Trabalho Prático 2 - Analisador Sintático

João Victor Bohrer Munhoz - 16/0071101

Universidade de Brasília, Brasília, DF, 70910-900, Brasil
160071101@aluno.unb.br

1 Motivação

Listas encadeadas são uma das estruturas de dados mais utilizadas atualmente. Entretanto, apesar de sua popularidade, é também uma potencial fonte de problemas, especialmente para programadores inexperientes, uma vez que para linguagens que não possuem suporte nativo a listas é preciso criar e manipular as estruturas de dados que definem seu comportamento. Portanto, na linguagem C-IPL [Nal21] proposta nesse trabalho, através do suporte nativo a listas junto com algumas operações básicas sobre elas, temos como motivação os objetivos a seguir:

1. Diminuir a quantidade de erros e aumentar a qualidade geral do código devido à implementação nativa.
2. Aumentar a produtividade dos programadores, pois estarão livres de ter que lidar com a criação da estrutura de dados e poderão focar somente em sua utilização.
3. Aumentar a reusabilidade e manutenibilidade de código, pois deixando a estrutura de dados nativa evita-se que cada programador tenha que implementar sua própria versão e portanto não será preciso entender as características de cada implementação específica, uma vez que, por ser nativa, será a mesma para qualquer programa utilizando a linguagem.

Nas seções abaixo serão descritos com detalhes os passos tomados para a implementação do projeto.

2 Descrição

Nessa linguagem **C-IPL** foi criada uma nova primitiva *list* para lidar com listas, uma nova constante NIL para indicar uma lista vazia e novas operações sobre as listas, podendo-se dividi-las em construtores, operações unárias e operações binárias. A explicação detalhada dos operadores se encontra na especificação da linguagem [Nal21].

Para realizar a análise léxica, utiliza-se em grande parte as operações regulares. Portanto, começamos o arquivo com várias macros definindo essas expressões que serão usadas para reconhecer os lexemas do código analisado. Todas essas expressões regulares estão presentes na Tabela 1, que se encontra como apêndice desse documento.

Ainda no analisador léxico, a partir do momento que identifica-se um lexema, é chamada uma função para criar uma estrutura de dados com as informações necessárias para a criação do *token*. As informações extraídas do lexema são: a **posição** em linha e coluna no código, o **escopo** em que se encontra o lexema e por fim o valor do **próprio lexema**. Após esse processo, esse *token* é passado ao analisador sintático, que começará sua análise.

O analisador sintático, a partir da gramática definida para a linguagem, presente na última página desse documento, define uma árvore sintática abstrata que representará a estrutura do código fonte sendo analisado. A partir dessa representação, será feita a análise para saber se o código analisado está gramaticalmente correto ou não.

Cada nó da árvore é uma estrutura de dados contendo o **nome** do nó, **quatro nós filhos**, podendo eles estarem populados ou não, e um *token*. Para cada regra da gramática é definido como essa árvore se desmembrará, sendo **nome** o elemento utilizado para identificar o tipo de nó, os filhos para armazenar informações que serão utilizadas somente mais à frente e o *token* que possui informação essencial para a tomada de decisão do tradutor, tal como o tipo de operação aritmética a ser realizada.

Com a estrutura da árvore sintática abstrata definida, já é possível montar a tabela de símbolos. Sabendo que a tabela é formada pela declaração de todas as variáveis, funções e parâmetros do programa, sua construção é trivial caso se construa ela junto com a árvore, sendo somente necessário criar uma estrutura de dados com as informações necessárias e inseri-la na tabela a cada instância de uma declaração de variável, função e parâmetro na árvore.

A estrutura de dados da tabela de símbolos possui o **identificador** do símbolo, seu **escopo**, sua **posição** em linha e coluna no código, o **tipo** do símbolo e um identificador para saber se o símbolo é uma **função ou uma variável**. A tabela é organizada como uma lista encadeada para que possua crescimento dinâmico e não ocupe espaço desnecessário nem fique inesperadamente cheia durante a análise sintática.

3 Arquivos de Teste

Ao total foram disponibilizados 5 arquivos de teste na entrega, três deles corretos e dois deles com erros sintáticos apontados pelo analisador. Os itens corretos são:

- **Ex_Correct_0.c:** Arquivo de teste fornecido junto com a descrição da linguagem.
- **Ex_Correct_1.c:** Arquivo próprio de testes.
- **Ex_Correct_2.c:** Arquivo próprio de testes.

Os itens incorretos são;

- **Ex_Incorrect_1.c:** Há um ponto e vírgula faltando na linha 3, coluna 14 e falta declarar o tipo de retorno da função *main* na linha 9, coluna 1.
- **Ex_Incorrect_2.c:** Falta um dos itens da iteração *for* da linha 11 e falta a chave para fechar a iteração *for* da linha 17.

4 Instruções de Execução

O programa foi desenvolvido e compilado utilizando as seguintes versões das ferramentas:

- **OS:** Fedora 34 (Workstation Edition) x86_64
- **Kernel:** 5.13.15
- **Flex:** 2.6.4
- **Bison:** 3.7.6
- **GCC:** 11.2.1

Para compilar, entre na pasta raiz e digite:

```
$ flex -o src/lex.yy.c src/C_IPL_Lex-Analyzer.l
$ bison -d -o src/C_IPL_Syntax.tab.c
src/C_IPL_Syntax.y -Wcounterexamples
$ gcc -o tradutor src/C_IPL_Syntax.tab.c
src/lex.yy.c lib/*.c -g -Wall -Wextra -Wpedantic
```

Ou, caso queira usar o makefile, digite:

```
$ make all
```

Uma vez com o arquivo *tradutor* já na pasta raiz, digite os seguintes comandos para rodar o programa nos arquivos de teste:

```
$ make run_correct_1
$ make run_correct_2
$ make run_incorrect_1
$ make run_incorrect_2
```

Referências

[Nal21] Claudia Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034, Agosto 2021. Acessado pela última vez em 10/08/2021.

Tabela 1. Projeto Léxico do trabalho

Token	**Expressão Regular**
Tipo	int | float | int list | float list
Constante Int	[0-9]+
Constante Float	[0-9]+"."[0-9]+
NIL	(NIL)
Símbolo de Menos	[-]
Operação de Soma	[+]
Operação de Multiplicação	[*/]
Símbolo de Exclamação	[!]
Operação Lógica	(&&) | (||)
Operação Relacional	[<] | (<=) | [>] | (>=) | (==) | (!=)
Atribuição	[=]
Keyword if	if
Keyword else	else
Keyword for	for
Keyword return	return
Operação de Leitura	read
Operação de Escrita	write | writeln
String	\"(\\.|[^"\\])*\"
Operações Unárias de Lista	[?%]
Operações Binárias de Lista	[:] | (>>) | (<<)
ID	[A-Za-z_][A-Za-z0-9_]*
Ponto e vírgula	[;]
Abre parêntesis	[(]
Fecha parêntesis	[)]
Abre chaves	[{]
Fecha chaves	[}]
Vírgula	[,]

1. program → declList
2. declList → decl declList | decl
3. decl → varDecl | funDecl
4. varDecl → type ID **;**
5. type → **int** | **float** | **int list** | **float list**
6. funDecl → type **ID** (params) compoundStmt
7. params → paramList | ϵ
8. paramList → paramTypeList **,** paramList | paramTypeList
9. paramTypeList → type **ID**
10. stmt → expStmt | compoundStmt | ifStmt | forStmt | returnStmt | readFunc | writeFunc
11. expStmt → exp **;** | **;**
12. compoundStmt → **{** localDecls **}** | **{ }**
13. localDecls → varDecl localDecls | stmt localDecls | varDecl | stmt
14. ifStmt → **if** (logExp) stmt **then** | **if** (logExp) stmt **else** stmt
15. forStmt → **for** (exp **;** exp **;** exp) stmt
16. returnStmt → **return ;** | **return** exp **;**
17. readFunc → **read (ID) ;**
18. writeFunc → writeType **(** logExp **)** | writeType **(STRINGCONST)**
19. writeType → **write** | **writeln**
20. exp → **ID** = exp | logExp
21. logExp → logExp logOp listExp | listExp
22. logOp → || | &&
23. listExp → listExp listOp relExp | relExp
24. listOp → : | >> | <<
25. relExp → relExp relOp sumExp | sumExp
26. relOp → < | <= | > | >= | == | ! =
27. sumExp → sumExp sumOp mulExp | mulExp
28. sumOp → − | +
29. mulExp → mulExp mulOp unaryListExp | unaryListExp
30. mulOp → ∗ | /
31. unaryListExp → unaryListOp unaryExp | unaryExp
32. unaryListOp → ? | ! | %
33. unaryExp → unaryOp factor | factor
34. unaryOp → −
35. factor → **(** exp **)** | call | constant | **ID**
36. call → **ID (** args **)**
37. args → argList | ϵ
38. argList → logExp **,** argList | logExp
39. constant → **INTCONST** | **FLOATCONST** | **NIL**