Centro Federal de Educação Tecnológica de Minas Gerais

Departamento de Computação

Curso de Engenharia de Computação

Josué Rocha Lima

Felipe Freitas

Pedro Henrique Silva

Trabalho Prático: implementação de um gerador de código

KPiler: compilador para a linguagem K

Relatório do trabalho prático apresentado à disciplina de compiladores.

Orientador: Kecia Aline Marques Ferreira

Belo Horizonte

2017

Sumário

1 Introdução

Com o intuito de reduzir a complexidade da implementação de sistemas de software, foi proposta a utilização de linguagens próximas da linguagem humana, denominadas linguagens de alto nível. Entretanto, para possibilitar a implementação de software dessa maneira, é necessário a tradução para uma linguagem que possa ser executada diretamente no *hardware* (AHO; SETHI; ULLMAN, 2007).

O compilador é um sistema de *software* responsável pela tradução automática de um programa em linguagem fonte de alto nível, para um programa semanticamente equivalente em uma linguagem alvo, conforme pode-se observar na Figura 1. Deste modo, os compiladores são considerados *software* básico para a computação.

Figura 1 – Tradução do programa fonte para a linguagem alvo realizada pelo compilador (AHO; SETHI; ULLMAN, 2007).

O compilador é comumente dividido em módulos para reduzir a complexidade do seu processo de implementação. Essa divisão é frequentemente compreendida pelos seguintes módulos:

- **Front-end (Análise)**
  - **Analisador léxico**: lê o programa fonte caractere a caractere e agrupa o programa em sequências significativas denominadas lexemas. Para cada um dos lexemas é gerado um *token* no formato <nome , valor>. O analisador léxico também é responsável por reconhecer e ignorar comentários e gerenciar numeração de linhas.
  - **Analisador sintático**: recebe o fluxo de *tokens* gerado pelo analisador sintático e verifica se a sequência está de acordo com a estrutura sintática da linguagem. É o núcleo do compilador.
  - **Analisador semântico**: Utiliza a tabela de símbolos e árvore sintática para verificar se o programa-fonte atende às regras semânticas da linguagem, como regras de compatibilidade de tipos.

  - **Gerador de código intermediário:** Código em representação intermediária em baixo nível é gerado para auxiliar a geração de código e otimização.

- **Back-end (Síntese)**
  - **Gerador de código:** mapeia a linguagem intermediária para a linguagem objeto.

A Figura 1 ilustra o fluxo entre os módulos do compilador:

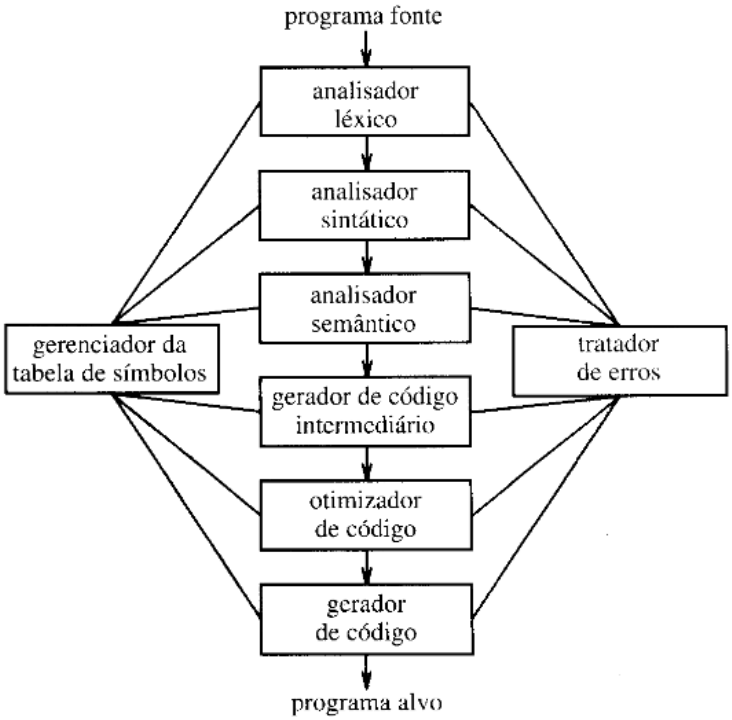

Figura 2 – Divisão em módulos do compilador (AHO; SETHI; ULLMAN, 2007).

1.1 Proposta de trabalho prático

Foi proposto o projeto e implementação de um compilador para uma linguagem hipotética *K* como trabalho prático e didático para a disciplina de Compiladores. Devido à complexidade do problema de compilação, o projeto foi dividido em quatro componentes com suas respectivas entregas:

1. Analisador léxico e tabela de símbolos;
2. Analisador sintático;
3. Analisador semântico;

4. Gerador de código.

O trabalho entregue anexo à este relatório corresponde ao gerador de código (3). A geração de código é a etapa final do processo de compilação, que consiste em traduzir o programa-fonte, livre de erros, para um programa semanticamente equivalente na linguagem objeto, que é comumente uma linguagem de máquina. Durante esse processo, o gerador de código utiliza as informações da árvore de sintaxe e da tabela de símbolos ou uma representação em código intermediário para executar a tradução.

Após as análises sintática e semântica, muitos compiladores geram uma representação intermediária em código de baixo nível, que pode ser pensado como um programa para uma máquina abstrata (AHO; SETHI; ULLMAN, 2007). Essa representação é realizada com o intuito de se evitar que, dado n arquiteturas alvo, seja necessário construir n compiladores diferentes.

Quando a linguagem alvo é uma linguagem de máquina, são selecionados registradores e endereços de memória para cada variável do programa. Deste modo, é necessário conhecimento detalhado acerca da arquitetura alvo para que as escolhas apropriadas possam ser realizadas, seja durante a geração de código intermediário ou a geração do código na linguagem alvo.

Existem requerimentos severos para a semântica e a qualidade do código gerado. Entretanto, o problema de gerar código ótimo dado um programa-fonte é matematicamente não-decidível (AHO; SETHI; ULLMAN, 2007). Deste modo, a melhor alternativa é a utilização de heurísticas, que permitem gerar código de qualidade não necessariamente ótimo.

O gerador de código possui três tarefas principais: (1) seleção de instruções, (2) ordenação das instruções, (3) alocação de registradores e endereços de memória. A seleção das instruções consiste em combinar instruções de baixo nível para compor as instruções de alto nível do programa-fonte. Por outro lado, a ordenação das instruções consiste na decisão da sequência de execução necessária para produzir o resultado desejado. Finalmente, a alocação de registradores e endereços de memória consiste na seleção de locais de armazenamento para cada variável do programa atendendo as especificações da arquitetura alvo (AHO; SETHI; ULLMAN, 2007).

O conjunto de instruções da máquina alvo influem diretamente no processo de compilação. Deste modo, as arquiteturas alvo mais comuns são máquinas *Complex Instruction Set Computer* (CISC), *Reduced Instruction Set Computer* (RISC) e arquiteturas baseadas em pilha (AHO; SETHI; ULLMAN, 2007).

As máquinas RISC possuem instruções simples, com tipos de endereçamento simples, instruções de três endereços e muitos registradores. Por outro lado, máquinas CISC possuem poucos registradores, instruções e endereçamento complexos e instruções

com efeitos colaterais. Em máquinas baseadas em pilha, os operados são armazenados em uma estrutura de dados do tipo pilha e as operações são aplicadas sobre os elementos no topo da pilha (AHO; SETHI; ULLMAN, 2007).

No contexto deste trabalho, gerar código em linguagem de montagem reduz a complexidade do problema, pois pode-se utilizar as funções de macros e as instruções simbólicas das linguagens de montagem evitando-se preocupar com os problemas das linguagens de máquina absolutas. Entretanto, existe um prejuízo para o tempo de compilação decorrente da necessidade da utilização do montador para gerar código de máquina (AHO; SETHI; ULLMAN, 2007).

Este trabalho está divido em quatro capítulos adicionais. O Capítulo 2 trata da metodologia adotada na construção do gerador de código. O Capítulo 3 apresenta orientações e instruções para utilização do gerador de código. O Capítulo 4 apresenta os resultados dos códigos gerados e sua comparação com os resultados desejados. Finalmente, o Capítulo 5 apresenta as conclusões deste trabalho.

2 Metodologia

A metodologia adotada neste trabalho consiste em gerar código em linguagem de montagem para a máquina virtual VM a partir de programas-fonte fornecidos como objeto de pesquisa e análise desta etapa de trabalho, sendo um desenvolvido pelos alunos para testes. Os códigos foram submetidos a análise léxica, sintática e semântica já implementados no compilador da linguagem *K* nas etapas antecedentes e agora utilizados para a geração dos seus respectivos códigos em linguagem alvo como teste desta implementação utilizando a abordagem.

2.1 Máquina virtual VM

A máquina virtual VM é uma máquina de pilhas, que é composta de duas pilhas de execução, uma pilha de chamadas de procedimentos, uma zona de código, duas *heaps* e quatro registradores.

A pilha de execução contem os valores que podem ser inteiros, reais ou endereços. As *heaps* armazenam *strings* e blocos (structs) respectivamente. As *strings* e blocos são acessíveis por meio de ponteiros armazenados na pilha de execução.

O registrador sp (*stack pointer*) aponta para o topo da pilha, fp (*frame pointer*) aponta para o endereço base do registro de ativação corrente, gp (*global pointer*) aponta para o endereço base das variáveis globais e o pc (*program counter*) aponta para a instrução corrente.

Maiores detalhes estão disponíveis em <http://www.lri.fr/~mandel/enseignement/projet-compilation/> e na documentação em português em <http://epl.di.uminho.pt/~gepl/LP/vmdocpt.pdf>. O código fonte foi descarregado e compilado para o ambiente Linux para possibilitar a utilização neste trabalho.

2.2 Projeto do gerador de código

O projeto do gerador de código foi iniciado pelo estudo de cada produção da gramática da linguagem *K*, visando identificar o significado semântico de cada uma das instruções da linguagem e suas variações.

Deste modo, percebeu-se que os operadores relacionais eram gerados na produção *relop*, os operadores aritméticos nas produções *mulop* e *addop*, o acesso à variáveis e constantes nas produções *factor* e *constant* e a composição de expressões nas produções *expression*, *simple-expression* e *term*.

Inicialmente, a geração do código para áreas em que não eram necessários desvios de fluxo de execução foram implementadas, devido à sua simplicidade. As operações básicas foram tratadas nas produções *mulop* e *addop*, gerando as instruções aritméticas ADD, SUB, DIV e MUL. A composição das expressões foi tratada nas produções *expression*, *simple-expression* e *term*. Cumpre observar que o tipo à que o operador '+' se ligava foi analisado, para possibilitar decidir se seria gerado uma instrução de concatenação ou de soma, de acordo com os tipos *string* e *int*, respectivamente.

Posteriormente, na produção *relop* foram geradas as instruções de desvio JUMP e JZ e instruções relacionais EQUAL, NOT, SUP, SUPEQ, INF e INFEQ necessárias para tratar as operações relacionais de igual, diferente, maior ou igual, menor ou igual, menor e maior.

Ademais, na produção *mulop* e *addop* foram tratadas o remendo dos rótulos das instruções JUMP e JZ gerados previamente na produção *relop* para tratar o *OR* lógico e o *AND* lógico. Foi utilizado o algorítmo *backpatching* abordado durante as aulas. As demais ações de tradução podem ser observadas na gramática anotada abaixo:

⟨*program_prime*⟩ ::= ⟨*Program*⟩ '$' (1)

⟨*Program*⟩ ::= 'program' ⟨*decllist*⟩ ⟨*stmt-list*⟩ 'end' (2)

⟨*decllist*⟩ ::= ⟨*decl-list*⟩ (3)
| 'λ' (4)

⟨*decl-list*⟩ ::= ⟨*decl*⟩ ⟨*decllist*⟩ (5)

⟨*decl*⟩ ::= ⟨*type*⟩ ⟨*identifier-list*⟩ {gen(PUSHN 1)} (6)

⟨*identifier-list*⟩ ::= ⟨*identifier*⟩ ⟨*possibleident*⟩ { } (7)

⟨*possibleident*$_1$⟩ ::= ',' ⟨*identifier*⟩ (8) ⟨*possibleident*⟩ (8)
| 'λ' (9)

⟨*type*⟩ ::= ⟨*int*⟩ (10)
| ⟨*string*⟩ (11)

⟨*stmt-list*⟩ ::= ⟨*stmt*⟩ ⟨*stmtlist*⟩ (12)

⟨*stmtlist*⟩ ::= ⟨*stmt-list*⟩ (13)
| 'λ' (14)

⟨*stmt*⟩ ::= ⟨*assign-stmt*⟩ ; (15)
| ⟨*if-stmt*⟩ (16)
| ⟨*while-stmt*⟩ (17)

| ⟨*read-stmt*⟩ ';' (18)

| ⟨*write-stmt*⟩ ';' (19)

⟨*assign-stmt*⟩ ::= ⟨*identifier*⟩ '=' ⟨*simple-expr*⟩ { gen(STOREL identifier.address) } (20)

⟨*if-stmt*⟩ ::= 'if' ⟨*condition*⟩ 'then' M ⟨*stmt-list*⟩ ⟨*if-stmt-prime*⟩ (21)
{ backpatch(condition.truelist,M.inst);
if-stmt-prime.fl = condition.fl }

⟨*if-stmt-prime*⟩ ::= 'end' (22)

| N 'else' {inst = gen(JUMP)}⟨*stmt-list*⟩⟨*end*⟩{*backpatch*(*if*−*stmt*−*prime.falselist*, (23)*N.inst*); *backpatc*

⟨*condition*⟩ ::= ⟨*expression*⟩ (24)

⟨*while-stmt*⟩ ::= 'do' M ⟨*stmt-list*⟩ ⟨*stmt-sufix*⟩ (25)

⟨*stmt-sufix*⟩ ::= 'while' ⟨*condition*⟩ 'end' N {backpatch(condition.tl,M.inst); (26)
backpatch(condition.fl,N.inst) }

⟨*read-stmt*⟩ ::= 'scan' '(' ⟨*identifier*⟩ ')' (27)
{ }

⟨*write-stmt*⟩ ::= ⟨*print*⟩ '(' ⟨*writable*⟩ ')' (28)

⟨*writable*⟩ ::= ⟨*simple-expr*⟩ (29)

| ⟨*literal*⟩ (30)

⟨*expression*⟩ ::= ⟨*simple-expr*⟩ ⟨*expression-prime*⟩ (31)
{ }

⟨*expression-prime*⟩ ::= ⟨*relop*⟩ ⟨*simple-expr*⟩ (32) (32)

| 'λ' (33)

⟨*simple-expr*⟩ ::= ⟨*term*⟩ ⟨*simple-expr-prime*⟩ (34)
{ }

⟨*simple-expr-prime*⟩ ::= ⟨*addop*⟩ ⟨*term*⟩ ⟨*simple-expr-prime*⟩ (35)
{ }

| 'λ' (36)

⟨*term*⟩ ::= ⟨*factor-a*⟩ ⟨*term-prime*⟩ (37)

⟨*term-prime*⟩ ::= ⟨*mulop*⟩ M ⟨*factor-a*⟩ ⟨*term-prime{1}*⟩ (38)
{ backpatch(term-prime.truelist,M.inst)
term-prime.truelist = term-prime1.truelist term-prime.falselist = merge(factor-a.falselist,term-prime.falselist) }

| 'λ' (39)

⟨*factor-a*⟩ ::= ⟨*factor*⟩ { factor-a = factor.attributes} (40)
| '!' ⟨*factor*⟩ {factor-a.fl=factor.tl (41)
factor-a.tl=factor.fl}
| '-' { gen(PUSHI 0)} ⟨*factor*⟩ { gen(SUB)} (42)

⟨*factor*⟩ ::= ⟨*identifier*⟩ { gen(PUSHL identifier.address) } (43)
| ⟨*constant*⟩ { factor = constant.attributes} (44)
| '(' ⟨*expression*⟩ ')' { factor = expression.attributes } (45)

⟨*constant*⟩ ::= integer-constant { gen(PUSHI constant.val)} (46)
| literal { gen(PUSHS literal.val) } (47)

⟨*relop*⟩ ::= '==' { gen(EQUAL) } (48)
| '>' { gen(SUP) } (49)
| '>=' { gen(SUPEQ) } (50)
| '<' { gen(INF) } (51)
| '<=' { gen(INFEQ) } (52)
| '!=' { gen(EQUAL); gen(NOT) } (53)

⟨*addop*⟩ ::= '+' { gen(MUL) } (54)
| '-' { gen(SUB) } (55)
| '||' (56)

⟨*mulop*⟩ ::= '*' { gen('MUL') } (57)
| '/' { gen('DIV') } (58)
| '&&' (59)

2.3 Implementação do gerador de código

Devido a natureza incremental deste trabalho, optou-se pela utilização da linguagem de programação Java para permitir incrementar as etapas anteriores. As instruções correspondentes ao gerador de código foram definidas em uma classe em separado. Entretanto, as funções do gerador de código são referenciadas na classe correspondente ao analisador sintático. Não foi utilizada nenhuma biblioteca externa à linguagem Java.

2.3.1 Arquitetura da implementação

O diagrama de classe gerado no trabalho anterior(Figura 3) foi atualizado, visto que as classes *Instruction*, *Attribute* e *CodeGenerator* foram criadas e a classe *Parser* foi alterada. A arquitetura do KPiler baseou-se nas diretrizes e recomendações de (ANDREW; JENS, 2002).

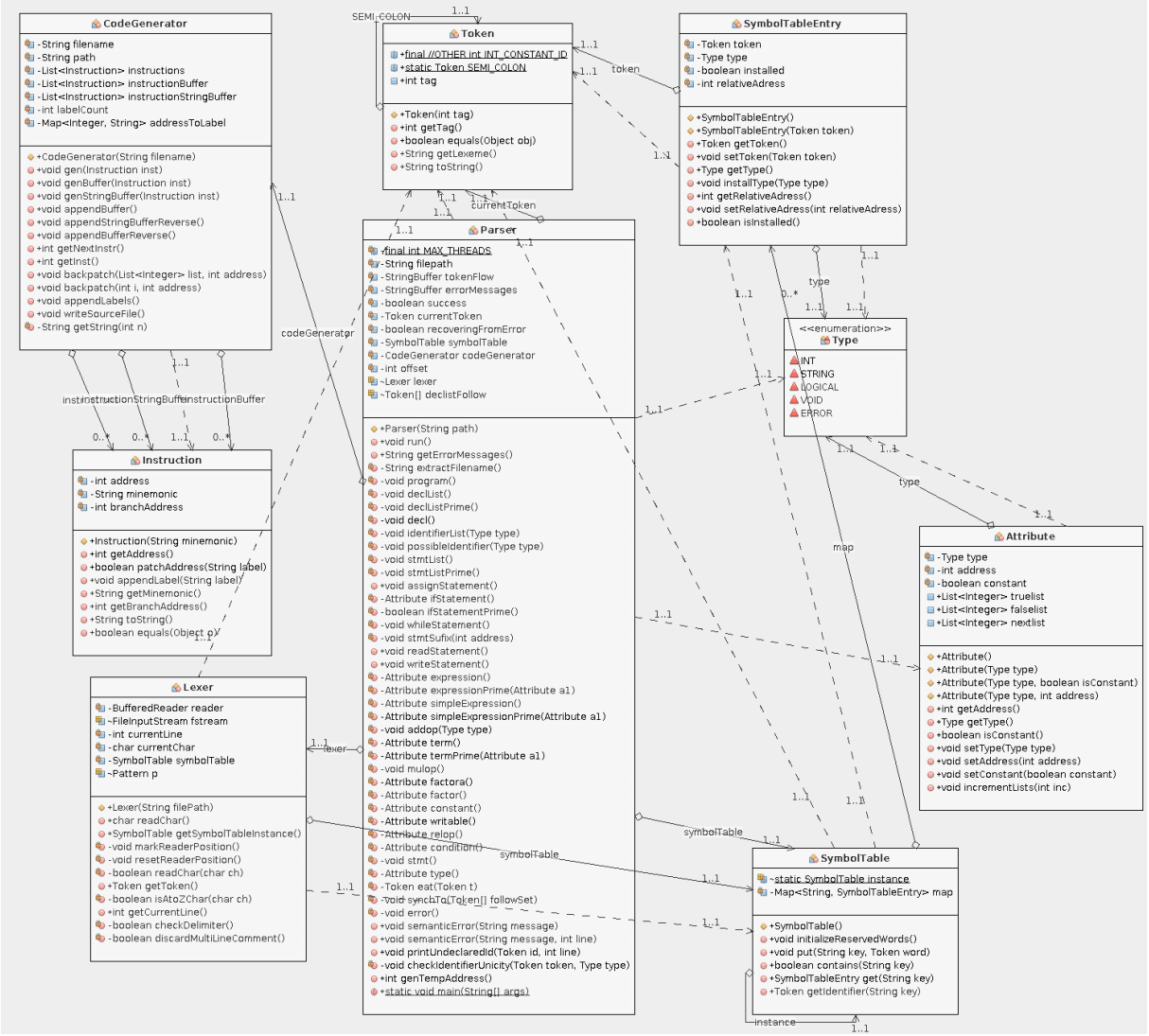

Figura 3 – Diagrama de classe do compilador, as classes utilitárias foram omitidas para facilitar o entendimento.

As principais modificações realizadas em cada uma das classes e as novas classes criadas estão detalhadas a seguir:

- **Parser (analisador sintático):** Em cada método correspondente à um símbolo não-terminal da gramática, foi adicionado código respectivo às ações semânticas especificadas previamente. As produções retornam um objeto da classe *Attribute*. As funções da classe *CodeGenerator* são referênciadas nos métodos da classe *Parser*.

- **Attribute:** foi criada para permitir compartilhamento de maior quantidade de atributos entre as produções do analisador sintático. Deste modo, a classe *Attribute* possui o campo *Type* e os campos adicionais *truelist*, *falselist*, *nextlist* e *address*.

- **CodeGenerator (gerador de código)**: armazena uma lista de instruções, possui

métodos para adição de instruções, backpatch de instruções e gravação do código em linguagem objeto em arquivo. Alguns métodos especiais para adição de instruções em *buffers* foram criadas para possibilitar posterior inversão de sua sequência. Isso foi feito pois a instrução CONCAT da máquina VM funciona com uma ordem inversa às instruções aritméticas da arquitetura.

- **Instruction:** essa classe foi criada para conter o minemônico das instruções geradas. O vetor de instruções na classe *CodeGenerator* é formado de objetos da classe *Instruction*. Esta classe também possui funções para auxiliar na realização do *backpatch* dos endereços de desvio.

2.3.2 Principais desafios

Foi necessária a utilização de um *buffer* para possibilitar a inversão da ordem dos operandos da instrução CONCAT. Isso ocorreu pois a instrução CONCAT atua sobre os operandos em uma ordem inversa às instruções aritméticas.

Além disso, percebeu-se que alguns atributos eram perdidos ao longo das produções sequenciais quando λ era produzido. Deste modo, o caso λ foi tratado. Cumpre observar que todas as dificuldades aqui relatadas foram resolvidas.

3 Instruções de utilização

O projeto está preparado para receber o caminho dos arquivos de código-fonte via linha de comando (terminal do Linux ou *prompt* de comandos do Windows). A passagem dos argumentos pode ser configurada no ambiente de desenvolvimento de preferência ou informado na inicialização do arquivo executável .jar ou na execução direta dos arquivos .class. Exemplo de execução:

- **Execução do .jar:** o seguinte comando deve ser executado na mesma pasta do arquivo .jar caso as configurações do projeto anexado sejam mantidas:

java -jar KPiler.jar test/test1.k test/test2.k

- **Execução dos arquivos .class:** deve-se navegar para a pasta do *build* e executar os seguintes comandos para compilar e executar o código:

javac Parser.java

java Parser test/test1.k test/test2.k

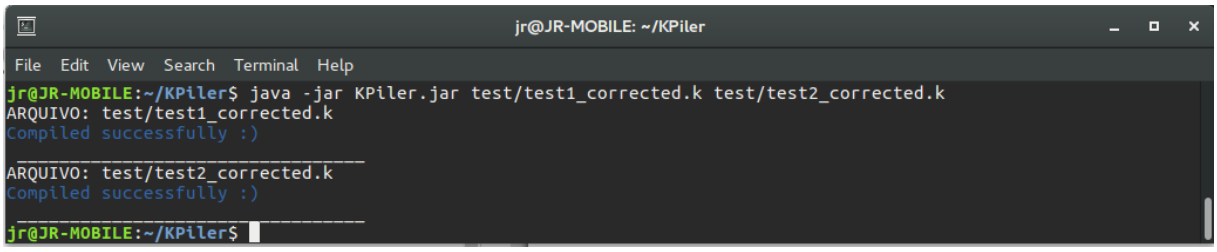

Figura 4 – Execução do programa via terminal.

Após a compilação do programa-fonte, é necessário executar os programas objeto na máquina VM. Os programas objeto são gerados e armazenados na pasta *output* na raiz do projeto. Para executar os programas na plataforma Linux, o seguinte comando deve ser informado:

./vm test1.asm

O executável compilado para Linux pelos autores desse trabalho também está disponível na pasta *output*.

4 Avaliação dos resultados

O programa foi submetido a seis testes de compilação. Cinco dos testes foram elaborados pela orientadora deste trabalho e um teste foi elaborado pelo autor. Os arquivos de teste estão disponibilizados em anexo no diretório *test* disponível neste projeto e em sua página do GitHub[1]. A página do GitHub se encontra e modo privado até a conclusão da disciplina.

Os programas-fonte foram submetidos ao compilador, o programa em linguagem objeto foi executado e resultado de execução foi relatado. Cumpre observar que os programas-fonte livres de erros léxicos, sintáticos e semânticos das etapas anteriores foram considerados nesta etapa. Portanto, os códigos em linguagem-fonte estão disponíveis na pasta *test* e os programas-objeto estão disponíveis na pasta *output*. Ambas as pastas estão localizadas na raiz deste projeto.

4.1 Teste 1

Código-fonte:

```
1  program
2     int a, b;
3     int result;
4     int x,y,total;
5
6     a = 2;
7     x = 1;
8     scan (b);
9     scan (y);
10    result = (a*b + 1) / 2;
11    print ("Resultado: ");
12    print (result);
13    print ("Total: ");
14    total = y / x;
15    print ("Total: ");
16    print (total);
17 end
```

[1] <https://github.com/josuerocha/KPiler>

Código objeto gerado:

```
START
PUSHN 2
PUSHN 1
PUSHN 3
PUSHI 2
STOREL 0
PUSHI 1
STOREL 3
READ
ATOI
STOREL 1
READ
ATOI
STOREL 4
PUSHL 0
PUSHL 1
MUL
PUSHI 1
ADD
PUSHI 2
DIV
STOREL 2
PUSHS "Resultado: "
WRITES
PUSHL 2
WRITEI
PUSHS "Total: "
WRITES
PUSHL 4
PUSHL 3
DIV
STOREL 5
PUSHS "Total: "
WRITES
PUSHL 5
WRITEI
STOP
```

Para a execução do programa, foram utilizados os parâmetros $a = 4$ e $y = 4$.

Resultado de execução:

```
1 VM version 1.7
2 read =>
3 4
4 read =>
5 4
6 Resultado: 4Total: Total: 4
```

O programa produziu as saídas $resultado = 4$ e $total = 4$ durante esta execução. O resultado correspondeu ao esperado pois a expressão $result = (a * b + 1)/2$ foi avaliada em $result = (2 * 4 + 1)/2$, cujo resultado é 4 devido ao truncamento da mantissa efetuada pela VM. De maneira similar, a expressão $total = y/x$ foi avaliada em $total = 4/1$, resultando em 4. Este resultado correspondeu ao esperado.

4.2 Teste 2

Código-fonte:

```
1  program
2     int a, b, c;
3     int d, e, val;
4     a = 0; d = 35;
5     c = d / 12;
6     scan (a);
7     scan (c);
8     b = a * a;
9     c = b + a * (1 + a*c);
10    print ("Resultado: ");
11    print (c);
12    a = (b + c + d)/2;
13    e = val + c + a;
14    print ("E: ");
15    print (e);
16 end
```

Código objeto:

```
1 START
2 PUSHN 3
3 PUSHN 3
```

```
4  PUSHI 0
5  STOREL 0
6  PUSHI 35
7  STOREL 3
8  PUSHL 3
9  PUSHI 12
10 DIV
11 STOREL 2
12 READ
13 ATOI
14 STOREL 0
15 READ
16 ATOI
17 STOREL 2
18 PUSHL 0
19 PUSHL 0
20 MUL
21 STOREL 1
22 PUSHL 1
23 PUSHL 0
24 PUSHI 1
25 PUSHL 0
26 PUSHL 2
27 MUL
28 ADD
29 MUL
30 ADD
31 STOREL 2
32 PUSHS "Resultado: "
33 WRITES
34 PUSHL 2
35 WRITEI
36 PUSHL 1
37 PUSHL 2
38 PUSHL 3
39 ADD
40 ADD
41 PUSHI 2
42 DIV
43 STOREL 0
44 PUSHL 5
45 PUSHL 2
```

```
46 PUSHL 0
47 ADD
48 ADD
49 STOREL 4
50 PUSHS "E: "
51 WRITES
52 PUSHL 4
53 WRITEI
54 STOP
```

O programa foi executado com as entradas $a = 2$ e $c = 4$.

Resultado de execução

```
1 VM version 1.7
2 read =>
3 2
4 read =>
5 4
6 Resultado: 22E: 52
```

É possível observar que a expressão $c = d/12$ foi avaliada em 2 devido ao truncamento da mantissa do número real. A expressão $b = a*a$ foi avaliada em 4, $c = b+a*(1+a*c)$ foi avaliada em $c = 4+2*(1+2*4) = 22$ e $a = (b+c+d)/2 = (4+22+35)/2 = 30$ devido ao truncamento da mantissa. Finalmente, $e = val + c + a$ foi avaliada em $e = 0 + 22 + 30 = 52$. Percebe-se que o teste atendeu aos resultados esperados.

4.3 Teste 3

Código-fonte:

```
1  program
2     int pontuacao, pontuacaoMaxima;
3     string  disponibilidade;
4     int pontuacaoMinima;
5     disponibilidade = "Sim";
6     pontuacaoMinima = 50;
7     pontuacaoMaxima = 100;
8     /* Entrada de dados
9     Verifica aprovação de candidatos */
10    do
11       print("Pontuacao Candidato: ");
12       scan(pontuacao);
13       print("Disponibilidade Candidato: ");
14       scan(disponibilidade);
15
16       if ((pontuacao > pontuacaoMinima) && (disponibilidade=="Sim")) then
17          print("Candidato aprovado");
18       else
19          print("Candidato reprovado");
20       end
21    while (pontuacao >= 0) end
22 end
```

Código objeto:

```
1  START
2  PUSHN 2
3  PUSHN 1
4  PUSHN 1
5  PUSHS "Sim"
6  STOREL 2
7  PUSHI 50
8  STOREL 3
9  PUSHI 100
10 STOREL 1
11 E: PUSHS "Pontuacao Candidato: "
12 WRITES
13 READ
14 ATOI
```

```
15 STOREL 0
16 PUSHS "Disponibilidade Candidato: "
17 WRITES
18 READ
19 STOREL 2
20 PUSHL 0
21 PUSHL 3
22 SUP
23 NOT
24 JZ A
25 JUMP D
26 A: PUSHL 2
27 PUSHS "Sim"
28 EQUAL
29 NOT
30 JZ B
31 JUMP D
32 B: PUSHS "Candidato aprovado"
33 WRITES
34 JUMP C
35 D: PUSHS "Candidato reprovado"
36 WRITES
37 C: PUSHL 0
38 PUSHI 0
39 SUPEQ
40 NOT
41 JZ E
42 JUMP F
43 F: STOP
```

O programa foi executado quatro vezes com as entradas (1)$pontuacao = 50$ e $disponibilidade = Sim$, (2)$pontuacao = 51$ e $disponibilidade = Sim$, (3)$pontuacao = 51$ e $disponibilidade = sim$ e (4)$pontuacao = -1$ e $disponibilidade = Nao$.

Resultado de execução:

```
1 VM version 1.7
2 Pontuacao Candidato: read =>
3 50
4 Disponibilidade Candidato: read =>
5 Sim
6 Candidato reprovadoPontuacao Candidato: read =>
7 51
```

```
8  Disponibilidade Candidato: read =>
9  Sim
10 Candidato aprovadoPontuacao Candidato: read =>
11 51
12 Disponibilidade Candidato: read =>
13 sim
14 Candidato reprovadoPontuacao Candidato: read =>
15 -1
16 Disponibilidade Candidato: read =>
17 Nao
18 Candidato reprovado
```

O programa atendeu aos resultados esperados, visto que os requisitos para aprovação são: pontuação acima de 50 e disponibilidade igual à "Sim". Portanto, no caso 1 o resultado foi "Candidato reprovado"devido à pontuação não ser maior que 50. No caso 2, o candidato foi aprovado pois tinha pontuação maior que 50 e disponibilidade "Sim". Entretanto, no caso 3 o resultado foi "Candidato reprovado"pois o "sim"informado possui o 's' minúsculo, diferentemente do valor de referência. Finalmente o candidato foi reprovado devido à pontuação -1 inferior à 50 e a disponibilidade "Nao"e o programa também foi encerrado devido ao resultado falso na condição do laço *while* decorrente do valor negativo da variavel pontuacao.

4.4 Teste 4

Código-fonte:

```
1  program
2     int a, aux, b;
3     string nome, sobrenome, msg;
4     print("Nome: ");
5     scan (nome);
6     print("Sobrenome: ");
7     scan (sobrenome);
8     msg = "Ola, " + nome + " " +
9     sobrenome + "!";
10    msg = msg + "1";
11    print (msg);
12    scan (a);
13    scan(b);
14    if (a>b) then
15       aux = b;
```

```
16        b = a;
17        a = aux;
18     end
19     print ("Apos a troca: ");
20     print(a);
21     print(b);
22  end
```

Código objeto:

```
1  START
2  PUSHN 3
3  PUSHN 3
4  PUSHS "Nome: "
5  WRITES
6  READ
7  STOREL 3
8  PUSHS "Sobrenome: "
9  WRITES
10 READ
11 STOREL 4
12 PUSHS "!"
13 PUSHL 4
14 PUSHS " "
15 PUSHL 3
16 PUSHS "Ola, "
17 CONCAT
18 CONCAT
19 CONCAT
20 CONCAT
21 STOREL 5
22 PUSHS "1"
23 PUSHL 5
24 CONCAT
25 STOREL 5
26 PUSHL 5
27 WRITES
28 READ
29 ATOI
30 STOREL 0
31 READ
32 ATOI
```

```
33 STOREL 2
34 PUSHL 0
35 PUSHL 2
36 SUP
37 NOT
38 JZ A
39 JUMP B
40 A: PUSHL 2
41 STOREL 1
42 PUSHL 0
43 STOREL 2
44 PUSHL 1
45 STOREL 0
46 B: PUSHS "Apos a troca: "
47 WRITES
48 PUSHL 0
49 WRITEI
50 PUSHL 2
51 WRITEI
52 STOP
```

Foram realizadas duas sequencias de execução. Na primeira os parâmetros de entrada foram definidos em $nome =' Josue'$, $sobrenome =' Rocha'$, $a = 2$ e $b = 10$. Na segunda execução, os parâmetros foram definidos em: $nome =' Josue'$, $sobrenome =' Lima'$, $a = 10$ e $b = 2$

Resultado da execução 1:

```
1  VM version 1.7
2  Nome: read =>
3  Josue
4  Sobrenome: read =>
5  Rocha
6  Ola, Josue Rocha!1read =>
7  2
8  read =>
9  10
10 Apos a troca: 210
```

Resultado da execução 2:

```
1  VM version 1.7
2  Nome: read =>
```

```
3  Josue
4  Sobrenome: read =>
5  Lima
6  Ola, Josue Lima!1read =>
7  10
8  read =>
9  2
10 Apos a troca: 210
```

Em ambos os testes, a mensagem foi concatenada e exibida na ordem correta. Além disso, o maior valor sempre é mantido na variável b e impresso após o valor menor. Deste modo, o teste antendeu aos resultados desejados.

4.5 Teste 5

O código fonte inicial foi submetido ao compilador:

Código-fonte:

```
1  program
2     int a, b, c, maior, outro;
3
4     do
5        print("A");
6        scan(a);
7        print("B");
8        scan(b);
9        print("C");
10       scan(c);
11       //Realizacao do teste
12       if ( (a>b) && (a>c)) then
13          maior = a;
14       else
15       if (b>c) then
16          maior = b;
17       else
18          maior = c;
19       end
20       end
21       print("Maior valor:");
22       print (maior);
23       print ("Outro? ");
```

```
24        scan(outro);
25     while (outro >= 0) end
26  end
```

Código objeto:

```
1   START
2   PUSHN 5
3   G: PUSHS "A"
4   WRITES
5   READ
6   ATOI
7   STOREL 0
8   PUSHS "B"
9   WRITES
10  READ
11  ATOI
12  STOREL 1
13  PUSHS "C"
14  WRITES
15  READ
16  ATOI
17  STOREL 2
18  PUSHL 0
19  PUSHL 1
20  SUP
21  NOT
22  JZ A
23  JUMP F
24  A: PUSHL 0
25  PUSHL 2
26  SUP
27  NOT
28  JZ B
29  JUMP F
30  B: PUSHL 0
31  STOREL 3
32  JUMP D
33  F: PUSHL 1
34  PUSHL 2
35  SUP
36  NOT
```

```
37  JZ C
38  JUMP E
39  C: PUSHL 1
40  STOREL 3
41  JUMP D
42  E: PUSHL 2
43  STOREL 3
44  D: PUSHS "Maior valor:"
45  WRITES
46  PUSHL 3
47  WRITEI
48  PUSHS "Outro? "
49  WRITES
50  READ
51  ATOI
52  STOREL 4
53  PUSHL 4
54  PUSHI 0
55  SUPEQ
56  NOT
57  JZ G
58  JUMP H
59  H: STOP
```

O programa foi executado três vezes com os seguintes parâmetros: (1) $a = 1$,$b = -1$ e $c = -5$; (2) $a = 0$,$b = 1000$ e $c = 5$; e (3) $a = -1$,$b = -1$ e $c = 100$. Por fim a execução foi encerrada informando o valor $outro = -1$ Os resultados de execução são exibidos abaixo:

Resultado de execução:

```
1  VM version 1.7
2  Aread =>
3  1
4  Bread =>
5  -1
6  Cread =>
7  -5
8  Maior valor:1Outro? read =>
9  1
10 Aread =>
11 0
12 Bread =>
13 1000
```

```
Cread =>
5
Maior valor:1000Outro? read =>
1
Aread =>
-1
Bread =>
-1
Cread =>
100
Maior valor:100Outro? read =>
-1
```

A execução atendeu aos resultados esperados, pois o maior número entre a, b e c foi retornado em todas as execuções. Além disso, a execução parou ao informar um valor menor que 0 para o parâmetro *outro*, conforme desejado.

4.6 Teste 6

Código-fonte:

```
/*THIS IS A MULTIPLE LINE COMMENT
 Josu  Rocha Lima */

program
   //this is an unclosed string literal
   string str;
   int i;
   int maxtam;

   maxtam = 20;
   str = "HELLO WORLD";
   i = 0;


   do
      print(i);
      i = i + 1;
   while i < maxtam end

end
```

Código objeto:

```
1  START
2  PUSHN 1
3  PUSHN 1
4  PUSHN 1
5  PUSHI 20
6  STOREL 2
7  PUSHS "HELLO WORLD"
8  STOREL 0
9  PUSHI 0
10 STOREL 1
11 A: PUSHL 1
12 WRITEI
13 PUSHL 1
14 PUSHI 1
15 ADD
16 STOREL 1
17 PUSHL 1
18 PUSHL 2
19 INF
20 NOT
21 JZ A
22 JUMP B
23 B: STOP
```

Resultado de execução:

```
1  VM version 1.7
2  01234567891011121314151617181 9
```

O programa exibiu uma sequência de números de 0 a 19, atendendo aos resultados esperados.

5 Conclusão

Neste trabalho relatou-se a experiência dos autores durante a implementação do gerador de código, a etapa final do compilador.

Durante a execução do compilador, desde a etapa léxica ao sintático, foi possível perceber a função e a importância de cada etapa para o processo como um todo, pois a divisão em partes do processo, culminando na geração de código, permitiu abordar o problema em diferentes níveis de detalhe (AHO; SETHI; ULLMAN, 2007).

Além disso, notou-se a necessidade de conhecer os aspectos da arquitetura alvo para possibilitar a aplicação da instrução de baixo nível mais adequada às construções da linguagem fonte.

Todos os resultados dos testes do analisador semântico do *KPiler* atenderam aos resultados desejados, quanto à precisão dos resultados e tempo de execução.

Referências

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: principles, techniques, and tools.* [S.l.]: Addison-wesley Reading, 2007. v. 2.

ANDREW, W. A.; JENS, P. *Modern compiler implementation in Java.* [S.l.]: Cambridge, 2002.