Gerador de Código Intermediário

Pedro Henrique Santos Gonzaga - 140030131

Departamento de Ciência da Computção, Universidade de Brasília (UnB) - Abril/2021

1 Motivação e Proposta

Este relatório tem como objetivo apresentar a segunda parte do projeto da disciplina de Tradutores do curso de Ciência da Computação da Universidade de Brasília. O projeto é dividido em 4 partes e tem como finalidade a construção das etapas de análise de um compilador e seu gerador de código de três endereços. Nesta segunda etapa será apresentado um analisador sintático funcional, assim como a gramática a ser utilizada pelo tradutor no decorrer do semestre. A linguagem foco do projeto é a linguagem C com a adição de uma nova primitiva, onde sejam possíveis determinadas operações entre conjuntos. Com esta nova primitiva, o leque de possibilidades para a linguagem se expande, pois, em C puro, para se trabalhar com conjuntos, é necessária a declaração e manipulação de várias estruturas de dados, não havendo um padrão para tal. Portanto, a definição formal desta nova primitiva permite que suas operações sejam padronizadas e facilitadas.

2 Descrição da Análise Léxica

Para o desenvolvimento do analisador léxico, foram utilizadas expressões regulares para identificar os lexemas aceitos pela linguagem, assim como seus tokens auxiliares (pontuação, chaves, etc). As definições de todas as regras podem ser vistas no arquivo *skylang.l*. Quando o analisador é executado e lhe é dado um arquivo de entrada para análise, o mesmo inicia a leitura dos caracteres do arquivo seguindo as regras definidas pelas expressões regulares. Caso um lexema que não esteja previsto dentro das regras estipuladas seja encontrado, a função *LexicalError()* é chamada e uma mensagem de erro é mostrada ao usuário, explicitando a linha e a coluna do token inválido.

3 Descrição da Análise Sintática

Para o desenvolvimento da análise sintática, a gramática definida para o projeto foi escrita em forma de código utilizando-se a ferramenta **BISON** no arquivo skylang.y. Foi feita a integração com o analisador léxico (skylang.l) de forma que os tokens analisados são passados para o analisador sintático, que por sua vez avalia a sequência de tokens de acordo com a gramática. Caso um erro sintático seja encontrado, a função *yyerror()* é chamada, apontando a linha e a coluna do

erro sintático. Para que essa integração fosse possível, foi criada uma union no skylang.y, que permitiu passar as informações dos tokens entre os 2 analisadores. A gramática definida na etapa anterior também sofreu algumas mudanças, já que alguns erros só foram possíveis de se identificar ao executar-se o analisador sintático de fato.

4 Descrição da Análise Semântica

Para realizar a análise semântica, foi utilizado o algoritmo de 2 passagens. Foi criada uma variável global "passagem"para controlar qual parte do código deve ser executado em cada passagem. Na primeira passagem pelo código, é gerada a tabela de símbolos contendo os nomes, tipos e escopo de cada símbolo. Na segunda passagem, é feita a análise de fato. Para checar o escopo de cada símbolo, foi definida uma variável "EscopoAtual"que contém o escopo atual do código. Ao se encontrar um símbolo, é checado se o símbolo existe na tabela de símbolos. Caso exista, a variável "EscopoAtual"é comparada com o escopo da variável. Caso o escopo esteja errado, um erro é dado. O mesmo acontece caso o símbolo não exista na tabela de símbolos.

5 Tabela de símbolos

Para a implementação da tabela de símbolos foi utilizada uma biblioteca C que permite a criação uma tabela *Hash* [4], facilitando o processo de busca e adição dos símbolos à tabela (a *Hash table* também é mais performática do que salvar os símbolos em um vetor, por exemplo). A estrutura dessa tabela pode ser vista no arquivo symboltable.h. A tabela de símbolos armazena todas as funções e variáveis declaradas pelo programa, seus nomes, tipo (int, float, elem, ou set) e tipo do símbolo (variável, ou função), assim como um identificador único para cada entrada da tabela. Para esta nova etapa do projeto, a tabela de símbolos foi modificada para que fosse possível guardar também o escopo de cada símbolo.

6 Árvore Abstrata

Para a criação da árvore abstrata, foi utilizada uma nova estrutura de dados, definida no arquivo skylangTree.h. As regras sintáticas do arquivo skylang.y foram definidas como sendo do mesmo tipo da árvore, de maneira que, à medida que as regras sintáticas são avaliadas, nodos da árvore são criados. Ao final da execução do programa, a árvore é mostrada ao usuário por meio da chamada da função "print tree", que printa o nome das regras sintáticas utlizadas no momento da criação daquele nó.

7 Tratamento de Erros Semânticos

O analisador semântico implementado é capaz de apontar os seguintes erros:

1. Chamada de função não declarada;
2. Uso de variáveis não declaradas;
3. Uso de variáveis fora do seu escopo permitido;
4. Códigos sem a função main();
5. Funções com retorno do tipo errado;
6. Redeclaração de funções;

8 Geração de Código Intermediário

Para a geração de código intermediário, é feita, durante a segunda passagem, a tradução dos comandos em C para TAC (Three Adress Code). No modelo proposto por este trabalho, o cabeçalho .table ficará vazio, já que a cada declaração de variável, um registrador será alocado para esta variável. A tabela de símbolos foi atualizada em relação à etapa anterior do trabalho e agora apresenta o registrador que será utilizado pelo símbolo (valores -1 neste campo indicam que não será utilizado nenhum registrador para este símbolo, como é o caso de funções, ou parâmetros de funções).

9 Arquivos de Teste

Juntamento com o código foram disponibilizados dois arquivos de texto para teste. Os arquivos testeCorreto1.txt e testeCorreto2.txt possuem exemplos de código que não possuem erros sintáticos, léxicos, ou semânticos dentro da gramática definida para o projeto. Os arquivos testeErro1.txt e testeErro2.txt apresentam códigos com erros sintáticos, léxicos, ou semânticos que serão apontados (linha e coluna) ao se executar o programa.

10 Compilação e execução

Para a compilação do analisador sintático, em ambiente Linux, abra o terminal, navegue até a pasta com os arquivos .l e .txt e digite os seguintes comandos:

1 - flex skylang.l

2 - bison -d -v skylang.y

3 - gcc lex.yy.c skylang.tab.c -Wall -o skylang

Para executar o programa e realizar a análise dos arquivos de teste, digite o seguinte comando, também no terminal:

1- echo "arquivoDeTeste.txt"| ./skylang

Exemplo:

echo testeCorreto1.txt | ./skylang

Alternativamente, pode-se utilizar o o comando "make compile"para a compilação do código e os comandos make testeCorreto1, make testeCorreto2, make testeErrado1 e make testeErrado2 para realização dos testes.

Referências

1. Alfred, Aho., Monica, Lam., Ravi, Sethi., Jeffrey, Ullman: Compiladores: Princípios, Técnicas e Ferramentas. 2nd edition. Pearson, 2008
2. Vern Paxson, Flex Manual, https://westes.github.io/flex/manual/. Último Acesso 20 Março 2021
3. Robert Corbett, Bison Manual, https://www.gnu.org/software/bison/manual/. Último acesso 5 Abril 2021
4. Uthash Documentation https://troydhanson.github.io/uthash/userguide.html. Último acesso 5 Abril 2021
   Troy,Hanson., Arthur,O'Dwyer.

A Gramática

Foi definida uma gramática inicial a ser utilizada durante o projeto. Em comparação com a última etapa, várias regras foram modificadas e muitas outras adicionadas (para que fossem corrigidas os problemas apontados na etapa anterior).

1. *Programa* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *variable declaration* | *function declaration*
4. *variable declaration* → **Type ID**
5. *function declaration* → **Type ID** *(params) codeBlock*
6. *params* → params-list
7. *params-list* → *params-list* , *param* | *param*
8. *param* → **Type ID**
9. *codeBlock* → *codeBlock statement*
10. *statement* → *variable declaration* | *exp* | *ifStatement* | *ForallStatement* | *outPutStatement* | *inputStatement* | *callFuncStatement* | **Return** exp | forStatement;
11. *exp* → *SetExp* | *aritExp* | *relExp* | *assignmentExp* | *terminal*
12. *IfStatement* → **if** (exp) { codeBlock} | **if** (exp){ codeBlock} **else** { codeBlock }
13. *ForallStatement* → **forall** (exp in **ID**) { codeBlock }
14. *outPutStatement* → **write** (**STRING**) | **writeln** (**STRING**)
15. *inputStatement* → **read** (**STRING**)
16. *forStatement* → **For** (exp ; exp; exp) { codeBlock } | **For** (exp ; exp; exp) { codeBlock} **else** { codeBlock }
17. aritExp → *exp***exp* | *exp* + *exp* | *exp* - *exp* | *exp*/*exp*
18. *RelExp* → *exp* rel *exp*
19. *setExp* → **Add** (*termSet* **IN** *termSet*) | **Remove** (*termSet* **IN** *termSet*) | **Exists** (termSet **In** *termSet*)
20. *termSet* → **ID** | **Integer** | *setExp*
21. *assignmentExp* → *terminal* = *exp*
22. *rel* → > | < | <= | >= | ! = | == | | &&
23. *terminal* → **ID** | **Integer** | **Float**

ID = letter(letter | digit)*
Integer = -?digit+
Float = -?digit*[.]digit+
elem = {integer}|{Float}
STRING = ¨"
letter = [a-zA-Z]
digit = [0-9]

O regex completo de todas as construções pode ser visto no arquivo skylang.l

Palavras reservadas da linguagem: write, writeln, add, exists, for, forall, remove, in, empty, if, else, float, elem, set, return, read.