Analisador Lexico

Trabalho de Tradutores - parte 2

João Gabriel Lima Neves - 15/0131992

Profª. Cláudia Nalon

Setembro de 2020

1 Introdução

Entender o processo de tradução de um programa é essencial para a formação de um cientista da computação. O curso de Tradutores ministrado na Universidade de Brasília serve justamente a providenciar aos estudantes do curso de Ciência da Computação o conjunto de habilidades para compreender o processo de compilação de um programa. Desta forma, o trabalho do curso consistirá em 6 etapas aonde será desenvolvido de um tradutor sendo estas: Escolha do Tema, Analisador Léxico, Analisador Sintático, Analisador Semântico, Gerador de Código Intermediário e Apresentação do Trabalho. Este relatório tratara da primeira etapa.

2 Motivação

Atualmente, a física computacional possui atuação muito mais abrangente que a tradicional. O conhecimento em Ciência e Engenharia da Computação é utilizado como ferramenta para os avanços tanto em física teórica como experimental, ao mesmo tempo em que conceitos da Física são aplicados à Teoria da computação [2]. Ao mesmo tempo, o mercado de jogos eletrônicos teve um crescimento bastante significativo nas ultimas décadas sendo que em 1995 contávamos com 100 MM de jogadores (Gamers, do inglês) e passamos para 2.6 bilhões de jogadores ativos em torno do globo [1]. Uma necessidade comum nessas duas áreas é a de representar forças, velocidades e posições de objetos em um plano. Dessa forma é interessante para essas duas áreas a existência de uma estrutura de dados dentro de uma linguagem que permita representar vetores.

Linguagens como o `C#`[1] providenciam ao programador a capacidade de poder abstrair varias operações comuns relacionadas a manipulação de vetores como soma, subtração, normalização e distancia entre dois pontos por exemplo, de tal forma ela é a linguagem

[1] https://docs.microsoft.com/pt-br/dotnet/csharp/

usado pelo Unity, uma Game Engine bastante utilizada no mercado para o desenvolvimento de jogos eletrônicos. A linguagem `C` por outro lado, não oferece esse tipo de abstração.

3 Objetivo do Projeto

Este trabalho visa garantir operações básicas entre pares ordenados para facilitar e optimizar operações comuns relacionadas a manipulação de vetores. Dessa forma ele se propõem a implementar um tradutor para uma versão simplificada da linguagem `C` que suporte comandos de leitura e escrita e chamadas de sub-rotinas, operações de controle condicional de fluxo e laços de repetição, operações com inteiros e números de ponto flutuante e, como uma adição ao `C`, pares ordenados(Vectors) com operações de adição e subtração entre vetores físicos, multiplicação de entre vetores, multiplicação entre inteiros ou reais e vetores, calcular a distancia entre dois vetores e normalização de um vetor. Um exemplo do que seria um código dessa linguagem pode ser visto abaixo:

```
1  Vector2 vect1;
2  Vector2 vect2;
3  Vector3 vect3;
4
5  vect1 = <1.0 , 2.7>;
6  vect2 = <3.5 , 4.0>;
7  vect3 = vect1 + vect2;
8
9  write(vect3[0])
10 write(vect3[1])
11 write(vect3)
```

```
>> 4.5
>> 6.7
>> <4.5 , 6.7>
```

4 Gramatica

A forma utilizada no curso, e que será utilizada neste trabalho, de especificar uma sintaxe para uma linguagem é a de construir uma gramática livre de contexto. A gramática da linguagem que será desenvolvida neste projeto pode ser encontrada abaixo:

1. $prog \rightarrow declarationList$
2. $declarationList \rightarrow declarationList\ declaration \mid declaration$
3. $declaration \rightarrow variableDeclaration \mid functionDeclaration$
4. $variableDeclaration \rightarrow type\ \mathbf{ID\ ;} \mid type\ \mathbf{ID\ [\ INT\ ]\ ;} \mid type\ \mathbf{ID < INT\ ,\ INT >\ ;}$
5. $variableDeclaration \rightarrow type\ \mathbf{ID < FLOAT\ ,\ FLOAT >\ ;}$

6. $type \rightarrow$ **int** | **float** | **void** | **bool** | $Vector2$
7. $functionDeclaration \rightarrow type$ **ID** **(** $params$ **)** $compoundStmt$
8. $params \rightarrow paramList$ | **void**
9. $paramList \rightarrow paramList$ **,** $param$ | $param$
10. $param \rightarrow type$ **ID** | $type$ **ID** **[** **]**
11. $compoundStmt \rightarrow$ **{** $localDeclarations\ stmtList$ **}**
12. $localDeclarations \rightarrow localDeclarations\ variableDeclaration \mid \varepsilon$
13. $stmtList \rightarrow stmtList\ stmt \mid \varepsilon$
14. $stmt \rightarrow expressionStmt \mid conditionalStmt \mid iterationStmt \mid returnStmt \mid IOStmt \mid vectorStmt$
15. $expressionStmt \rightarrow expression$ **;**
16. $conditionalStmt \rightarrow$ **if** **(** $expression$ **)** $compoundStmt$ |
    if **(** $expression$ **)** $compoundStmt$ **else** $compoundStmt$
17. $iterationStmt \rightarrow$ **while** **(** $expression$ **)** $compoundStmt$
18. $returnStmt \rightarrow$ **return** $expression$ **;** | **return** **;**
19. $IOStmt \rightarrow\ read(var)$**;** | $write(var)$ **;** | $write($**STRING**$)$ **;**
20. $vectorStmt \rightarrow\ normalize(Vector2)$**;** | $distance(Vector2, Vector2)$ **;**
21. $expression \rightarrow var = expression \mid simpleExpression$
22. $var \rightarrow$ **ID** | **ID** **[** $expression$ **]**
23. $simpleExpression \rightarrow opExpression\ relop\ opExpression \mid opExpression$
24. $relop \rightarrow$ **<=** | **<** | **>** | **>=** | **==** | **!=**
25. $opExpression \rightarrow opExpression\ operators\ term \mid term$
26. $operators \rightarrow$ **+** | **-** | ***** | **/** | || | &&
27. $term \rightarrow term\ factor \mid factor$
28. $factor \rightarrow$ **(** $expression$ **)** | var | $call$ | **INT** | **FLOAT** | $Vector2$
29. $call \rightarrow$ **ID** **(** $args$ **)**
30. $Vector2 \rightarrow$ **< INT , INT >** | **< FLOAT , FLOAT >**
31. $args \rightarrow argList \mid \varepsilon$

32. $argList \rightarrow argList$ **,** $expression \mid expression$

ID = $letter\ (letter|digit)^*$

STRING = "$(letter|digit) *$ "

BOOL = $true|false$

INT = $digit\ digit^*$

FLOAT = $digit\ digit^*.\ digit^*$

$letter = a \mid ... \mid z \mid A \mid ... \mid Z$

$digit = 0 \mid ... \mid 9\ \backslash t$

Símbolos especiais: + - * / < <= > >= == != = , ; ( ) [ ] { } # “ ”

4.1 Revisões da gramatica

Do documento do trabalho anterior a este foram feitas tais revisões a gramatica:

1. O regex para NUM foi substituído por INT e FLOAT ja que faz mas sentido no código haver a distinção entre esses dois tipos de numero.
2. O tipo vector foi especificado para ser um vector de INTs ou FLOATs, dessa forma ficará mais fácil no futuro fazer operações somente em vectors do mesmo tipo .
3. O tipo bool foi adicionado, adicionado para realizar operações booleanas.
4. Os símbolos "true"e "false"foram adicionados como BOOL, para auxiliar o entendimento de operações booleanas.
5. Os terminais addOp e multiOp foram substituídos pelo terminal operators que inclui todas as operações, isso foi feito para simplificar a implementação.
6. as operações —"e ""foram adicionadas as operações da linguagem, isso foi necessário para linguagem poder fazer operações entre booleanos.

5 Semantica

1. A linguagem apresenta escopo estático.
2. As unicas converções implícitas serão de int para float, com a casa decimal do int sendo considerada como 0, e de float de para int, que sempre descartara a parte decimal.
3. A passagem de parametros e as atribuições se darão sempre por copia.

4. O ponto inicial de execução será sempre o método main, não sendo executado sem ele.

5. A primitiva vector só pode ser composta por dois ints ou dois floats.

6. Todos os outros construtos serão avaliados como em C

6 Analisador léxico

O analisador léxico a ser utilizado foi desenvolvido com o uso da ferramenta flex, que, uma vez definida as expressões regulares para os símbolos da nossa linguagem, gera o arquivo c++ que contém o código que fara a análise léxica desses símbolos definidos.

Ao ser executado em um arquivo txt contendo o código na linguagem desenvolvida, o analisador retorna todos os os símbolos (sejam eles identificadores, comentários, números inteiros, floats, vetores, strings, tipos, operadores, etc) na ordem que ele encontra ao fazer a analise. Ele também ao encontrar algum simbolo que não se encaixa em nenhuma da expressões regulares que identificam símbolos da linguagem retorna o simbolo não identificado na linha em que o encontrou, dessa forma ajuda desenvolvedores a facilmente identificar a presença do simbolo errado e remove-lo.

Na pasta aonde se encontra o analisador também se encontra quatro arquivos de txt de exemplo do funcionamento dele. Os arquivos "teste_correto1.txt"e "teste_correto2.txt"não apresentam erros léxicos, já os arquivos "teste_errado1.txt"e "teste_errado2.txt"apresentam símbolos não identificados pela linguagem e apresentarão erros léxicos ao serem rodados no analisador.

Referências

[1] João Victor Oliveira Eduardo Henrique Viva, Matheus de Souza Amorim. TendÊncias no mercado de games e sua importÂncia. `http://revista.faqi.edu.br/index.php/seminario/article/view/400` [Online; accessed 19-Março-2019].

[2] wiki. Física computacional. `https://pt.wikipedia.org/wiki/F%C3%ADsica_computacional` [Online; accessed 19-Março-2019].