PanelC: Um tradutor de mini C para manipulação de tabelas

Khalil Carsten do Nascimento[15/0134495]

Universidade de Brasília, Departamento de Ciência da Computação

Abstract. Esse relatório apresenta descritivamente todo o processo de desenvolvimento do tradutor para a linguagem PanelC. Tal linguagem é objeto do trabalho final da matéria Tradutores oferecida pelo Departamento de Ciência da Computação da Universidade de Brasília.

Keywords: Tradutores panelC GLC LinguagemC

1 Introdução

Computadores e humanos necessitam de uma forma clara e prática para descreverem como resolver problemas computáveis. As linguagens de programação surgiram com esse objetivo, se tornando a base de qualquer programa computacional. Essas linguagens são de fácil entendimento para o ser humano mas necessitam ser traduzidas para um código executável para a máquina. Essa tradução é feita por programas chamados compiladores [2].

Esse trabalho se dedica a construir um tradutor para uma linguagem com propósito específico destinada a alguma área da computação. O desenvolvimento será divido em cinco etapas principais: escolha do tema, analisador léxico, analisador sintático, analisador semântico e gerador de códigos intermediários. Ao fim é esperado ter uma mini linguagem com funcionalidades que procuram solucionar algum problema da área escolhida e com as seguintes estruturas básicas:

1. Comando de leitura e escrita
2. Comando condicional
3. Tratamento de expressões aritméticas
4. Chamada de sub rotinas

2 Motivação

Atualmente vivemos em uma era em que são coletados dados digitais sobre tudo, a qualquer hora e em qualquer lugar. A maioria desses dados são guardados de forma não estruturada, o que leva companhias a passarem por dificuldades ao lidarem com tamanho volume desses dados. Baseado nisso se apresenta o principal desafio de extrair informações valorosas dessas bases de dados seja qual for suas características singulares [1].

Quando se trata de extrair base de dados comumente encontramos estruturas no formato de tabelas, definidas como uma lista de observações. Cada observação possui campos, identificados por nomes, que podem assumir vários tipos [4]. Tal estrutura é amplamente usada nas áreas de estatística e econometria [3] e tornou-se a principal representação de dados na área de ciência de dados [1]. Atualmente, linguagens modernas como, *python*, possuem bibliotecas que implementam tal estrutura, e, em alguns casos, são nativamente implementadas, como na linguagem R[1].

[1] https://www.r-project.org/

No caso da linguagem C uma estrutura de tabela não é de fácil implementação, pois dependeria de estruturas de dados as quais não são nativamente implementadas, o que levaria à criação de uma larga biblioteca. Tendo isso em vista uma linguagem compilada de rápida execução se torna uma ideia atraente quando se trata de tratamento de dados. Manipular quantidades grandes de dados utilizando estruturas de dados complexas, implementadas nativamente, além de operações diversas entre várias tabelas se apresenta como uma oportunidade. Propõe-se um tradutor para uma linguagem com sintaxe e semântica semelhante a C, a *panelC*. Essa possui os requisitos citados na Seção 1 e adicionalmente a estrutura de tabela como tipo nativo da linguagem, além de operações específicas e adaptações de laços de repetição para esse novo tipo. Tal estrutura será composta por uma organização característica de listas para representar a grade de uma tabela.

2.1 Elementos Estruturais de uma Tabela

Considerando a proposta desse trabalho, o entendimento da estrutura de uma tabela se torna importante para uma correta implementação da linguagem. Começando pelo elemento mais básico temos as células. Células são responsáveis por armazenar uma única unidade de dado. São mapeadas por uma par ordenado $(index, celula)$ e sempre conterão o mesmo tipo de dado da coluna a qual pertence. Uma coluna é identificada por uma $<string>$ e, uma linha, por um índice numérico.

Em um nível mais alto da estrutura temos uma lista de $<string>s$ que representa as colunas da tabela. Cada elemento dessa lista possui um ponteiro para uma lista adjacente que armazena as células daquela coluna. Essa abordagem possibilita o aumento dinâmico das colunas tornando a tabela flexível a manipulação, escrita e deleção de dados. Veja Figura 1.

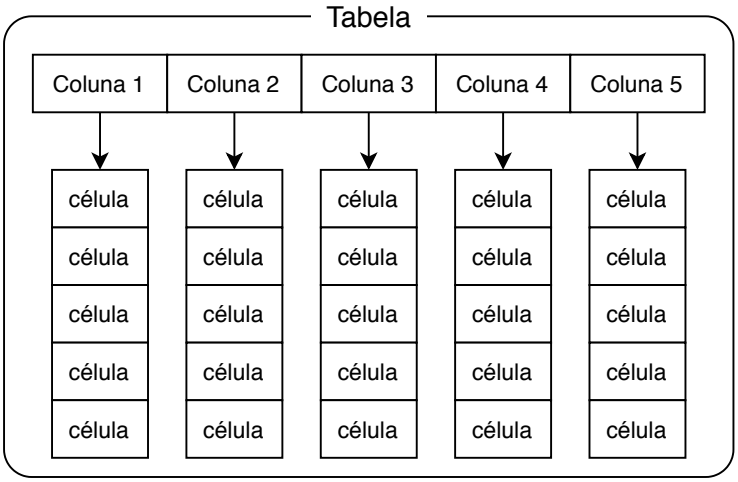

Fig. 1. Diagrama da organização de lista para formar a estrutura de tabela

2.2 Operações de Tabela

Seguindo para uma aplicação prática, para um eficiente tratamento de dados utilizando tabelas é necessários a algumas operações básicas de manipulação. A primeira delas é a concatenação, possibilitando unir uma ou mais

tabelas pelas suas linhas ou colunas. Em seguida a inserção, a capacidade de inserir tabelas dentro de outras sem mudar a estrutura original. Por último a deleção, possibilidade de deletar uma linha ou uma coluna sem deixar valores nulos no lugar.

Outra facilidade que a linguagem pode trazer é fornecer uma sintaxe mais intuitiva para laços de repetição em cima de estruturas de tabelas. Essa vantagem seria criar um laço de repetição onde o usuário poderá escolher de maneira simples se deseja iterar sobre as colunas, linhas ou células.

Abaixo temos um exemplo de uma possível aplicação da linguagem. O programa possui duas tabelas *alunoA* e *alunoB* declaradas usando o tipo *panel*. A atribuição dos valores às variáveis é feita de maneira parecida com um dicionário da linguagem *python*. Após isso uma observação é deletada de *AlunoA* seguida por um laço de repetição que reduz todas os valores da coluna de notas em 1. Ao final é mostrado na tela uma tabela que representa a concatenação de *alunoA* com *alunoB*.

```
panel alunosA;
panel alunosB;

alunosA = {"Notas": [10, 10, 9.5, 6, 8],
          "Alunos": ["Gabriel", "Joao", "Renato", "Alfredo", "Maria"],
          "Faltas": [2, 4, 6, 8, 10]}

alunosB = {"Notas": [7, 8, 9.5],
          "Alunos": ["Moutinho", "Nicholas", "Marcos"],
          "Faltas": [2, 4, 10]}

alunosA.deleteRow[3]

for rows["notas"] (cell;alunosA) {
    cell = cell - 1
}

print(alunosA v alunosB)
```

```
>>
index |   Notas   |   Alunos   |   Faltas   |
0          9          Gabriel        2
1          9           João          4
2          8.5        Renato         6
3          7          Maria          10
4          7         Moutinho        2
5          8         Nicholas        4
6          9.5        Marcos         10
```

3 Descrição Sintática

Para que um tradutor analise um programa da maneira correta é necessário que a linguagem traduzida siga uma sintaxe pré determinada. Para especificar uma sintaxe usa-se uma notação chamada gramática livre de contexto. Esta possibilita definir um conjunto de regras e terminais que construirão os padrões de <string>s encontradas no código do programa [2]. Abaixo são apresentadas 39 regras que descrevem a sintaxe da linguagem *panelC*.

1. programa → lista-declarações
2. lista-declarações → lista-declarações | declaração
3. declaração → dec-variável; | função
4. dec-variável → tipo <**identificador**> | tipo <**identificador**> [<**número**>]
5. função → tipo <**identificador**> (parâmetros) bloco
6. parâmetros → lista-parâmetros | ϵ
7. lista-parâmetros → lista-parâmetros | parâmetro
8. parâmetro → tipo <**identificador**> | tipo <**identificador**> []
9. bloco → { declarações-locais dec-lista }
10. declarações-locais → declarações-locais dec-variável | ϵ
11. dec-lista → dec-lista dec | ϵ
12. dec → expressão-op | condicional | iteração | iteração-tabela | return
13. expressão → variável = expressão; | expressão-simples | tabela
14. condicional → if (expressão-simples) bloco | if (expressão-simples) bloco else bloco
15. iteração → for (expressão;expressão-simples;expressão) bloco
16. iteração-tabela → for rows[<string>] (variável;variável) bloco | for cells (variável;variável) bloco
17. return expressão; | return;
18. variável → <**identificador**> | <**identificador**>[expressão]
19. expressão-simples → expressão-op cmp-op expressão-op | expressão-op
20. expressão-op → expressão-op operadores termo | termo
21. termo → termo operadores fator | fator
22. fator → (expressão-op) | variável | chamada | <**número**> | constante-bool
23. chamada → <**identificador**> (argumentos);
24. argumentos → lista-argumentos | ϵ
25. lista-argumentos → lista-argumentos, expressão | expressão
26. tabela → <**identificador**> = { lista-coluna }
27. lista-coluna → <**indentificador**>: [dados] | lista-coluna, lista-coluna | ϵ
28. dados → número | <string>
29. número → | <float> | <int> | ϵ
30. <identificador> → letra (letra|digito)*
31. <float> → digito$^+$. digito*
32. <int> → digito digito*
33. <string> → "Σ^*"
34. comentário → /*Σ^{**}/
35. operadores → +| − | ∗ |/|&&||| # retirada do operador 'v'
36. constantes-bool → true | false
37. IO → input() | print(fator)
38. letra → a | A | ... | z | Z
39. digito → 0 | ... | 9
40. tipo → **int** | **float** | **char** | **panel** | **string** | bool
41. cmp-op → <= | == | >= | > | < | != |

4 Descrição da Semântica

A linguagem segue uma semântica bem similar a linguagem C, utilizando chaves para identificação de blocos em trocas de fluxo e funções, parênteses para definição de expressões e colchetes para identificar posições de um

vetor. No caso de definições de funções e variáveis utiliza-se o padrão do tipo seguido pelo seu identificador. As novas adições feitas pelas linguagem PanelC se encontram na nova forma de laço para iterações em tabelas e suas definições. O usuário é capaz de indicar em qual coluna ele deseja iterar, em qual tabela e qual o nome da variável que irá receber o valor da célula lida. Já a definição de uma tabela se assemelha muito a definição de dicionário na linguagem *Python* onde uma tabela é definida utilizando chaves, identificadores e composições de lista.

5 Análise Léxica

Tendo definido uma gramática formal e a proposta de inovação da linguagem PanelC, pode-se agora iniciar as etapas de desenvolvimento de seu tradutor. A primeira etapa da construção de um tradutor é o analisador léxico. Esse é responsável por ler os caracteres do código fonte, agrupá-los em lexemas e produzir como saída uma sequência de *tokens* [2]. Será apresentado adiante uma introdução ao analisador léxico e como foi efetuada sua implementação.

5.1 Desenvolvimento

Um analisador léxico se baseia em três definições para executar sua tarefa. Primeiramente existem padrões no código, quando um padrão se apresenta instanciado no código é chamado de lexema. Ao final é construído o *token* a partir do lexema e do padrão encontrado. Entendido esses três conceitos, a função da análise léxica é encontrar padrões de lexemas no código e classificá-los quanto a um determinado *token* da linguagem. Mais tarde, essa lista será consultada pelo *analisador sintático*.

Tendo em vista que o objetivo dessa etapa é realizar a implementação do analisador léxico, deve ser definido um conjuntos de *tokens* e seus padrões, com esse intuito serão usados os terminais presentes na gramática apresentada anteriormente com alterações dependendo da necessidade.

Durante a implementação foi percebido várias falhas na definição da gramática inicial. Dentre essas falhas estão: (i) a falta de regra para introdução de comentários no código, (ii) a atual definição de número não aceita números reais, (iii) o caractere 'v' não é ideal que seja um operador, (iv) o tipo *string* não possuía definição, (v) não existe definição do tipo booleanos e (vi) as funções de entrada e saída não estavam definidas. Objetivando a implementação correta do tradutor, as devidas correções foram efetuadas na gramática e estão destacadas em vermelho na Seção 3.

5.2 Detalhes de Implementação

A implementação do analisador léxico se deu por meio da ferramenta Flex[2]. Esta é capaz de criar um *scanner* para reconhecer padrões descritos por expressões regulares em um código textual. O programa trabalha lendo um arquivo de entrada contendo descrições de regras e código na linguagem C. Ao final do processamento é gerado um arquivo "lex.yy.c" que, ao ser compilado e executado, inicia a análise léxica do código fonte passado para o mesmo.

O arquivo de entrada recebido pelo Flex deve seguir uma estrutura de definições, regras e funções, respectivamente. Começando pelas definições, estas são declarações de variáveis regex para auxiliar na montagem das futuras regras. Já na listagem de regras são descritos todos os padrões que serão reconhecidos pelo *scanner* e os códigos em C para caso o reconhecimento de cada padrão tenha sucesso. Nessa etapa a ordem de listagem

[2] http://westes.github.io/flex/manual/

das regras importa para a prioridade dos tokens. Por último as funções auxiliares são utilizadas na execução da função "main" e outras funções desejadas pelo usuário.

Para o desenvolvimento inicial do panelC um conjunto de definições foram escritas para facilitar a organização das regras no código. No arquivo "lexicalAnalyzer.l" são apresentadas as definições primárias dos *tokens* da linguagem.

Terminado as definições dos tokens as regras agora podem ser escritas. Uma sequência de regras são listadas e serão analisadas em ordem pelo *scanner*. Porém essa forma de definição gera um problema para o caso de *strings* e comentários. Tais tokens no código devem aceitar qualquer texto que esteja entre seus delimitadores. Isso implica que o *scanner* não pode reconhecer nenhum símbolo interno a estes elementos que não faça parte dos mesmos. Levando isso em consideração a funcionalidade de condicionais, fornecida pelo Flex, foi utilizada. Tal funcionalidade permite que após o reconhecimento de um padrão, seja possível limitar todos os outros reconhecimentos a um certo escopo identificado por um *label*. O código das regras e suas funções estará presente na entrega do próximo laboratório desse trabalho.

5.3 Tabela de Símbolos

Os *tokens* agora identificados e extraídos necessitam ser armazenados em ordem para que o analisador sintático faça futuras análises utilizando a gramática. Para isso foi implementado uma tabela de símbolos utilizando um vetor de estruturas do tipo *Token*, onde são armazenadas o lexema encontrado e seu respectivo tipo.

5.4 Tratamento de Erros

Como parte da responsabilidade de um analisador léxico a identificação de tratamento de erros deve ser executada para quaisquer lexemas que não pertençam à linguagem. Até o momento o analisador implementado do PanelC identifica os lexemas não pertencentes à linguagem e informa a linha e coluna que o carácter não reconhecido se encontra, afim de permitir a correção por parte do usuário. A implementação de tal funcionalidade se deu por meio de duas variáveis de contagem *current_line* e *current_colum* que ao identificar que nenhuma regra reconheceu o lexema de entrada, uma função mostra no terminal as informações do erro. A exemplo disso, abaixo um código fonte onde foi inserido o carácter '@' e uma *string* que não foi finalizada. Como saída o programa mostra o motivo do erro e sua localização.

```
alunosA.@deleteRow[3]
for rows["notas"] (cell;alunosA) {
    cell = cell - 1
}
print(alunosA + alunosB)
string s = "test
```

```
ID (alunosA) tamanho 7
------ERROR 1--------
símbolo: @
símbolo nao reconhecido pela analise lexica
Line: 1
Column: 9
-----------------------
```

```
------ERROR 2--------
símbolo:
Fim de string não encotrado
Line: 6
Column: 12
----------------------
```

Outro exemplo de erro se apresenta no código abaixo onde um comentário não foi finalizado:

```
alunosA.@deleteRow[3];
for rows["notas"] (cell;alunosA) {
    cell = cell - 1
}
/* Come
print(alunosA + alunosB);
string s = "teste"
```

```
ID (alunosA) tamanho 7
------ERROR 1--------
símbolo: @
símbolo nao reconhecido pela analise lexica
Line: 1
Column: 9
----------------------
------ERROR 2--------
símbolo:
Fim de string não encotrado
Line: 6
Column: 12
----------------------
```

5.5 Requisitos Mínimos e Execução

Para a correta execução do analisador léxico é necessário um ambiente Linux com o programa *Flex* instalado corretamente. O código em C gerado pelo *Flex* deve ser compilado pelo compilador GNU com as *flags* "-lfl" ou "-ll" para ambientes MACOS. A execução do programa compilado recebe como primeiro argumento o nome de um arquivo de entrada contendo um código PanelC.

6 Análise Sintática

A análise sintática é o próximo componente a ser desenvolvido em sequência ao analisador léxico. O código fonte, agora processado, se apresenta como uma lista de tokens e devem ser reconhecida pela gramática estabelecida dentro do analisador sintático. Como saída teremos uma árvore sintática que mais tarde será repassada para os outros componentes do tradutor.

6.1 Desenvolvimento

Durante a primeira etapa do compilador o analisador léxico processa da sequência de tokens e popula a tabela de símbolos com os identificadores e seus respectivos valores encontrados. Após isso o analisador sintático fará consultas a tabela de símbolos para construir a árvores sintática. A medida que os padrões da gramática são encontrados a tabela de símbolos além de consultada também é atualizada com informações a despeito dos identificadores, como quais identificadores representam funções e quais representam variáveis.

Para realizar a implementação do analisador sintático do PanelC foi utilizado o programa Bison[3], um gerador de analisadores sintáticos. A Utilização do Bison é bem semelhante a do programa Flex apresentado anteriormente, possui como entrada um arquivo divido nas estruturas de declarações, regras e funções auxiliares. As regras, diferentemente do *Flex*, são definidas utilizando o padrão BNF[4] de representação de gramáticas.

Como dito antes, a entrada do analisador sintáticos são as sequências de tokens extraídas pelo analisador léxico. Tendo isso em vista o Flex o Bison possuem uma interface de comunicação, a nível de compilação, sendo possível retorna os tokens reconhecidos pelo Flex diretamente para o Bison. Essa abordagem soluciona o problema de comunicação entre os analisadores.

6.2 Implementação da árvore sintática

O Bison é capaz de comparar regras da gramática com as entradas de token e executar alguma ação perante isso, porém a tarefa de criação da árvore sintática é por parte do usuário. Levando isso em consideração, uma estrutura de *structs* da linguagem C foi pensada para executar a implementação da árvore.

Cada nó presente na árvore representa uma regra da gramática e cada derivação são ponteiros para as respectivas *structs* das derivações. Além dos ponteiros mais informações são guardadas nas *structs* dependendo dos terminais que compõem as regras. Como por exemplo, a regra "DeclVar" da gramática armazena o tipo e o identificador da variável. A Figura 2 demonstra um diagrama exemplo do início da estruturação da árvore.

6.3 Implementação da Tabela Símbolos

Definido a implementação da árvore, resta agora a definição da tabela de símbolos. A tabela de símbolos cumpre um papel importante na consulta para os identificadores de cada variáveis e funções presentes no código. Portanto sua implementação deve ser feita utilizando uma estruturas de dados que permite consultas de baixa complexidade. Como solução, a estrutura de dicionário foi escolhida para esse fim. Os identificadores concatenados com o escopo da função foi usado como atributo de chave do dicionário. Os elementos identificados pela chave guardam o nome da variável, o tipo(int, float, panel), sua origem do nome (variável, argumento, função, parâmetro) e por último o escopo no qual ela foi encontrada.

Apresentado a estrutura da tabela símbolos uma outra dificuldade do projeto é reconhecer o atual escopo do identificador sendo analisado. A construção da árvore é feita começando pela pela criação dos filhos o que torna saber previamente qual será o escopo da variável que está sendo construída. Devido a isso, uma estratégia utilizando uma lista de variável foi criada. Durante a construção da árvore todas as variáveis encontradas são adicionadas a uma lista, ao encontrar a definição de uma função todas os identificadores presentes na lista são passadas para a tabela de símbolos com usando como o escopo a função encontrada.

A implementação da estrutura de dicionário foi feita utilizando o *header Uthash* da linguagem C, possibilitando criar um dicionário utilizando qualquer estrutura presente na linguagem. Para o caso da pilha e lista foi utilizado *headers* semelhante chamados *utstack* e *utlist* que implementam operações básicas dessas estruturas de dados.

[3] https://www.gnu.org/software/bison/manual/bison.html#Introduction
[4] http://www.garshol.priv.no/download/text/bnf.html

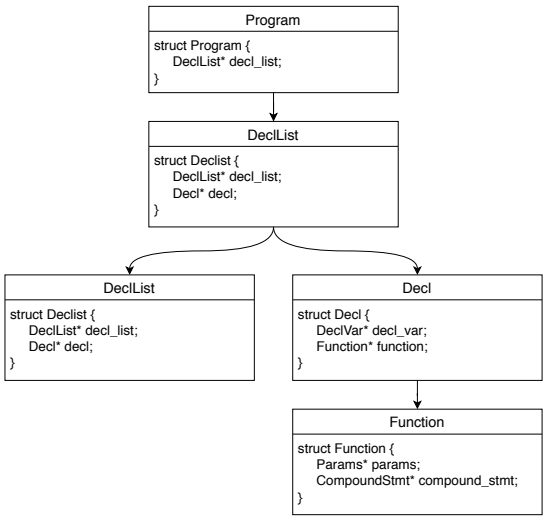

Fig. 2. Diagrama da organização de lista para formar a estrutura de tabela

6.4 Alterações no Projeto

A gramática do projeto PanelC apresentava vários problemas referentes a conflitos de *shift* e redução, além de ciclos dentro das próprias regras. Devido a isso, várias mudanças foram feitas e são apresentadas na seção 3. Outra mudança importante foi me relação as primeiras estruturas propostas no analisador léxico. As estruturas de tokens e erros foram desfeitas já que os tokens agora são passados diretamente para o *Bison*.

6.5 Política de Tratamento de Erros

O analisador sintático possui a tarefa de indicar erros sintáticos presentes nó código. Para esse trabalho os erros léxicos e sintáticos são diferenciados, o analisador sintático identifica a linha e o *token* esperado, já o analisador léxico identifica padrões que não pertencem à linguagem e apresenta a linha e a coluna que se encontra o símbolo. Abaixo duas exemplificações de ambos os erros.

```
=========Syntax Error========
16: syntax error, unexpected RCH, expecting $end
============================
-------Lex Error 1-------
Simbolo: &
Symbol does not belong to the grammar
Line: 3
Column: 2
-------------------------
```

6.6 Execução

Para a correta execução do programa o ambiente computacional deve ter um compilador GCC juntamente com os programas Flex e Bison instalados em um sistema operacional Linux. Nos arquivos fornecidos para esse trabalho

está incluído um *Makefile* na pasta raiz, executar o comando "make" no terminal irá efetuar todo o procedimento de compilação. O comando Bison já acompanha as opções "-vd" para facilitar a correção. A execução pode ser feita utilizando o comando "./bin/panelc <nome do arquivo>" a partir da pasta raiz. Na pasta testes estão presentes quatro exemplos de códigos dois sintaticamente errados e outros dois corretos. Para executa-los basta usar o comando "./bin/panelc testes/correto1.pc" por exemplo.

References

1. van der Aalst, W.: Data science in action. In: Process mining, pp. 3–23. Springer (2016)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles techniques and tools. 2007. Google Scholar Google Scholar Digital Library Digital Library (2006)
3. Kiviet, J.F.: Econometric analysis of panel data: Editorial introduction. The Singapore Economic Review **54**(03), 313–317 (2009)
4. McKinney, W.: Data structures for statistical computing in python. In: Proceedings of the 9th Python in Science Conference. vol. 445, pp. 51–56. Austin, TX (2010)