Laboratório Prático 4 - Gerador de Código Intermediário

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

2.1 Analisador Léxico

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 de entrada do Flex [PEM16] com várias

macros definindo essas expressões que serão utilizadas pelo programa para criar o analisador léxico, que é o autômato que reconhece a mesma linguagem do conjunto de expressões regulares definidas no arquivo de entrada. 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 o lexema { o contador é imediatamente aumentado e o valor atual do contador empilhado. Quando ele encontra o lexema (o escopo é aumentado somente quando esse lexema for parte de uma declaração de função, pois espera-se que os parâmetros declarados sejam parte do escopo da função e não do escopo global. 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.

2.2 Analisador Sintático

O analisador sintático, a partir da gramática definida para a linguagem presente na última página desse documento e implementado pelo *Bison* [Fou21], tem o papel de determinar se a sequência de *tokens* que recebe do analisador léxico corresponde a uma sequência válida na linguagem, tendo como subproduto uma árvore sintática abstrata que representará a estrutura do código fonte sendo analisado.

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***, o **tipo de retorno** do nó, uma ***flag* de conversão de tipo** e o **nome da variável temporária** responsável pela operação do nó no TAC.

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 auxiliar na verificação de tipos no analisador semântico. Por padrão, cada nó inicialmente tem como valor de retorno *"undefined"*. A *flag* de conversão de tipo serve para avisar ao compilador se aquela variável deve sofrer conversão de tipo para INT ou FLOAT durante a geração do código intermediário. O nome da variável temporária serve para facilitar a geração do código intermediário.

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, variável ou um parâmetro**, um contador que armazena a **quantidade de parâmetros** que o símbolo possui, o **retorno padrão** daquele símbolo e seu **nome de variável no TAC**. O contador de quantidade de parâmetros pode ser diferente de zero e o retorno padrão pode ser diferente de "nenhum"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.

2.3 Analisador Semântico

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 árvore, 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 cinco 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 é compatível 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 de nós e expressões encontradas na 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 conversão implícita conversão implícita 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 coerção de *float* para *int* nesse último caso, e vice-versa. Caso 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 for 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 conversão implícita se aplicam a essas operações.

Para as operações relacionais, o tipo de retorno delas é sempre *int*, havendo conversão implícita 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 a conversão implícita.

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 NIL*. São os valores dos terminais que determinam os tipos dos nós mais acima na árvore, com exceção do NIL que tem tipo atribuído em seu primeiro uso

Provavelmente o mais complicado é verificar se a quantidade e os tipos dos parâmetros estão corretos. Para isso, foi-se aproveitado do fato de que o *Bison* utiliza uma construção ascendente da árvore abstrata e para cada declaração de variável 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 com base na posição em que a função chamada está na tabela (pois de acordo com a gramática, a declaração da função ocorre logo após a declaração de seus parâmetros) e o número de parâmetros que devemos procurar. Para cada parâmetro acessado, comparamos os seus tipos com os tipos presentes na chamada da função, aceitando os argumentos ou fazendo a conversão implícita caso esteja tudo correto de acordo com as regras usuais ou lançando um erro caso não estejam.

Na checagem de tipo de retorno, para cada nó de retorno que o programa entra ele checa se o tipo de retorno atual é compatível com o tipo de retorno da função na tabela de símbolos, aceitando os tipos de retorno ou fazendo a conversão implícita caso esteja tudo correto de acordo com as regras usuais ou lançando um erro caso não estejam.

Além disso, como pedido na especificação do projeto, foi checado se existe uma função *main* no código, simplesmente vendo se ela está declarada na tabela de símbolos, e atribuindo valores de retorno padrão para funções, sendo 0 o valor padrão de retorno de funções do tipo INT, 0.0 o de funções do tipo FLOAT e NIL o de funções do tipo INT LIST ou FLOAT LIST.

2.4 Geração de Código Intermediário

A quarta e última parte do trabalho se refere à geração de código intermediário. Como pedido, foi utilizada a ferramenta TAC [dOS21] para tal, que fornece a especificação do código intermediário a ser gerado assim como um modo de executar esse código para confirmar que ele está correto.

O código intermediário do TAC pode ser divido em duas partes principais: A seção *table*, formada pelo marcador .table seguido de zero ou mais definições de símbolo, uma por linha e a a seção *code*, formada pelo marcador .code seguido de zero ou mais instruções, uma por linha. De modo a possibilitar a construção do código intermediário em somente uma passada, foi criado inicialmente um arquivo separado para cada seção, os quais serão atualizados separadamente conforme a necessidade.

A construção da seção *table* é relativamente trivial, sendo necessário somente, para cada novo símbolo de variável ou parâmetro adicionado na tabela de símbolos, colocar as informações necessárias desse mesmo símbolo na seção *table*, colocando 0 como valor inicial para símbolos do tipo INT e 0.0 como valor inicial para símbolos do tipo FLOAT.

Entretanto, diferentemente da tabela de símbolos normal, ela não pode conter identificadores iguais, uma vez que não se utiliza diretamente a noção de escopo no código intermediário. Para resolver isso, durante a criação de um novo símbolo na tabela de símbolos, já é atribuído um nome único para cada símbolo na própria tabela, o qual será usado para identificar a variável ou parâmetro no código intermediário e é trivial acessar quando for necessário referenciá-lo.

Já a seção *code* é um pouco mais complexa de construir, uma vez que envolve vários tipo de operações diferentes, portanto será analisado cada tipo de operação separadamente, sendo elas atribuição, lógico-aritmética, relacional, controle de fluxo, sub-rotinas de sistema, operações de lista, chamadas de sistema e conversões de tipo.

Para atribuições simples utiliza-se a instrução *mov*, cujo primeiro parâmetro é o símbolo á esquerda do operando = e o segundo parâmetro o valor ou variável à direita do operando =. Para atribuições mais complexas são utilizadas as operações explicadas mais à frente.

Para operações lógico-aritméticas, utiliza-se as instruções *add*, *sub*, *mul*, *div*, *and*, *or* e *not*. O primeiro parâmetro é o identificador do símbolo, retirado da tabela de símbolos, ou a variável temporária, retirado do nó atual da árvore abstrata, no qual o resultado será gravado. Os próximos dois parâmetros são para os símbolos, variáveis temporárias ou valores os quais serão utilizado para realizar a operação em si, com exceção do *not*, que possui somente 2 parâmetros no total, sendo esse único restante também para realizar a operação.

Em caso de operações aninhadas, as variáveis temporárias especificadas em cada nó da árvore e o fato do *Bison* [Fou21] utilizar uma abordagem ascendente na criação da árvore faz com que as operações com maior prioridade sejam realizadas antes e gravadas em variáveis temporárias únicas, as quais serão utilizadas pelas operações seguintes para realizarem suas próprias operações e assim por diante. Isso vale para basicamente todas as operações do código intermediário gerado.

Para as operações relacionais, utiliza-se as instruções *seq*, *slt* e *sleq*, que funcionam de forma análoga às operações lógico-aritméticas, com o primeiro parâmetro sendo o endereço de destino e os outros dois os quais serão comparados.

Para as operações de controle de fluxo, utiliza-se as instruções *brz*, *brnz* e *jump*. Para elas é necessário a criação de *labels* **antes** e **depois** de toda a estrutura (no caso, *for* e *if*) para que ele possa ir e voltar conforme necessário. Para as duas primeiras instruções, o primeiro parâmetro é o *label* para o qual o programa deve pular caso o valor do segundo parâmetro esteja correto e no caso do *jump* ele deve pular incondicionalmente para o *label* no primeiro parâmetro.

Para as operações de sub-rotina de sistema, utiliza-se as instruções *param*, *call*, *return*. Elas servem principalmente para lidar com funções, onde é imperativo que no começo delas haja um *label* que as identifique para que seja possível acessá-las a partir da instrução *call*, onde então poderão ser passados os parâmetros na pilha global através da instrução *param*, realizar as operações dentro da função e retornar ao endereço da chamada através da instrução *return*, que também desaloca tudo que foi alocado dentro da função para que não haja conflito de valores em futuras chamadas de função.

Para as operações de lista é necessário que se defina como serão tratadas as listas no gerador de código intermediário. Planeja-se utilizar os vetores na seção *table* para representar as listas, onde elas serão atualizadas a medida que forem realizadas operações sobre elas, assim como seria nas listas. Para as operações

de lista em si serão utilizadas as operações já definidas anteriormente mas no contexto dos vetores em que as listas estão armazenadas no código intermediário.

Para as operações de chamada de sistema, utiliza-se as instruções *print*, *println*, *scani* e *scanf*. Cada uma dessa operações possui um único parâmetro que é o símbolo no qual será guardado o valor lido ou o qual ele imprimirá no terminal.

E por fim, para as operações de conversão de tipo utiliza-se as instruções *inttofl* e *fltoint*, o qual será determinado de acordo com qual o valor da *flag* de conversão de tipo que o compilador leu do nó no momento da operação.

Uma vez que toda a árvore tenha sido construída, o compilador voltará para a sua função principal e, caso não houver erros léxicos, sintáticos ou semânticos no código fonte analisado, ele combinará os dois arquivos onde estão a seção *table* e a seção *code* para formar o código intermediário completo, o qual poderá em seguida ser executado pelo TAC [dOS21] para verificar a acurácia do gerador de código intermediário e do compilador como um todo.

3 Arquivos de Teste

Ao total foram disponibilizados 4 arquivos de teste na entrega, dois deles corretos e com código intermediário funcionando no TAC e dois deles com erros sintáticos e semânticos apontados pelo analisador. O arquivo .tac é sempre gerado na mesma pasta que os arquivos de teste. Os itens corretos são:

- **Ex_Correct_1.c:** Arquivo que testa as operações de soma e subtração. Espera-se que a saída dos *prints* seja 15 e 5, nessa ordem.
- **Ex_Correct_2.c:** Arquivo que testa as operações de multiplicação e divisão. Espera-se que a saída dos *prints* seja 50 e 2, nessa ordem.

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 uma vírgula para separar uma das expressões do 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.
- **Ex_Incorrect_2.c:** Os erros sintáticos são que falta um dos itens da iteração *for* da linha 13, 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 o return da função main na linha 25, coluna 12 é do tipo int list, entretanto deveria retornar um tipo simples.

4 Instruções de Execução

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

- **OS:** Fedora 34 (Workstation Edition) x86_64
- **Kernel:** 5.14.12
- **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 -Wall
$ 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. No caso dos exemplos corretos, o TAC já executará os códigos intermediários automaticamente:

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

Referências

[dOS21] Luciano Henrique de Oliveira Santos. Tac - three address code interpreter. https://github.com/lhsantos/tac, Janeiro 2021. Acessado pela última vez em 25/10/2021.

[Fou21] Free Software Foundation. Gnu bison. https://www.gnu.org/software/bison/manual/html_node/index.html, Setembro 2021. Acessado pela última vez em 14/10/2021.

[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 14/10/2021.

[PEM16] Vern Paxson, Will Estes, and John Millaway. Fast lexical analyzer - flex. https://westes.github.io/flex/manual/, Outubro 2016. Acessado pela última vez em 14/10/2021.

Tabela 1. Projeto Léxico do trabalho

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