Laboratório Prático 3 - Analisador Semântico

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**. Para controlar o escopo atual do lexema é utilizado um contador crescente e uma pilha de inteiros. Quando se encontra os lexemas { e (o contador é aumentado e o valor atual do contador empilhado (com *flags* para não abrir e empilhar dois escopos seguidos). Quando encontramos um lexema } desempilhamos o escopo no topo da pilha e o novo topo da pilha passa a ser o escopo atual. Após esse processo de criação do *token*, ele 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, um ***token*** e o **tipo de retorno** do nó. 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. O tipo de retorno do nó é utilizado para fazer auxiliar na verificação de tipos no analisador semântico. Por padrão, cada nó inicialmente tem como valor de retorno *"undefined"*.

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, um identificador para saber se o símbolo é uma **função ou uma variável** e um contador que armazena a **quantidade de parâmetros** que o símbolo possui, com esse número podendo ser diferente de zero somente em símbolos do tipo função. 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.

Uma vez definidas as estruturas de dados dos analisadores, já é possível descrever o processos de análise semântica. Foi decisão de projeto fazer a tradução em somente um passo, com análise semântica feita juntamente com a construção da arvore, o que por consequente faz com que as análises léxica, sintática e semântica ocorram concomitantemente.

Podemos separar os tipos de análise semântica realizados nesse trabalho em basicamente 5 tipos: checar por declaração de funções ou variáveis repetidas, checar a cada uso de uma função ou variável se ela existe na tabela de símbolos, checar tipos de retorno de expressões de acordo com os tipos de retorno dos *tokens* e nós filhos, checar quantidade e tipo de parâmetro de chamadas de função e checar se o tipo sendo de fato retornado por uma função bate com o tipo declarado de retorno.

A checagem por declaração de funções ou variáveis repetidas é talvez a mais simples das checagens, onde foi criada uma função que com o nome da função ou variável e o número do escopo atual retorna 1 se já existe uma função ou variável declarada naquele escopo ou 0 se não. É feita essa checagem para cada declaração de função, variável ou parâmetro no código.

A checagem se uma função ou variável existe na tabela de símbolos é feita a cada uso de uma função ou variável no código, sendo implementada de maneira similar à checagem anterior. Entretanto, ao invés de mandar o escopo atual para a função, manda a pilha inteira de escopos, onde a função atuará recursivamente procurando na tabela de símbolos um símbolo que possua o mesmo nome e pertença ao maior escopo no topo da pilha, desempilhando somente quando se tem certeza que não há nenhum símbolo com o mesmo nome no escopo no topo da pilha.

A checagem de tipos de retorno de expressões é talvez a checagem mais abrangente do trabalho, onde inclui vários tipos nós e expressões encontradas ao redor da árvore, portanto não haverá detalhes de como tudo é feito em cada nó. Para expressões de atribuição, o tipo de expressão do operador à direita deve ser compatível com o operador à esquerda, havendo suporte para *type casting* de acordo com a regra usual.

Na regra usual, se o operador que recebe o valor é do tipo *int*, o operador que é atribuído pode ser do tipo *int* ou *float*, havendo *casting* de *float* para *int* nesse último caso, e vice-versa. Caso o operador o operador que recebe o valor seja do tipo *int list* o operador que é atribuído só pode ser do tipo *int list*, valendo a mesma regra para o *float list*.

Em operações lógicas o tipo de retorno é sempre *int*, não importa as variáveis de entrada. Em operações com o uso do operador ! se o operando por do tipo *int list* ele retornará um *int list* e ser for um *float list* ele retornará um *float list*. Para qualquer outro caso ele retornará um *int* pois será considerada uma operação lógica unária.

Para operação binárias de lista, caso seja o operador construtor :, o primeiro argumento deve ser *int* ou *float* e o segundo argumento uma *int list* ou *float list*, com o retorno sendo o mesmo tipo do segundo argumento. Caso seja o operador *map* >>, o primeiro argumento deve ser uma lista unária (cujo argumento deve ser *int* ou *float)* e que retorna um *int* ou *float*. O segundo argumento deve ser uma *int list* ou *float list*, com o retorno dependendo do tipo de retorno da função. Já com o operador *filter* <<, o primeiro argumento deve ser uma lista unária (cujo argumento deve ser *int* ou *float)* e que retorna um *int* ou *float* e o segundo argumento deve ser uma *int list* ou *float list*, com o retorno sendo do mesmo

tipo do segundo argumento. As regras usuais de *type casting* se aplicam a essas operações.

Para as operações relacionais, o tipo de retorno delas é sempre *int*, havendo *type casting* nos operandos de acordo com as regras usuais. Somente em operações com os operandos ! = e == é possível comparar uma *int list* ou *float list* com a constante *NIL*. Para as operações aritméticas, os operandos podem ser somente *int* ou *float*, sendo feita sempre a conversão de *int* para *float* quando necessário fazer o *type casting*.

Para operações unárias de lista, operações com o operador ? retornam *int* caso o operando seja *int list* e *float* caso o operador seja *float list*. Já o operador % deve ter um operando do tipo *int list* ou *float list* retorna sempre o mesmo tipo do operando. O operador unário − deve ter um operando do tipo *int* ou *float* e também retorna sempre o mesmo tipo do operando.

Nós terminais retornam sempre um tipo fixo; o tipo com o qual foram declarados na tabela de simbolos no caso de IDs, *int* no caso de *tokens INT*, *float* no caso de *tokens FLOAT* e *NIL* no caso de *tokens FLOAT*. Muitas vezes são os valores dos terminais que determinam os tipos dos nós mais acima na árvore.

Provavelmente o mais complicado é verificar se a quantidade e os tipos dos parâmetros está correto. Para isso, foi-se aproveitado do fato de que o *Bison* utiliza uma construção *bottom-up* da árvore abstrata e para cada declaração de varável foi atualizado um contador global, que ao chegar na declaração da função em si continha o número exato de parâmetros da função, onde o guardávamos na tabela de símbolos. Quando é feita uma chamada de função, utiliza-se do mesmo método para contar os parâmetros colocados na chamada da função. Assim, caso o valor do contador na chamada da função seja diferente do que está na tabela de símbolos, é lançado o erro semântico.

Uma vez correta a quantidade de argumentos, procuramos as entradas dos argumentos na tabela de símbolos e comparamos os seus tipos com os tipos presentes na chamada da função, aceitando os argumentos ou fazendo o *type casting* caso esteja tudo correto de acordo com as regras usuais ou lançando um erro caso não estejam.

Para a checagem de tipo de retorno, todos os possíveis tipos dos retornos de uma função são armazenados em uma lista, que novamente aproveitando o fato de que o *Bison* utiliza uma construção *bottom-up* da árvore abstrata, quando chegamos à declaração da função em sim, utilizamos uma função recursiva para comparar todos os tipos de retorno da lista com o tipo declarado na função, aceitando os tipos de retorno ou fazendo o *type casting* caso esteja tudo correto de acordo com as regras usuais ou lançando um erro caso não estejam.

3 Arquivos de Teste

Ao total foram disponibilizados 5 arquivos de teste na entrega, três deles corretos e dois deles com erros sintáticos e semânticos 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:** Os erros sintáticos são que há um ponto e vírgula faltando na linha 8, coluna 12 e foi utilizada um virgula para separar o for em vez de ponto e vírgula na linha 15, coluna 15. Os erros semânticos são que o operando da operação tail deve ser uma INT LIST ou FLOAT LIST e não um INT na linha 24, coluna 24, e a função main não foi declarada em nenhum lugar do código. **QUAISQUER ERROS SEMÂNTICOS ADICIONAIS SÃO SUBPRODUTOS DOS ERROS SEMÂNTICOS APONTADOS!**
- **Ex_Incorrect_2.c:** Os erros sintáticos são que falta um dos itens da iteração *for* da linha 11, coluna 26, e que há um ponto e vírgula faltando na linha 15, coluna 18. Os erros semânticos são que não pode haver soma com um operando sendo uma lista na linha 20, coluna 16 e que falta o return da função main na linha 4, coluna 5. **QUAISQUER ERROS SEMÂNTICOS ADICIONAIS SÃO SUBPRODUTOS DOS ERROS SEMÂNTICOS APONTADOS!**

4 Instruções de Execução

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

- **OS:** Fedora 34 (Workstation Edition) x86_64
- **Kernel:** 5.13.19
- **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
$ 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**