Sintático

Para os arquivos *analisadorSintatico_certo1.c* e *analisadorSintatico_certo2.c* contidos no arquivo enviado junto a esse documento, há os casos em que os testes estão corretos. Foi pensado em uma forma de verificar até onde consegui desenvolver na parte da gramática do código ao qual desenvolvi, chegando até na parte de reconhecer a criação de uma função e, internamente, colocar apenas declaração de variáveis.

Por outro lado, para os códigos que contém erro, esses erros são referentes a gramática desenvolvida, quando a ordem dos *tokens* não estão de acordo com a ordem das regras de transições.

Para o primeiro caso errado, há um acréscimo de um tipo no momento da declaração da variável, o que não corresponde com a sintaxe da linguagem.

```
1 int main(){
2     int set oi;
3 }
```

Já para o segundo caso errado, foi declarado uma função, porém o parametro passado não está completo, há apenas o tipo sem o identificador. O que diverge da linguagem descrita gerando erro ao final.

```
1 int main(float){
2
3 }
```

4 Execução código

Para a compilação do código, será necessário abrir a raiz do arquivo em anexo enviado e executar os seguintes comandos:

```
#Caso nao possua o comando make instalado, utilizar o comando:
apt-get install build-essential
```

```
make
./programa.out <arquivo_a_ser_testado>
```

5 Gramática

Para a gramática descrita abaixo, as variáveis estão estão representadas entre '<' '>' e os terminais estão sem estes símbolos.

```
<tradutor> ::= <declaracoesExtenas>

<declaracoesExtenas> ::= <funcoes>
                       | <declaracoesVariaveis>
                       | <funcoes> <declaracoesExtenas>
                       | <declaracoesVariaveis> <declaracoesExtenas>

<funcoes> ::= <tipagem> <identificador> ( <parametros> ) <posDeclaracao>

<parametros>: parametroLista | e

<parametroLista> ::= <declaracoesVariaveis>
                   | <parametros> , <declaracoesVariaveis>

<declaracoesVariaveis> ::= <tipagem> <identificador> ; | <tipagem> <identificador>

<posDeclaracao> ::= { <declaracoesVariaveis> <conjuntoSetencas> }

<conjuntoSetencas> ::= <sentenca> <conjuntoSetencas> | e

<sentenca> ::= <condicionalSentenca>
            | <iteracaoSentenca>
            | <returnSentenca>
            | <leituraEscritaSentenca>
            | <expressoesSenteca>
            | <conjSetenca>


<condicionalSentenca> ::= if ( <expressao> ) <sentenca>
                        | if ( <expressao> ) <sentenca> else <sentenca>


<expressao> ::= <identificador> = <expressao>
               | <expressaoSimples>
               | <expressaoConjunto>


<operacoesConjunto> ::= add ( <elemento>)
                  | remove ( <elemento>)
                  | exists ( <elemento> in <elemento> )


<iteracaoSentenca> ::= for ( {<expressao>}? ; {<expressao>}? ; {<expressao>}? ) <setenca>

<returnSentenca> ::= return {<expressao>}? ;

<leituraEscritaSentenca> ::= write (<letra>)
                           | writeln (<letra>)
                           | read (<identificador>)
```

```
<expressoesSenteca> ::=  <expressao> ;
                        | ;

<conjSetenca> ::= forall ( <expressaoIn> ) <setenca>
                 | is_set ( <identificador> )

<expressaoIn> ::= <expressao> " in " <expressaoConjunto>
                 | <expressao> " in " <identificador>

<expressaoConjunto> ::= { <grupoElemento> }
                   | <operacoesConjunto>

<grupoElemento> ::= <elemento> , <grupoElemento>
                  | <elemento>

<elemento> ::= <identificador>
         | <inteiro>
         | <decimal>

<expressaoSimples> ::= <expressaoOperacao> <operacaoComparacao> <expressaoOperacao>
                        | <expressaoOperacao>


<expressaoOperacao> ::= <expressaoOperacao> <operacaoNumerica> <termo>
                        | <expressaoOperacao> <operacaoLogic> <termo>
                        | <termo>

<operacaoComparacao> ::=  ==
                        |!=
                        |>
                        |<
                        |>=
                        |<=

<operacaoNumerica> ::=  +
                        | -
                        | /
                        | *

<operacaoLogic> ::= ||
                    | &&
                    | !

<termo> ::= (<expressaoSimples>)
    |<identificador>
    |<digito>
    |<digito>.<digito>
    | EMPTY


<identificador> ::= (<sublinhado>|<letra>)+(<letra>|<digito>|<sublinhado>)*

<letra> ::= [a-zA-Z] <letra>
```

```
            | e

<digito> ::= [0-9] <digito>
            | e

<sublinhado>  ::= _
```

References

1. Manual Flex, `https://westes.github.io/flex/manual/`, Last accessed 16 Fev 2021.
2. Levine, J.: Flex Bison. 2nd edn. O'Reilly, California (2009)
3. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, Tools. 2nd edn. Pearson, Boston (2007)
4. The C programming language, 2nd edition, by Brian W. Kernighan and Dennis M. Ritchie,Prentice Hall, 1988.
5. GNU Bison, `https://www.gnu.org/software/bison`, Last accessed 17 Mar 2021.