Tradutor de linguagem baseada em c que trata com pares ordenados

Trabalho de Tradutores - parte 4

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.

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. Os vetores também podem ser de tamanho não arbitrários, permitindo-se criar e realizar as operações descritas com vetores de n dimensões, sendo que quando uma operação for realizada para vetores de tamanho distintos o vetor de menor tamanho é convertido para o tamanho do de maior tamanho preenchendo suas cordenadas faltantes com 0. 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. $declaration \rightarrow variableDeclaration\ \mathbf{;} \mid functionDeclaration$

4. *variableDeclaration* → TYPE **ID**
5. *functionDeclaration* → TYPE **ID** (*params*) *compoundStmt*
6. *params* → *paramList* | ε
7. *paramList* → *paramList* , *param* | *param*
8. *param* → TYPE **ID**
9. *compoundStmt* → { *stmtList returnStmt* }
10. *stmtList* → *stmtList stmt* | ε
11. *stmt* → *expressionStmt* ; | *conditionalStmt* | *iterationStmt* | *IOStmt* ; | *vectorStmt* ;
12. *stmt* → *declarationStmt* ; | *assingStmt* ;
13. *declarationStmt* → *variableDeclaration*
14. *expressionStmt* → *simpleExpression*
15. *conditionalStmt* → **if** (*simpleExpression*) *compoundStmt* |
    if (*simpleExpression*) *compoundStmt* **else** *compoundStmt*
16. *iterationStmt* → **while** (*simpleExpression*) *compoundStmt*
17. *returnStmt* → **return** *expression* ; | **return** ; | ε
18. *IOStmt* → *read*(*var*) | *write*(*var*)
19. *vectorStmt* → *normalize*(**VECTOR**) | *distance*(**VECTOR**, **VECTOR**)
20. *assingStmt* → *var* = *expression*
21. *expression* → *opExpression*
22. *var* → **ID**
23. *simpleExpression* → *opExpression relop opExpression* | *opExpression*
24. *relop* → **COMPARABLES**
25. *opExpression* → *opExpression operators term* | *factor*
26. *operators* → + | - | * | / | || | &&
27. *factor* → (*expression*) | *var* | *call* | **INT** | **FLOAT** | **VECTOR** | **BOOL**
28. *call* → **ID** (*args*)
29. *args* → *argList* | ε

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

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

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

BOOL = $true|false$

INT = $digit\ digit^*$

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

VECTOR = $<$ **INT** $(,$ **INT**$)^* > \mid <$ **FLOAT** $(,$ **FLOAT**$)^* >$

TYPE = $int|float|char|vector|bool|void$

COMPARABLES = $<= \mid == \mid >= \mid > \mid < \mid != =$

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

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

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

4.1 Revisões da gramatica

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

1. Regra **localDeclarations** foi removida da regra 9 par resolver problemas de shift/-reduce.
2. Regra **returnStmt** realocada para regra 9 com o intuito de permitir somente um return por função, essa decisão foi tomada pela facilidade de detectar mismatch de tipo no return da função quando ha apenas um return e por a pratica de mais de um return por função ser considerada uma pratica de programação ruim.
3. Remoção da regra **returnStmt** na regra 11 devido aos motivos apresentados acima.
4. Regra **localDeclarations** renomeada para **declarationStmt** pois o nome novo é mais adequado a sua nova utilização.
5. Mudanças feitas na regra 13, que agora transita somente para regra **variableDeclaration**, com o intuito de resolver problemas de shift/reduce.
6. Remoção da regra **term** da gramatica devido a sua redundância e para resolver problemas de shift/reduce.
7. Regra 25 transita diretamente para regra **factor** ao invés da regra **term** devido as rasões apresentadas anteriormente.

8. Substituição da regra **Vector2** pelo token **VECTOR** por fazer mais sentido já que que o tipo **VECTOR** é um lexema reconhecido na analise léxica e não na sintática.

9. Adicionada através da expressão regular **VECTOR** a capacidade de ter vetores com tamanho arbitrários na linguagem.

10. Realocações feitas no token de '**;**' na gramatica para resolver problemas de shift/-reduce.

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.

7 Analisador Sintático

O analisador sintático da linguagem foi desenvolvido utilizando-se a ferramenta **bison**. Para utiliza-la teve-se que escrever um código "analisadorSintatico.y"aonde as regras da

linguagem foram listadas e utilizadas para montar duas estruturas que searão utilizadas na pelo analisador semântico a ser implementado.

A primeira estrutura ,a tabela de símbolos, foi implementada como um hash com a biblioteca "uthash.h"recomendada pela professora. Essa tabela armazena todas as declarações de funções e variáveis encontradas no código. Na implementação cada elemento do hash contém informações como o nome do simbolo, se ele se trata de uma função ou variável, o tipo (de retorno no caso do simbolo ser uma função) da variável e o escopo em que se encontra o simbolo, ele também contém um token único por simbolo que é formado a partir da concatenação das informações listadas anteriormente.

A segunda estrutura é a arvore sintática, que representa o fluxo de execução das expressões da linguagem e que será utilizada pelo analisador semântico para executa-lo. Ela é formada por nós possuem ponteiros para outros dois nós, um da direita e o outro da esquerda, e possui informações como o nome do simbolo, o tipo de variavel do simbolo e o tipo do nó("FUNCTION", "VARIABLE", "OPERATOR", "VECTOR", "IO", "CONDITIONAL", "PARAMS"ou "STATEMENT").

Para a politica de tratamento de erros sintáticos esta sendo impressa, a cada erro sintático que o **bison** detecta durante a execução, uma mensagem de erro com a linha aonde o erro foi encontrado seguido da mensagem de erro do **bison**. Foi escolhido esse tratamento já que as mensagens de erro do **bison** já são bem explicativas do problema sintático, sendo o seu único problema não apresentar a linha do erro, essa parte da mostrar a linha foi implementada pois ajuda o desenvolvedor a identificar mais rápido aonde se encontra o erro.

8 Analisador Semântico

O analisador semântico da linguagem é executado em somente um passo junto com a análise sintática. Existem no código a possibilidade para 5 diferentes tipos de erros semânticos:

- *Erro de redeclaração de simbolo*: Toda vez que uma função ou variável é declarada a tabela de símbolos é checada, se esse simbolo já estiver na tabela o erro de redeclaração e lançado avisando que o simbolo em questão já foi declarada.

- *Erro de simbolo não declarado*: Quando um simbolo, seja variável ou função, é encontrado em uma regra que não seja de declaração, o algorítimo verifica a sua existência na tabela de símbolos, caso ele não seja encontrado o erro é lançado avisando que o simbolo não foi declarado previamente.

- *Erro de incompatibilidade de tipo*: Em qualquer operação da gramatica (soma, armazenamento, etc), o algoritmo faz a checagem se existem uma incompatibilidade de tipo entre os termos da operação, caso haja é lançado um erro apontando a incompatibilidade dos tipos em questão.

- *Erro de incompatibilidade de chamada*: Em chamadas de função se faz a verificação se os parâmetros passados são compatíveis com a da função previamente declarada,

para isso a tabela de símbolos armazena os parâmetros em ordem declarados pela função, caso tenha incompatibilidade de tipo em algum parâmetro, é lançado o erro avisando o tipo que era esperado pela função em contradição com o tipo que foi recebido.

- *Erro de incompatibilidade no retorno da função*: Quando uma expressão de retorno é encontrada seu tipo é passado para regra de sima até chegar na regra de declaração de função, dai o tipo da declaração é comparado ao tipo do returno, se tiver uma incompatibilidade é lançado um erro avisando o tipo de retorno que era esperado em contra partida do tipo de retorno feito.

- *Erro de não declaração da função main*: Quando o analisador sintático passa pela a ultima regra da gramatica, é feita a verificação se a função main esta presente na tabela de símbolos, caso não esteja o erro é lançado.

Como politica de tratamento de erro está se passando o tipo de erro semântico encontrado juntamente com a mensagem de erro semântico seguido da linha em que foi encontrado o tipo, os erros semânticos foram separados nesses 5 tipos pois dessa maneira a mensagem para cada um pode ser escrita de maneira mais clara. Por questão de organização os erros não são mostrados de uma vez, sendo armazenados numa lista de erros semânticos conforme são sendo encontrados para depois serem mostrados conforme a ordem em que foram encontrados.

A detecção de escopo para um simbolo é feita da seguinte maneira, a variável global "currentScope", inicialmente setada como "GLOBAL", dita o escopo atual de cada simbolo encontrado, ao se deparar com a regra 10 da gramatica, o analisador semântico sabe que esta dentro de um escopo de uma função, embora não saiba de qual, por isso "currentScope"é setado como -"e dessa forma todos símbolos adicionados a tabela possuirão o escopo -", quando o analisador encontra a regra 5 ele descobre qual função ele estava analisando e procura na tabela de símbolos os símbolos com escopo -"e troca para o nome da função, depois disso seta "currentScope"para "GLOBAL"denovo.

9 TAC

Apos a etapa de analise semântica do código, teremos como resultado das analises sintática,léxica e semântica a arvore sintática abstrata que representa o fluxo de execução do código compilado. De posse dessa arvore nosso tradutor gera um código de três endereços(TAC) que executa as operações da arvore a nível de maquina. Como a linguagem produzida se assemelha a linguagem C traduzir a maioria das operações necessárias foi de fácil implementação. A maior dificuldade foi em implementar as operações especificas da linguagem referentes a vetores. Na geração de código para estruturas condicionais, repetição e chamadas de função foi utilizada uma pilha para armazenar os rótulos necessários para a tradução correta do programa.

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].