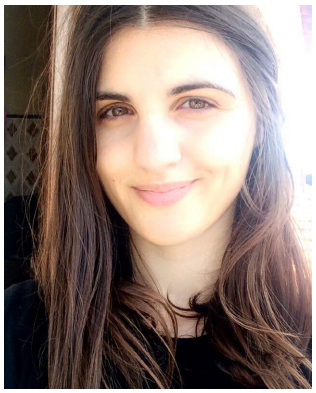
Departamento de Engenharia Informática
Mestrado Integrado em Engenharia Informática
Processamento de Linguagens

Universidade do Minho

Trabalho 2

Compilador de uma LPIS (Linguagem de Programação Imperativa Simples)

Grupo 38


Célia Figueiredo a67637

Gil Gonçalves a67738


Diogo Tavares a61044

Braga, 27 de Maio de 2016

Resumo

Este relatório descreve o processo de desenvolvimento e decisões tomadas para a realização do segundo trabalho prático da Unidade Curricular de Processamento de Linguagens.
O problema a resolver consiste na criação de uma linguagem imperativa simples (LPIS) e da respetiva gramatica independente de contexto **GIC** seguida do desenvolvimento de um compilador para a mesma que gera pseudo-código *Assembly*.
A linguagem desenvolvida foi baseada na linguagem de programação C, e suporta:

- Variáveis globais
- Ciclos: for, while, do while
- Estruturas de Condição: If ... Else
- Expressões Aritméticas e lógicas
- Funções com argumentos
- Declaração de variáveis locais dentro das funções

O compilador foi desenvolvido com recurso ao analisador léxico Flex e ao analisador sintático Yacc.

Conteúdo

Capítulo 1

Introdução

O presente trabalho enquadra-se na unidade curricular de Processamento de Linguagens da Licenciatura em Engenharia Informática da Universidade do Minho. O trabalho pretende aumentar a experiência em engenharia de linguagens, assim como incentivar o desenvolvimento de processadores de linguagens e compiladores em ambiente Linux. As ferramentas que suportam a sua implementação consistem no conjunto flex-yacc, sendo estes um gerador de analisadores léxicos e um gerador de analisadores sintáticos/semânticos, respetivamente.

Inicialmente é definida uma linguagem de programação imperativa simples, a qual deve permitir manusear variáveis do tipo inteiro assim como realizar operações básicas. As variáveis devem sem declaradas no início do programa e não pode haver re-declarações. Após a validação da linguagem criada com o docente, é desenvolvido um compilador para esta linguagem, com base na GIC criada e com recurso ao Gerador Yacc/Flex. O compilador da linguagem deve gerar pseudo-código Assembly da Máquina Virtual fornecida.

Estrutura do Relatório

Este documentos está dividido em xxxxx partes No capítulo 2 faz-se uma análise detalhada do problema proposto de modo a poder-se especificar as entradas, resultados e formas de transformação do ficheiro *.bib*.

No capítulo 3 serão descritas as estruturas de dados implementadas para a realização do problema descrito. No capítulo 4 serão mostradas as expressões regulares desenvolvidas para a implementação do caso de estudo, assim como os testes realizados e os resultados. No capítulo 5 termina-se o relatório com uma síntese do que foi dito, as conclusões e o trabalho futuro. No fim do documento estará incluido ficheiros anexos com o código desenvolvido.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema - Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples LPIS, a seu gosto.
Apenas deve ter em consideração que a LPIS terá de permitir:

- *declarar e manusear* variáveis atómicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- *declarar e manusear* variáveis estruturadas do tipo *array (a 1 ou 2 dimensões) de inteiros*, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- *efetuar* instruções algorítmicas básicas como a *atribuição de expressões a variáveis*.
- *ler* do *standard input* e *escrever* no *standard output*.
- *efetuar* instruções para controlo do fluxo de execução— *condicional* e *cíclica*—que possam ser aninhadas.
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado atómico (opcional).

Neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero).

Como descrito acima o desafio que nos foi proposto consiste na criação de uma linguagem imperativa simples apenas com as características mais básicas, o desenvolvimento de uma **GIC** para a mesma e ainda a criação de um compilador gerador de **pseudo-código** Assembly, utilizando os recursos ao Gerador YACC/LEX.

Posto isto tivemos então de, primeiramente, pensar e desenhar uma linguagem imperativa ao nosso gosto, seguindo os requisitos pedidos. De seguida foi necessário criarmos uma gramática que a descrevesse e, finalmente, com recurso a essa gramática, tornar possível o armazenamento dos dados vindos de um programa escrito na nossa linguagem necessários à geração do **pseudo-código** Assembly.

2.2 Especificação do Requisitos

2.2.1 Dados

Os dados fornecidos são um ficheiro *.bib*, este que é um ficheiro com as caracteristicas de um ficheiro *BibTex*.
Cada tipo de categoria tem os seus campos obrigatórios, neste acaso o objetivo de uma das tarefas será pesquisar através do campo **author =** o nome do autor e transformá-lo no formato "N. Apelido".

O nome da categoria é seguida por uma chaveta, e o primeiro campo será o nome para a referência a ser introduzida, os campos de cada categoria são separados por vírgula e a seguir ao campo aparecerá o simbolo igual (=), deixámos um exemplo da sintaxe da categoria @**phdthesis** e os respetivos campos presentes:

```
@phdthesis{Mos75a,
author = "P. D. Mosses",
title = "Mathematical Semantics and Compiler Generation",
year = 1975,
school = "Oxford University",
annote = "compilacao incremental, atributos, ambientes prog"
}
```

É também fornecido o nome de ferramentas de apoio à resolução do problema, neste caso o *Graph Viz*, que permitirá colocar gráficamente a informação dos grafos criados, sendo que tornará as iterações entre os autores mais percetiveis.

Capítulo 3

Concepção/desenho da Resolução

Como representado na figura 1 a estrutura do nosso projeto terá de passar pelo desenvolvimento de um analisador léxico em *Flex* que fará o reconhecimento dos *tokens* que são utilizados no analisador semântico que gera o código *Assembly*.

3.1 Desenho da Linguagem

3.2 Desenho da Gramatica

3.3 Estruturas de Dados

3.3.1 Stack

De forma a evitar confusão na atribuição de *labels* relativas a *ifs* e *loops* é utilizado um contador de condições. À medida que é encontrada uma instrução que implique o uso de uma condição, este contador é incrementado e o seu valor é colocado numa stack. Deste modo, o valor que se encontra no topo da stack é relativo ao último *ciclo/if* encontrado. Sempre que é encontrado o final de uma condição, o valor no topo da stack é removido. Através do uso de um contador e de uma stack, é muito mais simples gerir as *labels* e as operações de controlo, como *JUMPs* e *JZs*. A stack utilizada implementa apenas as funções necessárias para a sua inicialização, inserção, remoção e consulta. Com as operações de push/pop são inseridos/removidos valores no topo da stack, e com a operação de get apenas é consultado o valor no topo da stack, sem que este seja removido. Esta última operação é útil para a geração de instruções 'JZ' na geração de código VM.

3.3.2 HashMap

Como foi referido anteriormente, é utilizada uma *hashmap* com o objetivo de guardar as variáveis e as funções. Quanto às variáveis, é necessário guardar e aceder a informação como o seu endereço e tipo. Sendo assim, foi criada uma estrutura de dados auxiliar para armazenar essa informação. O nome da variável é utilizado como chave e, a partir dela, conseguimos aceder à sua informação correspondente na hashmap. Relativamente às funções, é necessário guardar e aceder a informação como o seu nome, informação relativa aos seus argumentos de entrada e o tipo de dados de saída. Para esse feito, também foram necessárias estruturas de dados auxiliares como uma lista ligada capaz de armazenar informação relativa aos argumentos de entrada e uma outra que contem a informação relativa ao tipo de dados de saída, dados de entrada e nome da função. Neste caso, o nome da função funciona como chave na *hashmap* e o seu valor é a estrutura que contem a informação mais geral sobre a função. A *hashmap* utilizada implementa funções necessárias para a sua inicialização, inserção, remoção e consulta, sendo que também contém outras funções que não foram utilizadas no desenvolvimento deste projeto.

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

4.1.1 Makefile

O principal objetivo da Makefile é facilitar a compilação e execução do programa. Para isso criamos o seguinte ficheiro:

4.2 Testes realizados e Resultados

4.2.1 Teste 1

4.2.2 Teste 2

4.2.3 Teste 3

4.2.4 Teste 4

4.2.5 Teste 5

4.2.6 Teste 6

Capítulo 5

Conclusão

Após a conclusão deste trabalho verificámos que não cumprimos todos os objetivos pedidos no enunciado.

Na alínea a pressupusemos que as categorias com o mesmo nome e que se diferenciavam apenas em maiúsculas ou minusculas pertenciam à mesma categoria. Pois assim não estariamos a repetir informação.

Na alínea b1 tratamos os nomes dos autores de modo a que ficassem com o formato **N.Apelido**, utilizando arrays.

Na alinea b2 não conseguimos colocar no início das categorias os campos autor e título.

Na alinea c foi implementado que o utilizador escolhesse o autor e assim gerar o grafo com os autores que publicam diretamente com ele. Um aspeto a ser melhorado poderia ser em vez de aparecerem as várias linhas que correspondem ao número de publicações conjuntas colocar apenas uma com a contagem.

Como trabalho futuro pretende-se melhorar os algoritmos que permitam um melhor funcionamento dos filtros.

Apêndice A

Código do Programa

Lista-se a seguir o código que foi desenvolvido para a alínea a:
Lista-se a seguir o código que foi desenvolvido para a alínea b1:
Lista-se a seguir o código que foi desenvolvido para a alínea b2:
Lista-se a seguir o código que foi desenvolvido para a alínea c: