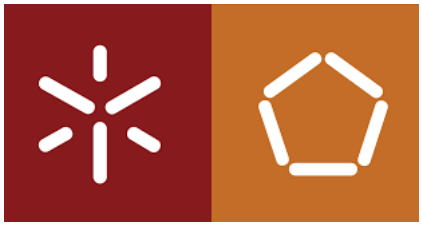

Universidade do Minho
Escola de Engenharia | Departamento de Informática
Licenciatura em Engenharia Informática

Processamento de Linguagens

Ano Letivo de 2023/2024

Projeto Final

Ricardo Miguel Queirós de Jesus (a100066)
Rui Pedro Fernandes Madeira Pinto (a100659)

PL

Índice

1 Introdução

O presente relatório descreve o trabalho prático realizado pelo Grupo 25 no âmbito da Unidade Curricular de Processamento de Linguagens do terceiro ano do curso de Engenharia Informática na Universidade do Minho, tem como objetivo o desenvolvimento de um compilador para Forth. Destina-se a produzir código para uma máquina virtual, oferecendo um ambiente controlado para explorar em detalhe as particularidades do *FORTH*. O relatório está dividido em várias secçõoes. Iniciamos com a introdução, que apresenta uma visão geral do trabalho realizado. Em seguida, descrevemos o problema a ser resolvido. Posteriormente, apresentamos a conceção/desenho da solução, detalhando as estratégias adotadas para o desenvolvimento do conversor. Prosseguimos com a secçao de testes, onde demonstramos alguns testes de funcionalidades do programa.Por fim, concluímos o relatório com uma síntese das principais conclusões obtidas ao longo do projeto.

2 Descrição do Problema

O projeto visa o desenvolvimento de um compilador para a linguagem *FORTH*, reconhecida por sua eficiência e flexibilidade, especialmente em sistemas embarcados e de baixo nível. A proposta consiste em criar um compilador capaz de traduzir programas escritos em Forth para código executável em uma máquina virtual específica. Utilizando ferramentas como Yacc e Lex em Python, o compilador deve ser capaz de realizar análise léxica e sintática dos programas *FORTH*, abrangendo desde expressões aritméticas até a manipulação de strings e variáveis.

Em resumo, o objetivo é desenvolver uma solução robusta e eficiente que possibilite a criação de programas complexos em *FORTH*, garantindo sua execução adequada na máquina virtual fornecida.

3 Concepção/desenho da Resolução

Neste capítulo iremos abordar a fase de conceção e desenho da solução para o desenvolvimento do conversor de linguagem Ford para lingugem da Virtual Machine. Nesta etapa, iremos explorar as duas componentes essenciais do projeto: o *Expression_lex* (Analisador Léxico) e o *Parser_* (Analisador Sintático).

O *Expression_lex* será responsável por realizar a análise léxica do código fonte em *FORTH*, identificando os diferentes tokens presentes no ficheiro. Já o *Parser_* irá realizar a análise sintática, definindo a estrutura gramatical do código *FORTH* e permitindo a construção do código da virtual machice correspondente.

3.1 Lexer

3.1.1 Definição de Tokens

Para o Lexer foram defidos diversos tokens que serviram para no texto lido identicar palavras/expressões/carácteres chave que permitissem identificar que situação estava presente. Os diversos tokens podem se categorizar da seguinte forma:

- **Literals :** Aqui definimos aqueles que queremos na gramática definimos diretamente, a razão para incluir este literals é principalmente para facilitar a leitura da gramática. Nesta categoria estão definidos: '(','),'=','!'.
- **Operadores Aritméticos :** Nesta categoria identificamos os diversos operadores das operações aritméticas. Aqui defimos as operações de soma, multiplicação, subtração, divisão e resto da divisão inteira;
- **Funções Pré-Definidas :** Nesta categoria identificamos algumas funções pré-definidas na linguagem *FORTH*. As funções que temos definidas são: ' . ', *CHAR*, *EMIT*, *SPACE*, *CR*, *KEY*, *DUP* e *SPACE*.
- **Tipos de Dados e Identificadores:** Para categoria identificamos tipos de dados. Temos o *NUM* para um qualquer inteiro, *ID* para identificar variaveis e nome de

funções e temos o *STRING* que identifica toda a string desde que esta esteja entre aspas.

- **Controles de Estrutura :** Nesta categoria identificamos esturas de features da linguagem. Temos o *VARIABLE* para idenficar a declaração de uma variável nova, os *DEF_START* e *DEF_END* para identificar a definição de uma função e os *DO* e *LOOP* para identicar um ciclo.

3.1.2 Estrutura do Código

A linguagem foi concebida com o propósito de permitir a manipulação de expressões aritméticas básicas, como adição, subtração, multiplicação e divisão. Essas operações são essenciais para a realização de cálculos matemáticos fundamentais. Além disso, a capacidade de interpretar e executar expressões aritméticas complexas é crucial para tornar a linguagem prática e útil em cenários de uso real.

Além das operações aritméticas convencionais, também foi definida a operação de módulo, representada pelo símbolo '%'. Esta operação calcula o resto de uma divisão inteira, proporcionando uma funcionalidade adicional importante para diversos tipos de cálculos e algoritmos.

Para além disso temos a inclusão das seguintes funções pré-definidas do *FORTH*:

- **CHAR :** Retorna o código ASCII de um caráter específico.
- **DUP :** Duplica o topo da stack.
- **SPACE :** Imprime um único espaço, útil para formatação de texto.
- **KEY :** Permite ler um input do usuário, facilitando a interação em tempo real.
- **'.' :** Imprime um valor, para tem de ser posto à frente do número e para Strings tem de estar atrás.
- **CR :** Imprime um parágrafo, facilitando a formatação de texto
- **EMIT :** Retorna o caráter correspondente a um código ASCII.

Tem então a declaração de variáviveis, que faz através do uso do VARIABLE, vai sendo guardado o número de variáveis existentes no programa para prosteriormennte ser alocado o espaço necessário.

De uma forma semalhante, as funções também vão sendo guardadas para quando posteriormente chamadas ser executado o seu código atribuido. Sendo que para declarar estas é necessário primeiro usar *STARTDEF*, depois escrever o nome da função, seguido do bloco de código da função e finalizar com o *FINISHDEF*.

O programa ainda permite a inclusão dos ciclos, estes vem dentro de funções e necessitam da atribuição de um limite superior e outro inferior para as interações após isso começasse com o *DO* escrevesse o bloco de código para o ciclo e finalizasse com o *LOOP*.

3.2 Parser

3.2.1 Visão Geral da Gramática

Para compreender profundamente a implementação e o funcionamento do analisador sintático desenvolvido com YACC, é essencial realizar uma análise detalhada de cada componente da gramática definida. Este analisador desempenha um papel central na tradução e interpretação do código-fonte, convertendo-o em uma sequência de instruções executáveis por uma máquina. A estrutura da gramática não é apenas uma representação formal das regras de uma linguagem de programação; é também um reflexo das decisões arquitetônicas que garantem a eficácia e a eficiência do processo de compilação. Nos próximos pontos, cada seção da gramática será explorada em detalhes, fornecendo uma compreensão abrangente de como o analisador sintático interpreta e processa o código-fonte Forth.

- **Prog :** Esta é a estrutura mais geral do sistema. Aqui é que é alocado necessário para as variáveis que vão ser usadas no programa.
- **Frase :** atua como um núcleo recursivo para o parser, proporcionando a flexibilidade de definir múltiplas expressões
- **ExpRel :** desempenha um papel essencial na estruturação do código Forth. Esta regra atua como uma entidade intermediária que direciona o parser para interpretar que tipo de expressão está à sua frente.
- **AtribDecl :** serve para haver a declaração de variáveis. Aqui é verificado os casos em que variáveis já foram anteriormente declaradas.
- **Atrib :** ser para interpretar a atribuição de valores para variáveis. Também aqui é verificado se a variável que se está atribuir já foi anteriormente declarada.
- **Exp :** esta regra serve principalmente para abranger os casos de contas aritméticas simples mas também possuí os finaliadores.
- **ExpStr :** esta regra serve exclusivamente para englobar os casos do token *CHAR*, houve necessidade de tantos casos específicos para ocorrer a correta interpretação por parte do parse.

- **FuncDef :** Semelhante à *AtribDecl* aqui também serve para a declaração das funções. Também com a verificação de re-declarações.
- **LoopDef :** Esta regra é a encarregue de lidar com os ciclos, sendo que ela também faz uma verificação a ver se o limite superior não é menor que o limite inferior.

3.2.2 Gramática

Prog : Frase

Frase : Frase ExpRel

|

| FuncDef

ExpRel : Exp

| PRINT

| EMIT

| SPACE

| CR

| KEY

| DUP

| AtribDecl

| Atrib

| ExpStr

| PRINT STRING

| LoopDef

AtribDecl : VARIABLE ID

Atrib : Exp ID '!'

Exp : Exp Exp OPAD

| Exp Exp OPSUB

| Exp Exp OPMUL

| Exp Exp OPDIV

| Exp Exp OPMOD

| NUM

| ID

| Exp Exp

ExpStr : CHAR ID

| CHAR '!'

| CHAR OPAD

| CHAR OPSUB

| CHAR OPMUL

| CHAR OPDIV

| CHAR OPMOD

| CHAR STARTDEF

| CHAR FINISHDEF

FuncDef : STARTDEF ID Frase FINISHDEF

| FuncDef FuncDef

LoopDef : Exp Exp DO Frase LOOP

3.2.3 Controlo de Erros

Na implementação do lexer e do yacc para análise léxica e sintática do código fonte, foram sendo adotadas algumas estratégias para detecção e tratamento de erros.

- **Erro de Caráteres Ilegais :** Na análise léxica temos a presença do t_error que deteta qualquer sequencia de caracteres que não correspondem a nenhuma Expressão regular dos tokens.
- *Erro de na identificação do token :* Na manipulação de erros sintáticos temos o p_error que é invocado quando um token inesperado é encontrado.
- *Erro de Divisão por Zero :* Durante operações de divisão, se o divisor for zero, o analisador gera um erro específico para evitar a divisão por zero.
- **Declaração de Variáveis e Funções :** Se uma variável ou função for declarada mais de uma vez ou utilizada sem declaração, o analisador gera erros.
- **Chamada de Função ou Variável Não Definida :** Quando uma função ou variável

é chamada mas não foi definida anteriormente e não se encontra no dicionário.

- **Início de Loop inválido :** Caso o valor inicial de iterações seja maior que o valor final dá um erro no loop e avisa o utilizador sobre isso.

4 Testes

Aqui temos alguns testes a ver o programa a funcionar.

```
3 2 4 + *
Parsing teminou com sucesso!
pushn 0
start
pushi 3
pushi 2
pushi 4
ADD
MUL
stop
```

```
3 0 /
Erro: Divisão por zero.
```

```
VARIABLE a
3 a !
a 3 +
Parsing teminou com sucesso!
pushn 1
start
pushi 3
storeg 0
pushg 0
pushi 3
ADD
stop
```

```
VARIABLE a
VARIABLE b
3 a !
5 4 + b !
a b +
Parsing teminou com sucesso!
pushn 2
start
pushi 3
storeg 0
pushi 5
pushi 4
ADD
storeg 1
pushg 0
pushg 1
ADD
stop
```

```
VARIABLE a
VARIABLE b
VARIABLE c
3 2 + a !
6 4 - b !
a b + c !
Parsing teminou com sucesso!
pushn 3
start
pushi 3
pushi 2
ADD
storeg 0
pushi 6
pushi 4
SUB
storeg 1
pushg 0
pushg 1
ADD
storeg 2
stop
```

```
." ola tudo bem?"
Parsing teminou com sucesso!
pushn 0
start
pushs " ola tudo bem?"
WRITES
stop
```

```
3 2 + .
Parsing teminou com sucesso!
pushn 0
start
pushi 3
pushi 2
ADD
WRITEI
stop
```

```
65 EMIT
Parsing teminou com sucesso!
pushn 0
start
pushi 65
WRITECHR
stop
```

```
CHAR W
Parsing teminou com sucesso!
pushn 0
start
pushs "W"
CHRCODE
stop
```

```
: OLA 3 2 + ;
OLA
Parsing teminou com sucesso!
pushn 1
start
pushi 3
pushi 2
ADD
stop
```

```
: TOFU ." Yummy bean curd!" ;
: SPROUTS ." Miniature veg" ;
: MENU CR TOFU CR SPROUTS CR ;
MENU
Parsing teminou com sucesso!
pushn 3
start
WRITELN
pushs " Yummy bean curd!"
WRITES
WRITELN
pushs " Miniature veg"
WRITES
WRITELN
stop
```

5 Conclusão

Após a conclusão deste projeto, foi possível alcançar uma compreensão mais profunda dos conceitos fundamentais de compiladores e linguagens de programação. O desenvolvimento do compilador para a linguagem Forth proporcionou uma oportunidade única para explorar aspectos práticos da análise léxica e sintática, bem como para compreender a complexidade envolvida na tradução de código-fonte em uma linguagem de alto nível para instruções executáveis na máquina virtual.

Durante o processo de concepção e implementação do compilador, foram enfrentados desafios significativos, como a definição precisa da gramática da linguagem Forth, a gestão eficiente de variáveis e funções, e a detecção e tratamento de erros durante a análise do código-fonte. Também sentimos bastante dificuladades em às vezes perceber como deveria ficar o código da máquina virtual.

Em conclusão, sentimos que alcançamos um bom nível de parser do código *FORTH* mas que ainda haveria espaço para mais melhoramento.