Analisador Sintático

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 de um compilador completo. 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 token 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 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 criado 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 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.

5 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 dentro da gramática definida para o projeto. Os arquivos testeErro1.txt e testeErro2.txt apresentam códigos com erros sintáticos que serão apontados ao se executar o programa. No testeErro1.txt existe um ; a mais (linha 2 coluna 8). No testeErro2.txt, os () vem antes do ID na declaração de função (linha 2, Coluna 9).

6 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- ./skylang

Por último, quando a mensagem "Digite o nome do arquivo a ser analisado" aparecer, digite o nome do arquivo de teste que deverá ser analisado e pressione a tecla ENTER. É necessário digitar a extensão do arquivo também, como exemplificado abaixo:

1 - testeCorreto1.txt

References

1. Alfred, Aho., Monica, Lam., Ravi, Sethi.: Compiladores: Princípios, Técnicas e Ferramentas. 2nd edition. Pearson, 2008
2. Flex Manual, https://westes.github.io/flex/manual/. Último Acesso 20 Março 2021
3. 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

A Gramática

Foi definida uma gramática inicial a ser utilizada durante o projeto. Em comparação com a última etapa, algumas regras sofreram algumas mudanças.

1. *Programa* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *variable declaration* c *function declaration*
4. *variable declaration* → **int ID** | **float ID** | **set ID** | **elem ID**
5. *function declaration* → **int ID** *(params) codeBlock* | **float** *ID (params) codeBlock* | **set** *ID (params) codeBlock* | **elem** *ID (params) codeBlock*
6. *params* → params-list
7. *params-list* → *params-list* , *param* | *param*
8. *param* → **int ID** | **float ID** | **set ID** | **elem ID**
9. *codeBlock* → *codeBlock statement*
10. *statement* → *variable declaration* | *variable assignment* | *exp* | *ifStatement* | *ForallStatement* | *outPutStatement* | *inputStatement* | *callFuncStatement* | **Return** exp;
11. *variable assignment* → **ID** = exp
12. *exp* → *SetExp* | *aritExp* | *relExp* | *terminal*
13. *IfStatement* → **if** (exp) codeBlock | **if** (exp) codeBlock **else** codeBlock
14. *ForallStatement* → **forall** (exp in **ID**) codeBlock
15. *ExistsStatement* → **exists** exp **in ID**
16. *AddStatement* → **add** exp **in ID**
17. *RemoveStatement* → **remove** exp **in ID**
18. *outPutStatement* → **write** (**ID**) | **writeln** (**ID**)
19. *inputStatement* → **read** (**ID**)
20. aritExp → *terminal*terminal* | *terminal* + *terminal* | *terminal* - *terminal* | *terminal*/*terminal*
21. *RelExp* → *terminal* rel *terminal*
22. *rel* → > | < | <= | >= | ! = | == | | &&
23. *terminal* → **ID** | **DIGIT**

ID = letter(letter | digit)*
NUM = digit+
letter = [a-zA-Z]
digit = [0-9]

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