Analisador Léxico

Manuela Matos Correia de Souza - 16/0135281

Universidade de Brasília

1 Motivação

1.1 Tradutores

Um tradutor é um software básico que realiza a conversão de um código fonte escrito em uma linguagem de programação para código fonte em outra linguagem, preservando a semântica do código original. Atualmente, os tradutores são softwares essenciais para qualquer atividade computacional, uma vez que todo programa moderno precisa ser traduzido de linguagens de alto nível para linguagem de máquina antes de ser executado. Tradutores contemporâneos também são responsáveis por apontar erros léxicos, sintáticos e semânticos nos programas e por otimizar os códigos que traduzem.

Como uma forma de estudo desses softwares, esse trabalho se propõe a documentar a implementação de um tradutor em quatro etapas, são elas: análise léxica, análise sintática, análise semântica e geração do código intermediário. As próximas seções deste documento discorrem sobre a primeira etapa da implementação: a análise léxica.

1.2 Linguagem

A linguagem para a qual o tradutor será implementado é um subconjunto da linguagem C acrescido de uma nova primitiva de dados e operações sobre ela. Essa primitiva é o tipo *set*, criado para facilitar a manipulação de conjuntos em códigos C. As operações relacionadas a *set* são operações de inclusão e exclusão de elementos, pertinência de elementos, seleção de elementos, iteração sobre o conjunto, verificação do tipo conjunto e atribuição.

2 Análise léxica

A primeira etapa realizada em um processo de tradução é chamada de análise léxica. Seu propósito é transformar o código fonte original em uma sequência de **tokens**, que serão usados posteriormente na etapa de análise sintática.

Inicialmente, o analisador léxico percorre os caracteres do código fonte e agrupa-os em unidades significativas chamadas de **lexemas**. Para reconhecer um lexema, o analisador léxico passa cada caracter lido por um autômato finito, que reconhece todos os padrões léxicos válidos da linguagem. Um lexema é encontrado quando o autômato chega a um estado de aceitação e não é possível realizar outra transição. Nesse momento, um novo token é gerado para esse lexema em correspondência com o estado que o aceitou.

2.1 Implementação

Nesse trabalho, o analisador léxico foi construído usando a ferramenta FLEX [1], que é responsável, entre outras coisas, por gerar o autômato finito que reconhece a linguagem especificada. Para isso, foram criadas definições regulares que representam os padrões da linguagem e a cada definição foi associada a ação de criar e imprimir um token. As seções seguintes descrevem brevemente as estruturas de dados usadas para isso.

2.2 Estruturas utilizadas na implementação

A estrutura central do analisador léxico é a struct token, que contém três elementos: um inteiro representando a classe do token, um inteiro representando um atributo opcional e um ponteiro representando uma entrada para a tabela de símbolos. Tokens podem ser de classe **ID**, **CONST**, **TYPE**, **RELOP**, **ARTOP1**, **ARTOP2** ou de classe própria (por exemplo, os símbolos terminais de pontuação e as palavras-chave como **IF, ELSE, FORALL**, etc). Para criar tokens das classes **ID** e **CONST**, é preciso criar e inserir o lexema encontrado na tabela de símbolos, e em seguida definir o ponteiro da struct como uma referência para essa entrada. Para criar tokens das classes **TYPE**, **RELOP**, **ARTOP1** e **ARTOP2**, o valor do atributo opcional deve ser definido de acordo com o lexema encontrado, equanto o ponteiro para a tabela de símbolos é definido como NULL. Por fim, para criar tokens das demais classes não é preciso inserir na struct informações de atributo opcional ou de referência.

A tabela de símbolos foi implementada como uma lista ligada de tabelas hash, onde cada tabela hash guarda símbolos de um determinado escopo do programa. Cada símbolo nessas tabelas é uma struct **sym**, que contém um campo para um identificador e um campo para tipo. As funções de manipulação da tabela de símbolos são de inicialização da lista global de tabelas, criação e inserção de uma tabela hash na lista, criação e inserção de um elemento nas tabelas hash. É importante ressaltar que a implementação atual é um protótipo da tabela de símbolos final, e que os ajustes necessários serão realizados nas etapas seguintes do projeto.

Padrões e caracteres não reconhecidos pelo analisador foram tratados como erros e sua localização é apontada ao longo da execução com auxílio das variáveis **line** e **column**.

3 Testes e compilação

Os arquivos usados para teste do analisador léxico foram **c1.txt**, **c2.txt**, **e1.txt** e **e2.txt**. Os dois últimos possuem erros lexicográficos, localizados respectivamente na linha 2, coluna 5; e na linha 4, coluna 5. Para executar o analisador léxico basta utilizar os seguintes comandos no diretório *src*:

```
flex lexAnlz.l
gcc -g -Wall lex.yy.c symbtable.c tokens.c
./a.out ../testfiles/nome_do_arquivo.txt
```

Appendix A

Gramática

1. *program* → *declarations*
2. *declarations* → *declarations declaration* | *declaration*
3. *declaration* → *varDecl* | *funcDecl*
4. *varDecl* → *type* **id ;**
5. *type* → **int** | **float** | **set** | **elem**
6. *funcDecl* → *type* **id (** *parameters* **) {** *stmtBlock* **}**
7. *parameters* → *paramList* | ϵ
8. *paramList* → *paramList* **,** *type* **id** | *type* **id**
9. *stmtBlock* → **{** *stmtBlock* **}** | *stmtBlock stmt* | *stmt*
10. *stmt* → *varDecl* | *exprStmt* | *condStmt* | *iterStmt* | *returnStmt*
11. *exprStmt* → *expr* **;** | **;**
12. *expr* → *assign* | *inExpr* | *outExpr* | *simpleExpr*
13. *assign* → **id =** *simpleExpr*
14. *inExpr* → **read (id)**
15. *outExpr* → **write (** *out* **)** | **writeln (** *out* **)**
16. *out* → *simpleExpr* | **char** | **string**
17. *simpleExpr* → *simpleExpr* || *disjExpr* | *disjExpr*
18. *disjExpr* → *disjExpr* && *negExpr* | *negExpr*
19. *negExpr* → **!** *negExpr* | *relExpr*
20. *relExpr* → *relExpr relOp artExpr1* | *artExpr1*
21. *relOp* → ≤ | < | == | > | ≥
22. *artExpr1* → *artExpr1 artOp1 artExpr2* | *artExpr2*

23. *artOp1* → **+** | **−**
24. *artExpr2* → *artExpr2 artOp2 factor* | *factor*
25. *artOp2* → ***** | **/**
26. *factor* → **id** | **(** *simpleExpr* **)** | *callStmt* | *constant* | **is_set** **(** *simpleExpr* **)** | *pertExpr* | **exists** **(** *pertExpr* **)** | *setOp* **(** *pertExpr* **)**
27. *setOp* → **add** | **remove**
28. *callStmt* → **id** **(** *args* **)**
29. *args* → *argsList* | ϵ
30. *argsList* → *argsList* **,** *simpleExpr* | *simpleExpr*
31. *constant* → **integer** | **float** | **empty**
32. *pertExpr* → *simpleExpr* **in** *simpleExpr*
33. *condStmt* → **if** **(** *simpleExpr* **)** *stmtBlock optElse*
34. *optElse* → **else** *stmtBlock* | ϵ
35. *iterStmt* → **for** **(** *rule* **;** *condition* **;** *rule* **)** *stmtBlock* | **forall** **(** *pertExp* **)** *stmtBlock*
36. *rule* → *assign* | ϵ
37. *condition* → *relExpr*
38. *returnStmt* → **return** *exprStmt*

References

1. Paxson, V., Estes, W., Millaway, J.: The flex manual (2016), https://westes.github.io/flex/manual/, last accessed 20 Feburary 2021