Universidade do Minho
Licenciatura em Engenharia Informática

Projeto PL

Forth Compiler

Grupo 52
Hugo Abelheira - a95151
João Carvalho - a94015

12 de maio de 2024

Conteúdo

Lista de Figuras

1 Introdução

Nesta secção introdutória, apresentaremos uma visão geral do projeto, detalhando o enunciado proposto e a descrição do problema a ser resolvido.

1.1 Enunciado

No âmbito da cadeira de Processamento de Linguagens, foi nos proposto o desenvolvimento de um compilador de Forth capaz de gerar código para a máquina virtual EWVM, também desenvolvida como parte desta unidade curricular.

1.2 Descrição do Problema

Desenvolver um compilador é uma tarefa desafiadora que requer a integração de diversas etapas e a resolução de diversos problemas. Neste projeto específico, enfrentamos também o desafio de integrar duas linguagens distintas: Forth, conhecida pela simplicidade e flexibilidade baseada em stack, e a linguagem da máquina virtual EWVM, que representa uma arquitetura alternativa de um processador.

Para resolver este desafio, recorremos à ferramenta Ply do Python, que oferece os módulos lex e yacc para construir analisadores léxicos e sintáticos. Esses analisadores são essenciais para interpretar e transformar o código Forth em instruções compreensíveis pela máquina virtual EWVM.

Ao longo deste relatório, detalharemos a arquitetura do sistema que desenvolvemos, explicaremos o processo de implementação e discutiremos os desafios enfrentados durante o desenvolvimento do compilador Forth para EWVM.

1.3 Forth

Forth é uma linguagem de programação única, conhecida pela sua abordagem minimalista e baseada em *stack*. Em Forth, os programas são construídos a partir de palavras (*words*), que são unidades de código que manipulam dados na *stack*. A sua sintaxe simples e flexível torna a linguagem altamente legível e fácil de entender.
Um exemplo simples de uma função em Forth para calcular a soma de dois números é:

- : soma1 1 + ;

Exemplo de um condicional em Forth para verificar se um número é maior que zero:

- 42 0 > IF ."Positivo"ELSE ."Não-positivo"THEN

Exemplo de um loop em Forth para imprimir os números de 1 a 10:

- VARIABLE ID 10 2 DO ID @ 2 + ID ! LOOP ID @ .

1.4 EWVM

A máquina virtual desenvolvida no âmbito desta UC, conhecida como EWVM, representa uma peça fundamental no contexto do projeto. Projetada como uma alternativa ao *assembly*, a EWVM oferece uma abstração de mais alto nível para a execução de código, simplificando o processo de programação e permitindo uma maior portabilidade entre plataformas.

Com um conjunto abrangente de operações, a EWVM capacita os programadores a realizar uma ampla variedade de tarefas, desde operações aritméticas simples até manipulações avançadas de dados. A sua interface intuitiva e uniforme torna o desenvolvimento de programas mais eficiente e acessível, contribuindo significativamente para o sucesso do projeto.

2 Arquitetura do Sistema

Nesta secção delineamos a arquitetura adotada para o desenvolvimento do compilador de Forth, projetado para gerar código para a EWVM. A abordagem foi estruturada em várias etapas distintas, cada uma desempenhando um papel crucial na realização do objetivo final.

Na primeira fase do projeto, concentramo-nos na definição de uma gramática inicial que capturasse a essência da linguagem Forth. Essa gramática serviu como um ponto de partida essencial, permitindo-nos esboçar uma estrutura inicial para o compilador e identificar os principais elementos da linguagem a serem considerados.

Em seguida, passamos para a implementação da análise léxica, utilizando a ferramenta Ply do Python para criar um *tokenizer* capaz de reconhecer os tokens necessários para a linguagem Forth. Durante esse processo, refinamos os *tokens* inicialmente identificados e ajustamos a estrutura do *tokenizer* para garantir uma análise precisa do código fonte.

Posteriormente, avançamos para a análise sintática e semântica, onde desenvolvemos um parser capaz de interpretar a estrutura gramatical do código Forth e realizar verificações semânticas para garantir a sua correção e consistência. Este foi um estágio crucial, onde enfrentamos desafios significativos, incluindo a resolução de conflitos de shift-reduce e reduce-reduce, além de ajustes na gramática para lidar com casos complexos.

Por fim, implementamos a geração de código, onde traduzimos as estruturas sintáticas reconhecidas pelo parser em instruções específicas da EWVM. Este processo envolveu a tradução cuidadosa das construções Forth para as operações suportadas pela máquina virtual, garantindo a eficiência e correção do código gerado.

3 IO

O compilador lê o código Forth de um arquivo de texto específico (*input.txt*), garantindo uma entrada estruturada e eficiente para o processo de compilação. Em seguida, o *tokenizer* entra em ação, identificando e classificando os *tokens* do código, fornecendo *insights* valiosos para fins de *debug*.

Quanto ao *output*, o compilador direciona o resultado da compilação para um arquivo de texto específico, contendo o código EWVM gerado. Essa abordagem permite uma separação clara entre os estágios de análise e geração de código, facilitando a manutenção e o entendimento do processo de compilação.

4 Implementação

Na secção de Implementação, fornecemos uma visão detalhada da metodologia adotada durante o desenvolvimento do compilador Forth para a EWVM. Este capítulo apresenta uma análise das decisões arquiteturais, os desafios enfrentados durante a implementação e as soluções encontradas para garantir o funcionamento eficiente do compilador.

Além disso, destacamos as principais etapas do processo de desenvolvimento, desde a definição da gramática inicial até a geração de código para a máquina virtual alvo. Este texto visa proporcionar uma compreensão clara das estratégias utilizadas para transformar o código Forth em instruções executáveis pela EWVM.

4.1 Gramática Inicial

Inicialmente, estabelecemos um ponto de partida para a construção do compilador Forth. Esta gramática serve como uma representação simplificada e esquemática da linguagem, destinada a esclarecer a estrutura básica do problema e a antecipar possíveis desafios na implementação do compilador. É importante ressaltar que esta gramática não é uma tentativa exaustiva de capturar todas as nuances da linguagem Forth, mas sim uma abordagem inicial para delinear a sua sintaxe fundamental:

OP = +,-,*,/,*/,%,‸
Potencia = DUP * = ^
S -> Comandos Funcoes $
Funcoes -> Funcao Funcoes
Funcoes -> &
Comandos -> Comando Comandos
Comandos -> &
Comando -> Expressao
Comando -> Variavel
Comando -> Imprime
Comando -> Statement
Comando -> Store
Comando -> UnStore
Comando -> Loop
Comando -> ID
Comando -> Comentario
Expressao -> Expressao Fator +
Expressao -> Expressao Fator -
Expressao -> Expressao Termo *
Expressao -> Expressao Termo /
Expressao -> Expressao Termo MOD
Expressao -> Termo
Termo -> Numero
Termo -> Unstore
Imprime -> Expressao '.'
Imprime -> ."Texto "Comandos
Imprime -> ASCII EMIT Comandos
Imprime -> &
Statement -> Expressao IF Executar ELSE Executar THEN Executar
Statement -> Expressao IF Executar THEN Executar
Executar -> Comandos
Variavel -> VARIABLE ID
Store -> Fator ID !
UnStore -> ID @
Funcao -> : ID Comandos;
Loop -> Expressao Expressao 'DO' 'ID' Comandos 'LOOP'
Comentario -> (Texto)
Comentario -> Texto

A gramática define regras para expressões, comandos, funções e outras construções típicas do Forth, como loops, condicionais, armazenamento e impressão de valores. Além disso, introduzimos elementos como variáveis, comentários e operações de manipulação de pilha. No entanto, é crucial reconhecer que esta gramática foi expandida e bastante otimizada à medida que avançamos no desenvolvimento do compilador, incorporando detalhes adicionais, lidando com casos mais complexos à medida que surgem e, obviamente, resolver alguns conflitos que esta gramática levanta.

Por meio desta abordagem, preparamos terreno para a análise léxica e sintática do código Forth, estabelecendo uma base sólida para a construção do compilador. Ao seguir esta estrutura inicial, podemos explorar e compreender melhor os requisitos e desafios específicos associados à geração de código para a EWVM.

4.2 Análise Léxica

A análise léxica desempenha um papel fundamental na interpretação do código fonte, sendo responsável por identificar e classificar os diferentes elementos presentes na linguagem. Ela consiste na segmentação do código em unidades léxicas, ou *tokens*, como números, operadores, identificadores e palavras-chave. O processo ocorre em paralelo à leitura do código fonte e é geralmente a primeira etapa do processo de compilação ou interpretação.

No contexto do nosso projeto, implementamos um analisador léxico utilizando a ferramenta Ply-Lex do Python. Inicialmente, identificamos os *tokens* essenciais para representar a sintaxe da linguagem Forth, como números (inteiros e reais), operadores aritméticos, palavras-chave (IF, THEN, ELSE, DO, LOOP) e outros símbolos relevantes. Durante o desenvolvimento, ajustamos a nossa abordagem para garantir que os *tokens* fossem simples e específicos, facilitando a análise sintática posterior. É importante mencionar que as primeiras versões do nosso *lexer* continha *tokens* bastante complexos, associando demasiada sintáxe a estes elementos. Um exemplo básico disso é por exemplo o *token VARIABLE*. Antes de decidirmos que iamos capturar a palavra que seria seguida por um *token ID*, estavamos a capturar na expressão regular da variável o próprio ID, dificultando o tratamento sintático posterior.

Além disso, utilizamos estados adicionais para lidar com casos específicos, como comentários e definições de funções. O analisador léxico gera uma lista de *tokens* que é fundamental para a análise sintática e semântica subsequente.

Lista final de tokens:

- **FUNC_START**: Indica o início da definição de uma função.
- **FUNC_END**: Indica o fim da definição de uma função.
- **NUMBER**: Representa números inteiros.
- **NUMBERF**: Representa números em ponto flutuante.
- **VARIABLE**: Representa a palavra-chave VARIABLE.
- **ID**: Representa identificadores, como nomes de variáveis e funções.
- **IF**: Representa a palavra-chave *IF*.
- **THEN**: Representa a palavra-chave *THEN*.
- **ELSE**: Representa a palavra-chave *ELSE*.
- **DO**: Representa a palavra-chave *DO*.
- **LOOP**: Representa a palavra-chave *LOOP*.
- **EMIT**: Representa a palavra-chave *EMIT*.
- **DOT**: Representa o operador de ponto '.'.
- **SUPEQ**: Representa o operador de comparação maior ou igual (>=).
- **INFEQ**: Representa o operador de comparação menor ou igual (<=).
- **DOTSTRING**: Representa o comando para imprimir *strings*.
- **STRING**: Representa uma string delimitada por aspas duplas '"'.
- **COMMENT_START**: Indica o início de um comentário *multiline* delimitado por parênteses.
- **COMMENT_END**: Indica o fim de um comentário *multiline* delimitado por parênteses.
- **ONELINE**: Representa um comentário de uma única linha, começando com '\'.
- **MLOOP**: Representa a palavra-chave *+LOOP*.
- **ROT**: Representa a função pré-definida *ROT*.
- **DROP**: Representa a função pré-definida *DROP*.
- **DUP**: Representa a função pré-definida *DUP*.
- **SWAP**: Representa a função pré-definida *SWAP*.
- **MOD**: Representa a operação resto da divisão.

4.3 Análise Sintática e Semântica

Nesta fase do projeto, desenvolvemos simultâneamente a análise sintática e semântica do código fonte, etapas crucias para compreender a estrutura e o significado dos comandos escritos em Forth.

Durante essa análise, deparamo-nos com desafios no *tokenizer*, como expressões regulares capturando casos inadequados e a presença de *tokens* desnecessários. Esses contratempos exigiram uma revisão meticulosa da gramática do *lexer* para garantir uma análise precisa e confiável do código.

Além disso, enfrentamos dificuldades ao implementar recursos específicos do Forth, como a *stack*, devido às suas características exclusivas. Após observamos que esta abordagem iria levantar um problema bastante complexo, decidimos optar por outra solução, pois nem todos os valores podiam ser guardados na nossa *stack*, por exemplo no caso dos condicionais, que só queremos guardar os valores ou do bloco ELSE ou do bloco THEN, algo que não conseguimos adaptar. A necessidade de alinhar a estrutura gramatical do código com o funcionamento intrínseco da linguagem Forth exigiu ajustes significativos na gramática e na lógica do compilador.

Durante a análise sintática, também lidamos com conflitos do tipo *reduce-reduce* e *shift-reduce*. Utilizamos técnicas de resolução de conflitos, como reestruturação das regras de produção e priorização de ações, para mitigar esses problemas e garantir a correta interpretação do código. Para clarificar, um conflito *shift-reduce* acontece quando o parser pode reduzir ou transitar no mesmo estado e um conflito *reduce-reduce* é quando a gramática contém ambiguidade e o parser pode reduzir duas regras diferentes no mesmo estado.

O uso do parser LALR1 gerado pelo Ply Yacc foi fundamental para o processo de *debug* e progresso do projeto. Ele proporcionou nos informação valiosa sobre a estrutura do código fonte e facilitou a identificação e correção de erros.

No desenvolvimento das regras de produção e na definição da gramática final, buscamos refletir com precisão a complexidade e a flexibilidade da linguagem Forth. O resultado foi uma gramática robusta que captura a essência da linguagem e orienta a análise sintática e semântica de forma eficaz e precisa.

Conseguimos capturar imensos aspetos da linguagem Forth, como operações aritméticas, facilitado pela característica do Forth ser escrito na notação *post-fix*, eliminando a necessidade de aplicar uma prioridade de operações, a criação de variáveis (e o seu mapeamento numa tabela), condicionais (juntamente com algumas variações destes), loops (incluindo a funcionalidade de +LOOP) e, claro, comentários (tanto *multiline* como *singleline*). As funções por sua vez foram um desafio maior, pois o Forth permite que as funções aceitem um número variado de parâmetros, bem como é possível serem retornados vários valores. Ainda assim, tentamos capturar as funções para termos uma base sólida de um compilador de Forth, mas ainda há bastante para otimizar em relação a este aspeto. Algumas funções pré-definidas também foram implementadas, mas estas não representam um desafio de complexidade, apenas seria necessário mais tempo para definir o comportamento de mais métodos. É também de mencionar que, provavelmente o maior problema do nosso trabalho, também devido à falta de tempo, é o tratamento de erros. Na realidade o nosso compilador permite a geração de código mal escrito, desde que simplesmente não exista erros de sintaxe.

4.4 Geração de Código

A geração de código é uma fase crucial do projeto, na qual transformamos a estrutura abstrata do código Forth em instruções concretas para a máquina virtual EWVM. Durante esta etapa, enfrentamos diversos desafios relacionados à tradução precisa dos comandos Forth para as operações suportadas pela máquina virtual.

Para realizar a geração de código de forma eficiente, mapeamos cada comando Forth para as instruções correspondentes da EWVM, levando em consideração as operações básicas suportadas pela máquina virtual, como operações aritméticas, controle de fluxo e manipulação de dados. Além disso, garantimos que a geração de código seja otimizada para maximizar a eficiência e o desempenho do programa final.

O resultado final é um compilador Forth capaz de traduzir comandos Forth em código EWVM de forma precisa e eficiente, garantindo a correta execução dos programas gerados na máquina virtual.

5 Gramática Final

```
Comandos -> Comandos Comando
          | &

Comando -> Expressao
        | Imprime
        | Comment
        | Store
        | Variavel
        | Conditional
        | Loop
        | Func
        | FuncP

Expressao -> Expressao '+'
          | Expressao '-'
          | Expressao '*'
          | Expressao '/'
          | Expressao '>'
          | Expressao SUPEQ
          | Expressao '<'
          | Expressao INFEQ
          | Expressao '='
          | Expressao MOD
          | Termo

Termo -> NUMBER
      | Unstore

Imprime -> DOT
        | DOTSTRING
        | EMIT

Comment -> COMMENT_START COMMENT_END
        | ONELINE

Store -> ID '!'

Unstore -> ID '@'

Variavel -> VARIABLE ID

Conditional -> Expressao IF Comandos THEN
             | Expressao IF Comandos ELSE Comandos THEN

Loop -> Expressao DO Comandos LOOP
     | Expressao DO Comandos Expressao MLOOP

Func -> FUNC_START ID Comandos FUNC_END
     | ID

FuncP -> ROT  | DROP | DUP | SWAP
```

```
S -> Comandos
T = {+,-,*,/,SUPEQ,<,>,=,INFEQ,NUMBER,
DOT,DOTSTRING,EMIT,COMMENT_START,
COMMENT_END,ONELINE,ID,!,@,VARIABLE,
IF,THEN,ELSE,FUNC_START,FUNC_END,LOOP,
MLOOP,ROT,DROP,DUP,SWAP}

N = {Comandos, Comando, Expressao,
Imprime, Comment, Store, Unstore,
Variavel, Conditional, Loop, Func,
FuncP, Termo}
```

6 User Guide

Esta secção detalha os comandos essenciais para utilizar o sistema desenvolvido, além de fornecer informações sobre os requisitos necessários para sua execução.

Além disso, serão apresentados exemplos de comandos e estruturas da linguagem Forth, juntamente com explicações sobre sua utilização. Esses exemplos visam auxiliar os usuários na compreensão da sintaxe da linguagem e no desenvolvimento de programas mais complexos.

Ao seguir as orientações desta secção, os usuários estarão aptos a utilizar o sistema de forma eficaz e desenvolver programas Forth para a máquina virtual EWVM com facilidade e precisão.

6.1 Como utilizar o sistema

Para começar, é necessário ter instaladas as ferramentas Python e Ply. Após a instalação, o usuário deve criar um arquivo de texto chamado "input.txt"contendo o código Forth a ser compilado, se este já existir apenas será necessário escrever o código Forth para este ser compilado. Este arquivo deve ser salvo no mesmo diretório que o compilador Forth está localizado. Em seguida, o usuário pode executar o compilador Forth, que lerá o código do arquivo "input.txt"e gerará o código da máquina virtual EWVM.

Para correr o programa, apenas será necessário entrar na pasta
src que se encontra dentro da pasta principal e nessa diretoría será necessário correr o ficheiro *yacc.py* através do *prompt* `python3 yacc.py` (ou a respetiva versão do python instalada na máquina do utilizador).

Após isso o código da EWVM gerado estará no ficheiro *output.txt* e os tokens capturados no texto que foi lido encontra-se no ficheiro *tokenizer.txt*, para efeitos de debug. A ferramenta Ply gera ainda 2 ficheiros automaticamente, um deles serve para a construção do parser LALR(1) e o outro é o "*pretty print*"do próprio autómato.

6.2 Exemplos

6.2.1 Operações Aritméticas

Input: `2 3 + 10 - 5 * 2 MOD .`

Output:

```
pushn 0
start
pushi 2
pushi 3
add
pushi 10
sub
pushi 5
mul
pushi 2
mod
writei
stop
```

6.2.2 Manipulação de Variáveis

Input: `VARIABLE ID 10 ID ! ID @ .`

Output:

```
pushn 1
start
pushi 10
storeg 0
pushg 0
writei
stop
```

6.2.3 Loops e as suas variações

Input: `VARIABLE ID 10 2 DO ID @ 2 + ID ! LOOP ID @ .`

Output:

```
pushn 3
start
pushi 10
storeg 1
pushi 2
storeg 2
WHILE0:
pushg 1
pushg 2
sup
jz ENDWHILE0
pushg 0
pushi 2
add
storeg 0
pushi 1
pushg 2
add
storeg 2
jump WHILE0
ENDWHILE0:
pushg 0
writei
stop
```

Input: `VARIABLE ID 10 2 DO ID @ 2 + ID ! 2 3 + +LOOP ID @ .`

Output:

```
pushn 3
start
pushi 10
storeg 1
pushi 2
storeg 2
WHILE0:
pushg 1
pushg 2
sup
jz ENDWHILE0
pushg 0
pushi 2
add
storeg 0
pushi 2
pushi 3
add
pushg 2
add
storeg 2
jump WHILE0
ENDWHILE0:
pushg 0
writei
stop
```

6.2.4 Condicionais

Input: `1 1 = IF ."EQUAL"ELSE ."DIFFERENT"THEN ."DONE"`

Output:

```
pushn 0
start
pushi 1
pushi 1
equal
jz ELSE0
pushs " EQUAL"
writes

jump ENDIF0
ELSE0:
pushs " DIFFERENT"
writes

jump ENDIF0
ENDIF0:
pushs " DONE"
writes
stop
```

Input: `2 2 - IF ."EQUAL"THEN ."DONE"`

Output:

```
pushn 0
start
pushi 2
pushi 2
sub
jz ENDIF0
pushs " EQUAL"
writes

jump ENDIF0
ENDIF0:
pushs " DONE"
writes
stop
```

6.2.5 Funções

Input: `: FuncEx 1 + ; 10 FuncEx`

Output:

```
pushn 0
start
pushi 10
pusha FuncEx
call
stop
FuncEx:
pushl -1
pushi 1
add
storel -1
return
```

6.2.6 Funções pré-definidas

Input: `1 2 3 ROT .`

Output:

```
pushn 1
start
pushi 1
pushi 2
pushi 3
storeg 0
swap
pushg 0
swap
writei
stop
```

7 Conclusão

A conclusão deste projeto representa não apenas o término de um ciclo de desenvolvimento, mas também um ponto de partida para reflexões sobre engenharia de linguagens e programação generativa. Durante o processo de conceção e implementação do compilador Forth para a máquina virtual EWVM, enfrentamos diversos desafios que nos levaram a repensar e aprimorar continuamente nossas abordagens e estratégias.

Ao finalizar este projeto, pudemos compreender melhor os princípios fundamentais da engenharia de linguagens, explorando a complexidade envolvida na conceção e implementação de compiladores. A análise detalhada da sintaxe e semântica da linguagem Forth proporcionou nos *insights* valiosos sobre a estruturação e organização de linguagens de programação, bem como nos permitiu abordar os desafios de projetar sistemas que sejam eficientes e flexíveis.

Além disso, a experiência adquirida ao desenvolver um compilador que gera código para uma máquina virtual específica permitiu explorar os conceitos de programação generativa, nos quais o código é gerado automaticamente com base em determinadas regras e especificações. Com esta abordagem foi possível compreender melhor as vantagens e desvantagens da programação generativa, bem como sua aplicabilidade em diferentes contextos e domínios.

Em suma, este projeto proporcionou uma experiência enriquecedora e permitiu-nos aprofundar conhecimentos em engenharia de linguagens e programação generativa. Esperamos que as lições aprendidas ao longo deste processo possam ser aplicadas em projetos futuros e inspirem outros a explorar e inovar no campo da computação.