Arthur Manuel Bandeira
Carlos Henrique Paisca
Marcos Felipe Pereira Vieira

Relatório Técnico
Implementação de um Compilador para a Linguagem CMM

Maringá - PR - Brasil

Julho de 2017

Arthur Manuel Bandeira
Carlos Henrique Paisca
Marcos Felipe Pereira Vieira

Relatório Técnico
Implementação de um Compilador para a Linguagem CMM

Relatório Técnico da Implementação de um Compilador para a Linguagem CMM – Trabalho da disciplina Implementação de Linguagens de Programação

Universidade Estadual de Maringá – UEM

Departamento de Informática – DIN

Ciência da Computação

Maringá - PR - Brasil

Julho de 2017

Sumário

Introdução

Este relatório técnico tem como objetivo detalhar o desenvolvimento do trabalho de implementação de um compilador para a linguagem CMM. Esta nomenclatura é um acrônimo para C Mais-ou-Menos, uma linguagem mais ou menos parecida com C.

Apresenta-se uma breve fundamentação teórica sobre as fases de um compilador, as decisões de projeto e modo de implementação das fases que foram realizadas. Este projeto foi desenvolvido com uso da linguagem de programação Python e a biblioteca *PLY (Python Lex-Yacc)*, que oferece uma implementação direta e simplificada de ferramentas clássicas no desenvolvimento de compiladores, *lex* e *yacc*.

No capítulo 1 são apresentadas as fases um compilador, seguindo a classificação definida por Aho, Sethi e Ullman (1986). Seguindo esta subdivisão, primeiramente descreve-se as fases de análise: léxica na seção 1.2, sintática na seção 1.3 e semântica na seção 1.4 e posteriormente as fases de síntese, geração de código intermediário contida na seção 1.5, otimização de código na seção 1.6 e por fim, geração de Código presente na seção 1.7.

O capítulo 2 introduz o conceito de seleção de instruções, apresentando sua utilidade e a fase onde ocorre no processo de compilação, os métodos de seleção *Maximal Munch* na seção 2.1 e Programação Dinâmica contido na seção 2.2. Estes conceitos foram baseados em Appel e Palsberg (2003).

No capítulo 3 é apresentado o conceito de alocação de registradores, e assim como no capítulo anterior, sua utilidade e em que fase é empregada no processo de compilação. Ademais, descreve-se o processo de análise de longevidade de variáveis na seção 3.1, o processo de coloração de grafos na seção 3.2 e um algoritmo para alocação de registradores na seção 3.3.

Por fim, apresentam-se as considerações finais deste relatório e as referências utilizadas.

1 Fases de um Compilador

Conceitualmente, segundo a visão de Aho, Sethi e Ullman (1986) as fases de um compilador podem ser divididas em análise e síntese, que por sua vez, são subdivididas em outras fases, sendo a análise dividida em análise léxica, análise sintática e análise semântica e a síntese dividida em geração de código intermediário, otimização de código e geração de código. Essas fases traduzem o programa fonte através de cada fase gerando um programa na linguagem alvo. A figura 1 demonstra esse processo.

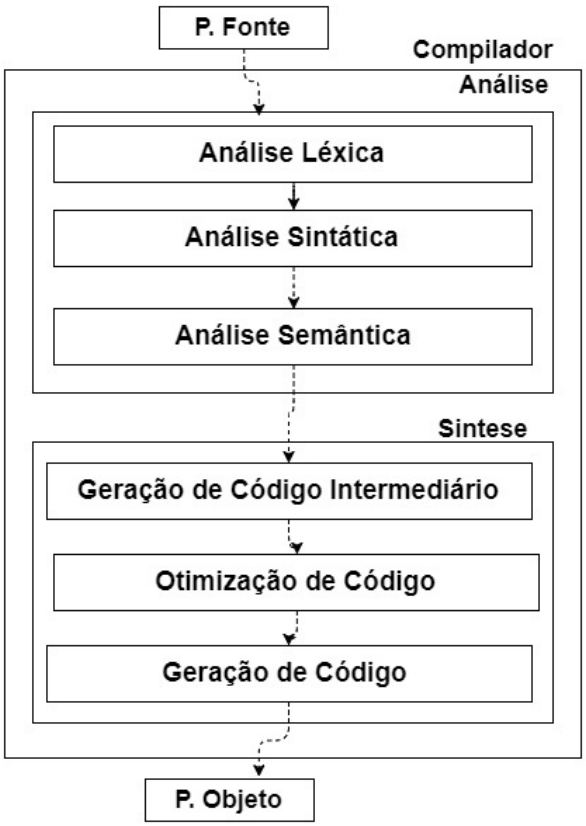


Figura 1 – Fases de um compilador.

1.1 Ferramentas utilizadas

Para o desenvolvimento da análise léxica e sintática, utilizamos a ferramenta `PLY (Python Lex-Yacc)` (BEAZLEY, 2001–2017) que é uma implementação de duas ferramentas de análise, sendo o `lex` um analisador léxico e `yacc` um analisador sintático. A escolha da ferramenta foi feita devido à algumas características essenciais da ferramenta como:

- Implementação toda em *python*.

- Utiliza análise sintática LR que é bastante eficiente e adequada a gramáticas de tamanho considerável.
- Fornece características comuns à implementações *lex/yacc* incluindo suporte à produções vazias, regras de precedência, recuperação de erros e gramáticas ambíguas.
- Seu uso e entendimento é simples e descomplicado, além de fornecer uma extensa verificação de erros.
- Fornece apenas as funcionalidades *lex/yacc*, não sendo dependente de um sistema maior ou de grande quantidade de módulos.

1.2 Análise Léxica

A análise léxica é a primeira fase de um processo de compilação, sua função consiste em fazer a leitura de uma programa fonte, passando por todos os seus caracteres e agrupando-os em lexemas e retornar uma sequência de símbolos denominados *tokens*. Destacamos que, essa sequência de *tokens* é enviada para ser processada pela análise sintática.

Para o desenvolvimento da análise léxica, utilizamos o módulo `lex.py` da ferramenta `PLY` que divide o texto de entrada em uma coleção de *tokens* baseando-se em expressões regulares. Palavras reservadas e *tokens* foram determinadas de acordo com a especificação da linguagem, como exibido nos trechos de código 1 e 2.

```
palavras_reservadas = {
    'bool': 'BOOL',
    'string': 'STRING',
    'int': 'INT',
    'for': 'FOR',
    'if': 'IF',
    'else': 'ELSE',
    'while': 'WHILE',
    'true': 'TRUE',
    'false': 'FALSE',
    'return': 'RETURN',
    'break': 'BREAK',
    'read': 'READ',
    'write': 'WRITE',
}
```

Trecho de código 1 – Palavras reservadas

```
tokens = [
    'ID', 'CADEIA', 'NUMBER',
    'ABREPAREN', 'FECHAPAREN',
    'ABRECOLCH', 'FECHACOLCH',
    'ABRECHAVE', 'FECHACHAVE',
    'VIRGULA', 'PONTOVIRGULA',
    'MAIS', 'MENOS', 'MULT', 'DIV', 'MOD',
    'IGUAL', 'DIFERENTE',
    'MAIOR', 'MAIORIGUAL', 'MENOR', 'MENORIGUAL',
    'OU', 'E', 'NEG',
    'ATRIB', 'MAISATRIB', 'MENOSATRIB',
    'MULTATRIB', 'DIVATRIB', 'MODATRIB', 'SINAL',
    'TERNARIOSE', 'TERNARIOSENAO',
] + list(palavras_reservadas.values())
```

Trecho de código 2 – *Tokens*

Para cada *token* especificado foi criada uma expressão regular de acordo com a biblioteca `re` da linguagem *python*, de modo que em cada regra foi definida com a declaração do prefixo `t_` indicando a definição de um *token*. Alguns *tokens* necessitam que a função da classe `lex` seja sobrescrita para atender as necessidades da gramática, como ocorre com os identificadores em `t_ID`, os trechos de código 3 e 4 demonstram a implementação.

```
t_MAIS = r'\+'
t_MENOS = r'-'
t_MULT = r'\*'
t_DIV = r'/'
t_MOD = r'%'
t_IGUAL = r'=='
```

Trecho de código 3 – Expressões regulares

```
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = palavras_reservadas.get(t.value,'ID')
    return t
```

Trecho de código 4 – Expressão regular `t_ID`

Durante a análise léxica, cada *token* se torna uma instância da classe `LexToken` e esta classe que possui os atributos `t.type`, `t.value`, `t.lineno` e `t.lexpos`.

O atributo `t.type` e armazena o tipo do *token*, por padrão, quem segue o prefixo `t_` na expressão regular que reconhece o *token*, `t.value` é o lexema em si, `t.lineno` é a linha em que o *token* se encontra e `t.lexpos` a coluna.

A função `t_error()` é usada para lidar com erros léxicos que ocorrem quando caracteres ilegais são detectados. Neste caso, os atributos `t.type`, `t.value`, `t.lineno` e `column` contém o tipo, valor, linha e coluna do *token* de entrada que não foi analisada. A função de erro é apresentada no trecho de código 5.

```
def t_error(t):
    column = find_column(t.lexer.lexdata, t)
    print('LexError(%s,%r,%d,%d)' % (t.type, t.value, t.lineno, column))
    t.lexer.skip(1)
```

Trecho de código 5 – Função `t_error`

1.3 Análise Sintática

A análise sintática recebe a sequência de *tokens* gerada pela análise léxica. Esta fase tem como papel principal determinar se o programa de entrada representado pela sequência de *tokens*, possui as sentenças válidas para a linguagem de programação definida. Na maioria dos casos utiliza-se gramáticas livres de contexto para especificar a construção de uma linguagem de programação.

A construção de gramáticas não são tarefas triviais e torna-se mais complexa quando se trata da construção de uma gramática sem ambiguidade. Assim, quando uma gramática ambígua é usada como entrada, a ferramenta `yacc` imprime mensagens como "mudar / reduzir conflitos". Isso ocorre quando o gerador do analisador não consegue decidir se deve realizar um *reduce* ou não em uma regra, ou se deve dar um *shift* em um símbolo na pilha de análise.

Por padrão, todos os conflitos de *shift/reduce* são tomados para *shift*. Embora essa estratégia da ferramenta funcione para a maioria dos casos, como o clássico *if-then-else*, a mesma não é suficiente para expressões aritméticas.

Na análise sintática utilizamos o módulo `yacc.py` que é aplicado para reconhecer uma sintaxe da linguagem que foi especificada na forma de uma gramática livre de contexto. A sintaxe da gramática é especificada na forma normal de Backus – BNF (*Backus Normal Form*), sendo esta uma meta sintaxe usada para expressar gramáticas livres de contexto de modo formal.

Tal especificação, é um conjunto de regras de derivação escritas como:

$$< simbolo > ::= < expressao\ com\ simbolos >$$

na qual $< simbolo >$ é um não terminal, e $< expressao\ com\ simbolos >$ é composta por sequências de símbolos e/ou sequências separadas pela barra vertical, | indicando uma escolha. Esta notação indica as possibilidades de substituição para símbolo da esquerda, já

os símbolos terminais sempre ficam do lado direito da regra. O trecho de código 6 retrata esta representação.

```
def p_var(p):
    '''
    var : ID
        | ID ABRECOLCH exp FECHACOLCH
    '''
```

Trecho de código 6 – Gramática na forma BNF

Cada regra da gramática é definido por uma função na qual o `docstring` desta contém a especificação da regra da gramática livre de contexto associada. O conteúdo da função implementam as ações semânticas da regra. Cada função, aceita apenas um argumento `p`, sendo este uma sequência contendo os valores de cada símbolo da gramática na regra correspondente. Iterando `p`, mapeia-se os símbolos da gramática, como mostrado no trecho de código 7.

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    #   ^            ^        ^    ^
    #  p[0]         p[1]     p[2] p[3]

    p[0] = p[1] + p[3]
```

Trecho de código 7 – Mapeamento dos valores da gramática

Para *tokens* o valor mapeado, `p[1]`, de `p` é equivalente ao `t.value` obtido pela análise léxica. Para não-terminais o valor é determinado por `p[0]` quando as regras são reduzidas, nesta implementação, cada não-terminal é uma classe que herda características de uma classe abstrata. Esta funcionalidade é demonstrada no trecho de código 8.

```
def p_var(p):
    '''
    var : ID
        | ID ABRECOLCH exp FECHACOLCH
    '''
    if len(p) == 2:
        p[0] = Variable(id_=p[1])
    else:
        p[0] = Variable(id_=p[1], exp=p[3])
```

Trecho de código 8 – Análise sintática para variáveis – criação da classe `Variable`

A ferramenta implementa uma função para tratar a produção vazia, denominada `p_empty`, esta por sua vez, quando executada apenas continua a execução. Esta função é descrita no trecho de código 9.

```
def p_empty(p):
    ''' empty : '''
    pass
```

Trecho de código 9 – Produção vazia

Para o tratamento de erros sintáticos, é definida a função `p_error` que, na ocorrência de erros sintáticos, retorna o *token* no qual o erro ocorreu além da posição de linha e coluna. O trecho de código 10 exemplifica esta função.

```
def p_error(p):
    last_cr = p.lexer.lexdata.rfind('\n', 0, p.lexer.lexpos)
    column = p.lexer.lexpos - last_cr - 1

    if p:
        print("Erro de sintaxe em {0} na linha {1} coluna {2}".format(
                p.value,
                p.lexer.lineno,
                column
        ))
    else:
        print("Erro de sintaxe EOF")
```

Trecho de código 10 – Função para tratamento de erros sintáticos

Os erros sintáticos que são acusados e param a execução do programa são:

- Nomes de função duplicados – caso sejam definidas mais de uma função para a mesma regra da gramática.
- Conflitos de *shift/reduce* gerados por gramáticas ambíguas.
- Regras da gramática especificadas de maneira incorreta.
- Recursão infinita causada por regras que nunca terminam.
- Regras ou tokens indefinidos ou inutilizados.

A análise sintática é executada para todo o programa de entrada e caso não ocorram erros sintáticos, a mesma retorna uma instância da classe `Program`, sendo esta a raiz da árvore sintática abstrata. Ao percorrer e imprimir a árvore sintática abstrata, obtêm-se um resultado na forma exibida na figura 2.

```
Program
DecSeq
Dec
VarDec
Type
('type', 'int')
VarSpecSeq
VarSpec
('id', 'a')
VarSpec
('id', 'b')
VarSpec
('id', 'c')
DecSeq
Dec
VarDec
Type
('type', 'bool')
VarSpecSeq
VarSpec
('id', 'd')
DecSeq
Dec
Type
('type', 'int')
('id', 'main')
('Variaveis declaradas', ['a', 'b', 'c', 'd'])
```

Figura 2 – Árvore sintática abstrata

1.4 Análise Semântica

A análise semântica é responsável por verificar aspectos relacionados ao significado das instruções. Esta análise utiliza uma árvore sintática e uma tabela de símbolos, com intuito de identificar regras, tais como: identificadores declarados antes de serem usados e análises de tipo. Isto torna-se fundamental para garantir que o programa fonte esteja coerente e possa ser convertido para linguagem de máquina.

1.5 Geração de Código Intermediário

A tradução de um código de alto nível para o código do processador está associada em traduzir para uma linguagem alvo a representação da árvore gramatical obtida nas fases anterior para as diversas expressões do programa. Esta atividade pode ser realizada para a árvore completa após a conclusão da arvore sintática, em geral é efetivada por meio das ações semânticas.

A geração de códigos não ocorre diretamente para a linguagem *assembly* do processador alvo. Por conveniência o analisador gera código para uma máquina abstrata próxima a *assembly*, contudo independente de processadores específicos. Tem-se ainda, o código intermediário que é traduzido para a linguagem *assembly* desejada.

Vale ressaltar que, existem formas usuais para este tipo de representação, tais como: notação posfixa e código de três endereços. Desta forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores.

1.6 Otimização de Código

O objetivo desta etapa é aplicar um conjunto de heurísticas para identificar sequências de código ineficiente e substituí-las por outras que removam estas situações de código ineficiente. Tais técnicas utilizadas devem, além de manter a semântica do programa original, ser capaz de capturar a maior parte de possibilidade de melhoria do código dentro dos limites de esforço gasto para tal fim.

1.7 Geração de Código

Esta é a fase final de um compilador, a geração de código alvo normalmente constituído de código de máquina realocável ou código de montagem. As memorias localizadas são selecionadas para cada uma das variáveis usadas pelo programa, sendo assim as instruções intermediarias são uma a uma traduzida em uma sequência de instrução de máquina que realizam a mesma tarefa. Vale destacar que, um aspecto crucial desta fase é a atribuição das variáveis aos registradores.

2 Seleção de Instruções

A árvore da representação intermediária expressa uma operação "simples"em cada nó, podendo ser o Acesso à Memória, Operador Binário ou Salto Condicional. O objetivo da **Seleção de Instruções** é encontrar um conjunto de instruções de máquina para geração de código baseado na representação intermediária, utilizando o menor número de padrões existentes para a máquina alvo. Uma arquitetura que pode ser utilizada para introduzir e exemplificar a seleção é a arquitetura *Jouette*, com as seguintes características:

- Arquitetura composta por *Load/Store*;
- Qualquer instrução pode acessar qualquer registrador;
- Registradores podem armazenar dado ou endereço;
- r0 sempre contém zero;
- Cada instrução gasta 1 ciclo;
- Executa apenas uma instrução por ciclo.

Seu objetivo é cobrir a árvore com padrões, sem sobreposição entre padrões como vemos nos exemplos apresentados nas Figuras 3 e 4.

Name	*Effect*	*Trees*
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j \quad r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST; + CONST; CONST
SUBI	$r_i \leftarrow r_j \quad c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST; MEM + CONST; MEM CONST; MEM

Figura 3 – Padrões de Árvore - *Jouette*

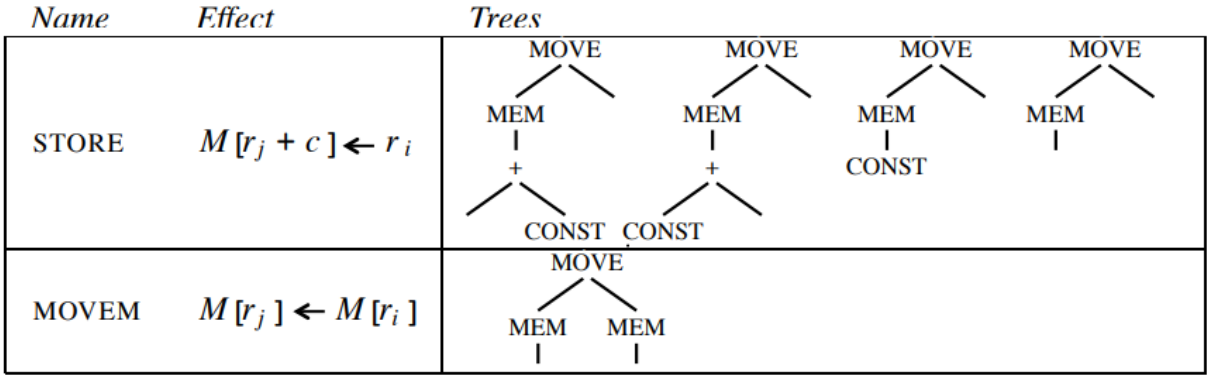


Figura 4 – Padrões de Árvore - *Jouette*

Caso as instruções tenham latências diferentes, é escolhida a de menor tempo total. Assim, cada instrução recebe um custo e a melhor cobertura da árvore é a que a soma dos custos dos padrões utilizados é a menor possível, caracterizando o *tiling Optimum*. O *tiling Optimal* caracteriza-se quando em uma cobertura nenhum par de padrões adjacentes possa ser combinado em um par de menor custo. Em caso de haver um padrão de seleção que possa diminuir o tempo total, porém possa ser quebrado, este padrão deve ser descartado.

É de grande importância que os algoritmos de seleção de instruções sejam eficientes para que possam trabalhar em qualquer arquitetura. Como por exemplo, as arquiteturas CISC e RISC.

A CISC (*Complex Instruction Set Computers*) possui instruções que realizam varias operações, desta forma os padrões para estas instruções são maiores e as diferenças entre o *tiling Optimal* e *Optimum* são notáveis. Enquanto na arquitetura RISC (*Reduced Instruction Set Computers*) - arquitetura utilizadas por máquinas mais modernas - o número de operações efetuado por instrução é menor (todas as instruções da arquitetura *Jouette* apresentadas nas figuras 3 e 4). Uma vez que a cobertura é pequena e de custo uniforme, não há muita diferença entre o *tiling Optimum* e *Optimal*, portanto o uso do mais simples é suficiente.

Vários algoritmos eficientes são encontrados para encontrar o *tiling Optimum* e *Optimum*, porém o algoritmo para encontrar o *tiling Optimal* é mais simples.

2.1 Método de Seleção *Maximal Munch*

Maximal Munch é o algoritmo utilizado para a cobertura *Optimal*. É um algoritmo simples que inicia da raiz para as folhas e encontra o maior padrão que se encaixe nesse nó, ou seja, que cubra a raíz e os outros nós. Para cada sub-árvore encontrada o processo é repetido. Desta forma, a cada padrão selecionado, uma instrução é gerada e a execução é efetuada na ordem inversa, ou seja, a raíz é a última a ser executada.

No algoritmo *Maximal Munch* o maior padrão é aquele com maior número de nós e caso haja dois padrões com mesmo tamanho, a escolha entre qualquer deles é arbitrária. Normalmente é um algoritmo de fácil implementação utilizando funções recursivas

2.2 Programação Dinâmica

Um algoritmo de programação dinâmica é uma técnica para construção de algoritmos visando encontrar uma solução ótima. É aplicável para encontrar soluções ótimas para um problema com base em soluções ótimas de cada subproblema. Para o problema de seleção de instruções os subproblemas são as *tilings* das subárvores.

O algoritmo atribui uma árvore a partir das folhas, ou seja, *botton-up*, atribuindo um custo para cada nó da árvore e este custo é a soma dos custos das instruções da melhor sequência de instruções que percorre a sua subárvore. Esse processo se repete para cada nó até chegar na raiz. A seguir trazemos a figura 5 da árvore de programação dinâmica como exemplo.

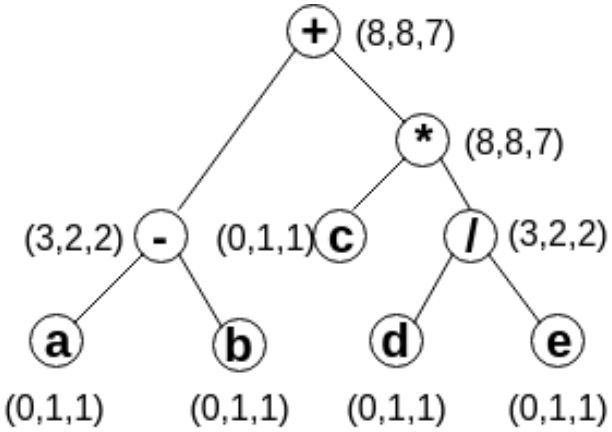


Figura 5 – Árvore Programação Dinâmica

Todos os nós da árvore apresentam um custo associado com 3 valores o que representa o custo das operações no nó utilizando (memoria, reg1, reg2), e de acordo com o algoritmo sempre buscamos as operações que reflitam em menores custos para o nó. Para isso definimos as operações bem como seus custos para realizar os cálculos, conforme mostra o trecho de código 11

```
1 LD Ri Mj    custo=1
2 Op Ri Ri Rj custo=1
3 Op Ri Ri Mj custo=1
4 LD Ri Rj    custo=1
5 ST Rj Mi    custo=1
```

Trecho de código 11 – Operações e custos

De posse das operações e os custos, basta realizarmos os cálculos de acordo com o nó e a operação e como temos o dado, se ele se encontra em memória ou registrador. O processo e o mesmo para todos os nó primeiro olhamos os valores das folhas que nesse caso é (0,1,1) usando a operação LD, o nó (-)(3,2,2) por exemplo e calculado pegando o custo da OP 2 ou 3 nesse caso pois possuem o mesmo custo, entre os nós (a) e (b) e realizamos a OP com memória, reg1 e re2 e atribuir os custos. Note que o custo na memoria é 3 pois o tenho o custo de LD para cada dado (a) e (b) mais o custo da OP.

3 Alocador de Registradores

A alocação de registradores esta na fase de otimização de código no desenvolvimento de um compilador, e tem por objetivo destinar um número grande de valores de um programa para um número pequeno de registradores de máquina. Sendo assim, para o autor Muchnick [4] a alocação de registradores é uma das mais importantes otimizações aplicadas por um compilador e assim determinar quais valores do programa (variáveis e temporários) devem ser mantidos em registradores ou representados em memória.

Em uma máquina real o número de registradores é reduzido, porém rápidos para acessar, dessa forma o problema de alocação de registradores consiste em minimizar a quantidade de valores do programa que são representados em memória, uma vez que, registradores são muito mais rápidos. Neste contexto, faz-se necessário uma análise da vivacidade de uma variável para determinarmos como estão dispostas dentro do código e também, temos algoritmos baseados em coloração de grafos que têm sido tradicionalmente utilizados para resolução do presente problema.

3.1 Análise de Vida de Variáveis

A tradução de um programa em uma linguagem intermediária possui uma quantidade ilimitada de temporários que devem ser executados em uma máquina com um número limitado de registradores. Dessa forma, o compilador precisa analisar a representação intermediaria para determinar quais temporários estão em uso ao mesmo tempo.

Sendo assim, uma variável é considerada "viva"se tiver um valor que possa ser utilizado em outro ponto do código. Por meio de uma análise de vivacidade de uma variável, pode-se dois temporários alocar-se no mesmo registrador caso não estejam em uso ao mesmo tempo. Para realizar análises em um programa, muitas vezes, é útil confeccionar um gráfico de fluxo de controle ou por meio de um algoritmo interativo, em que é possível analisar os *live-in* e *live-out* mediante o algoritmo abaixo.

$$in[n] = use[n] \cup (out[n] - def[n])$$

$$out[n] = \bigcup_{s \epsilon succ[n]} in[s]$$

Algorithm 1 Computação do tempo de vida por iteração

1: **for** n **faça**
2: $in[n] \leftarrow \{\}; out[n] \leftarrow \{\}$
3: **repeat**
4: **for** n **faça**
5: $in'[n] \leftarrow in[n]; out'[n] \leftarrow out[n]$
6: $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
7: $out[n] \leftarrow \bigcup_{s \epsilon succ[n]} in[s]$
8: **until** in'[n] = in[n] and out'[n] = out[n] for all n

A seguir temos um exemplo aplicando o algoritmo 1. Esse processo está demostrado de forma mais otimizada em *bottom-up* e com apenas um passo analisamos os *live-in* e *live-out* de cada instrução do código.

Algorithm 2 Código intermediário

$livein\ k, j$
2: $g = M[j + 12]$
$h = k - 1$
4: $f = g * h$
$e = M[j + 8]$
6: $m = M[j + 16]$
$b = M[f]$
8: $c = e + 8$
$d = c$
10: $k = m + 4$
$j = b$
12: $liveout\ d, k, j$

Tabela 1 – Análise de Vida das Variáveis

def	uso	in	out
g	j	j,k	k,g,j
h	k	k,g,j	g,h,j
f	g,h	g,h,j	j,f
e	j	j,f	j,f,e
m	j	j,f,e	f,e,m
b	f	f,e,m	e,m,b
c	e	e,m,b	c,m,b
d	c	c,m,b	m,b,d
k	m	m,b,d	b,d,k
j	b	b,d,k	d,j,k

3.2 Coloração de Grafos

O problema de coloração de grafos consiste na atribuição de um número mínimo de cores, pertencentes a um conjunto finito para grafo não orientado. A ideia do problema, e colorir os vértices de um grafo de tal forma que não haja dois vértices adjacentes com a mesma cor.

Para solucionar este tipo de problema começamos a colorir o "Vértice A"com uma cor definida dentro do conjunto finito de cores. Posteriormente, colorir o "Vértice B", com uma cor pertencente ao conjunto finito, mas que não foi utilizada pelo vértice adjacente a ele, assim até que todo o grafo seja preenchido.

Para exemplificar esta ideia, segue a figura abaixo:

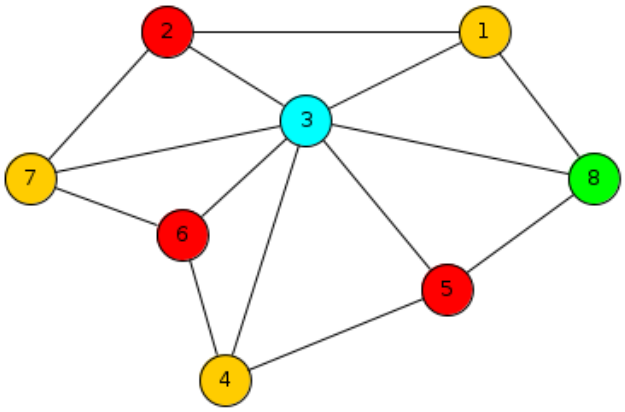


Figura 6 – Coloração de Grafos

Este tipo abordagem é utilizado em muitas aplicações práticas, como por exemplo: coloração de mapas, escalonamento de tarefas em sistemas operacionais, alocação de recursos para sistemas distribuídos, dentre outros. Este problema de coloração de grafo é descrito como um problema de decisão, ou seja, dado um grafo G e um número K de cores possíveis, o algoritmo responde se é possível ou não colorir o grafo G com as K cores.

Para solucionar o problema da coloração de grafos, a computação se utiliza de algumas técnicas, como: algoritmos gulosos com heurísticas, técnica simples que se baseia em escolher uma cor para um vértice no grafo de tal forma que não tenha vértices adjacentes da mesma cor, caso existam vértices de cor branca, pintamos com outra cor, até colorir o grafo completamente. Outra técnica seria o algoritmo de coloração utilizando busca em largura, entre outras técnicas conhecidas.

Observando a descrição do problema, e a forma de solucioná-lo, em um primeiro momento podemos dizer que este é um simples problema. Para resolver basta escolher n cores para n vértices e o problema estaria resolvido. Com certeza esta seria uma solução, porém, para dizer que a abordagem foi eficiente, devemos colorir os n vértices, com o

mínimo de cores disponíveis no conjunto finito. Partindo deste ponto, o problema ganha seu alto grau de dificuldade, classificando como de difícil solução, pertencente a Classe de problemas `NP-Completo`.

3.3 Algoritmo de Appel para Alocação de Registradores

Com base em Appel e Palsberg (2003) o processo de alocação de registradores consiste em atribuir temporários para um pequeno número de registradores de máquina. Assim, o trabalho ocorre em seis etapas como mostra a figura 7. Vale ressaltar que, cada etapa do algoritmo é executada até falhar e só com isso avança-se para próxima etapa.

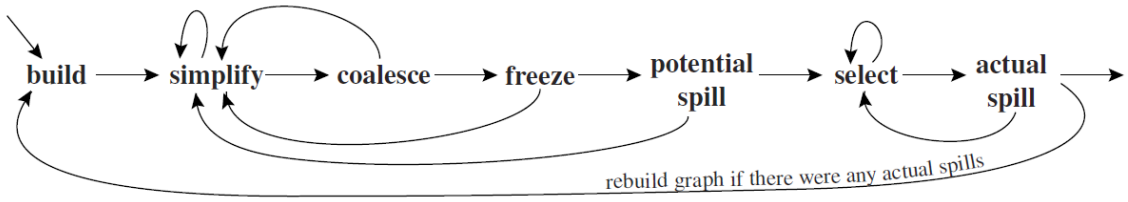


Figura 7 – Algoritmo de Appel para colocação de grafos

Na operação *Build*, ocorre a construção do gráfico de interferência, por meio da análise de longevidade das variáveis, na qual consiste o conjunto de temporários que estão simultaneamente vivos em cada ponto do programa.

Em seguida temos a operação *Simplify* que consiste em simplificar o gráfico utilizando uma heurística, assim, definimos um valor K referente ao número de registradores de máquina. Supomos que o grafo G tenha um nó `m` com menos que K vizinhos, faça G' = G - m, que significa remover o nó m do grafo G e assumir que, se G' pode ser colorido com K cores, G também pode. Desta forma, os nós removidos são colocados em uma pilha e cada remoção diminui o grau dos nós em G, dando oportunidades para novas remoções.

A etapa de *Coalesce* refere-se a simplificação do gráfico, essa operação só pode ser aplicada em nós classificados como nós de movimentação do tipo a = C. Temos ainda, duas abordagens para realizar o coalesce, que são: *George* e *Brigss*. A abordagem segundo *George* consiste em unir dois nós a, b se todo vizinho t do nó a interfere com b ou o grau (t) é menor que k. Já na abordagem de *Brigss* aplica-se quando no nó resultante a quantidade de vizinhos com grau maior ou igual a k tem que ser menor do que k.

Na etapa *freeze* é executada quando nem a *simplify* e nem o *coalesce* podem ser aplicados, assim busca identificar os nós de movimentação marcados no gráfico de grau baixo e aplica-se *freeze* nesses nós, o que faz com que eles deixem de ser nós de movimentação e passam a ser nós comuns, o que torna-os possíveis candidatos a simplificação.

A seguir temos a *Spill*, neste ponto não temos nenhum nó com grau < K, desta forma temos que marcar um nó para *Spill*, e essa escolha, também baseia-se em heurística simples, tais como: nó que reduz o maior número de grau de outros nós ou nó com menor custo relacionado a operação de memória.

E por fim a etapa *Select* que consiste em reconstruir o gráfico G adicionando os nós na ordem determinado pela pilha, quando adiciona-se um nó deve atribuir uma cor para ele, isso não vale para nós marcados como *Spill*, uma vez que, se os vizinhos já usarem K cores não adiciona o nó *Spill* no gráfico. Pode ocorrer que o *select* não consiga atribuir uma cor a algum nó, com isso, retorna-se a etapa 1, reescrever o código, pegar os valores da memória antes de cada uso e armazena-los na memória após o uso e assim, aplicar o algoritmo seguindo as etapas.

A seguir, trazemos um exemplo completo da aplicação do algoritmo de appel para alocação de registradores. Algumas definições do grafo de exemplo são que as arestas marcadas de azul são os nós de movimentação e a quantidade de registradores definidos é K=4.

Inicialmente a figura 8a mostra o grafo construído a partir da análise de vida das variáveis, método apresentado anteriormente neste trabalho, já as figuras 8b 8c 8d 8e 8f 8g trazem o processo de simplificação para os nós com grau insignificante (<k). Por sua vez, a figura 8h mostra a falha do passo de *Simplify*, que de acordo com a definição, não podemos simplificar um nó de movimentação. Então neste momento, com a falha da etapa *Simplify* o algoritmo segue para aproxima etapa que é a *Coalesce*.

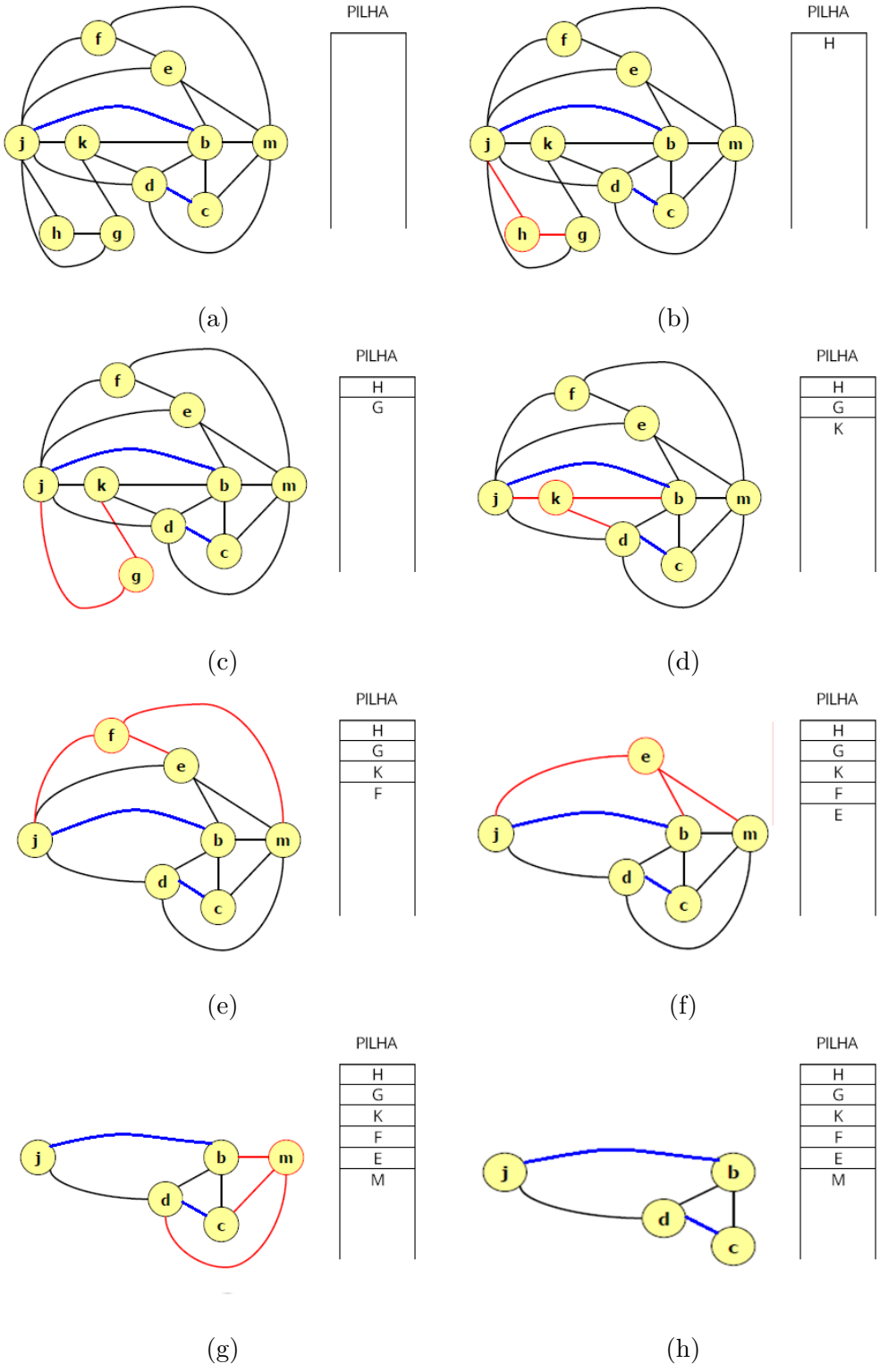
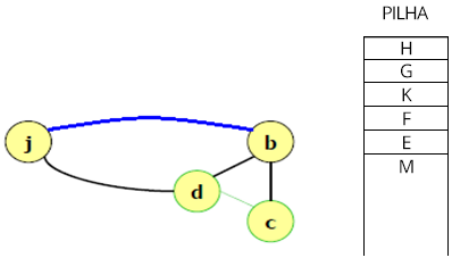


Figura 8 – (8a) Grafo inicial (8b) *Simplify* Nó H (8c) *Simplify* Nó G (8d) *Simplify* Nó K (8e) *Simplify* Nó F (8f) *Simplify* Nó E (8g) *Simplify* Nó M (8h) *Falha simplify.*

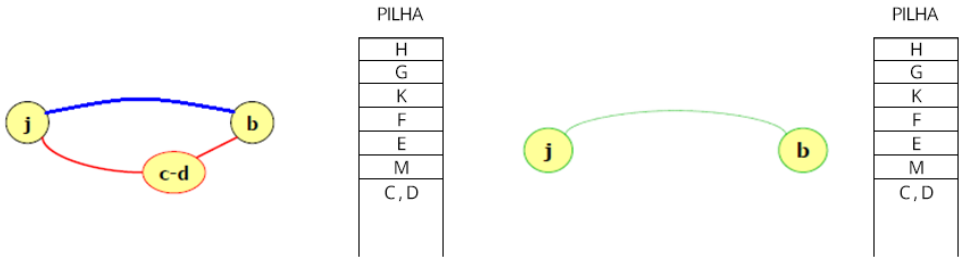
A figura 2 (a) apresenta a operação de coalesce, que é a união de dois nós de movimentação marcados com aresta de cor azul entre eles. Existe duas abordagens para determinar se os nós vão ser unidos: Georges e Brigss. E para o exemplo abaixo, os nós estão utilizando a de Brigss.



(a)

Figura 9 – (9a) *Coalesce* entre nos nós C,D.

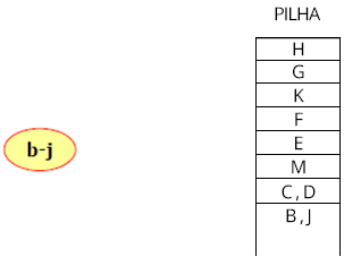
Após a operação de *coalesce* entre os nós, o algoritmo retrocede para a etapa de *simplify*. Assim, analisamos o grafo com possibilidades de simplificação e conforme notamos, podemos realizar a simplificação entre o nó (c,d).



(a) (b)

Figura 10 – (10a) *Simplify* entre nos nós C,D (10b) falha *Simplify*.

Posteriormente, temos novamente a falha desta etapa, e o algoritmo avança para a coalesce e realizamos a união dos nós restantes que são de movimentação. Temos então, um único nó para realizar a *simplify*.



(a)

Figura 11 – (11a) *Simplify* entre nos nós B,J.

Apos a simplificação de todos os nós do grafo, as etapas: *simplify*, *coalesce*, *freeze* e *Spill* falham, então é o momento da etapa *select*. Esta etapa consiste em colorir o grafo e ocorre o desempilhamento de cada nó, atribuindo assim uma cor a ele. Desta maneira pode-se apenas utilizar a quantidade de cores disponíveis, de acordo com o número de registradores definidos como: k=4. Este processo ocorre até a falha, o que significa que a pilha foi desempilhada por completo e as k cores conseguiram colorir o grafo ou a quantidade de cores não foram suficientes para colorir esse grafo.

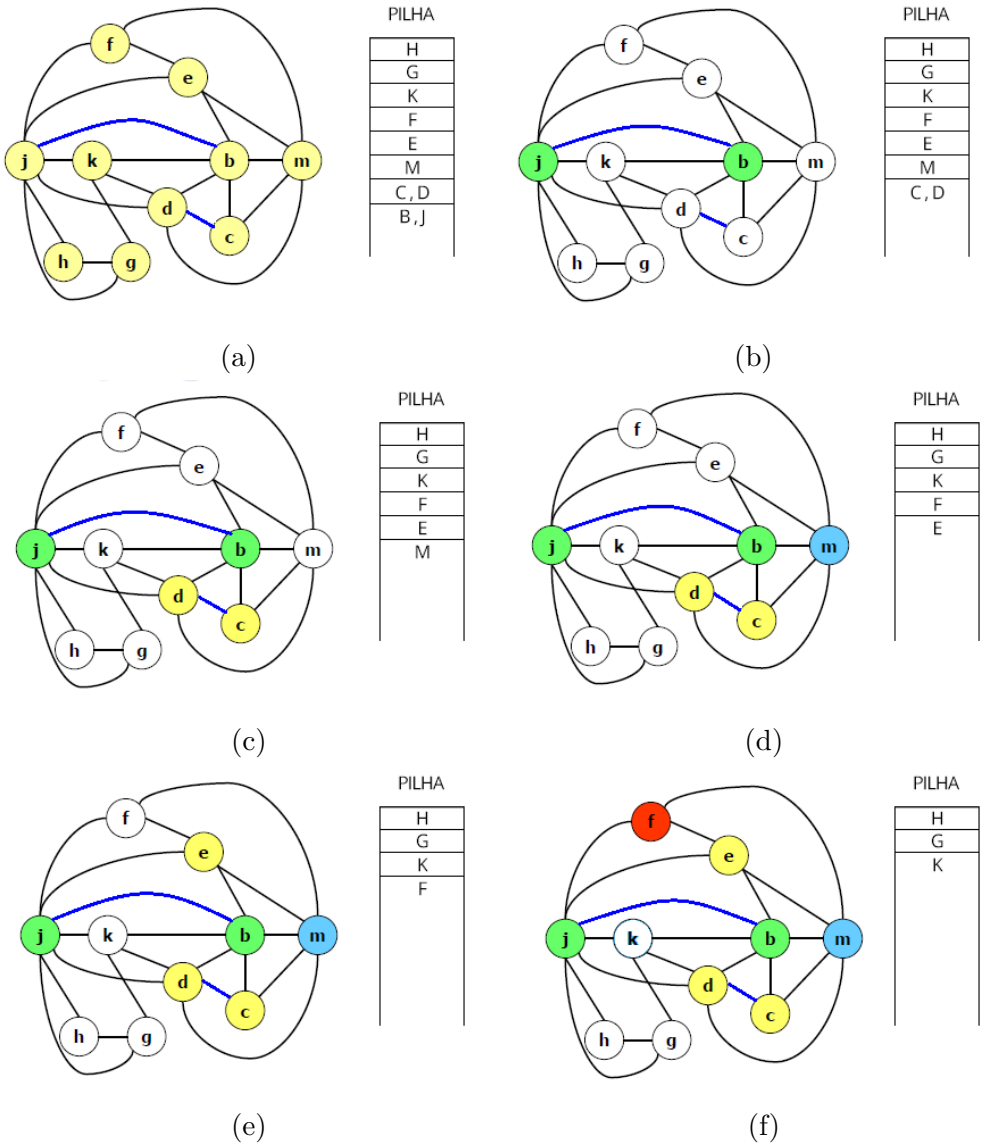


Figura 12 – (12a) Grafo inicial (12b) *Select* Nó H (12c) *Select* Nó G (12d) *Select* Nó K (12e) *Select* Nó F (12f) *Select* Nó E.

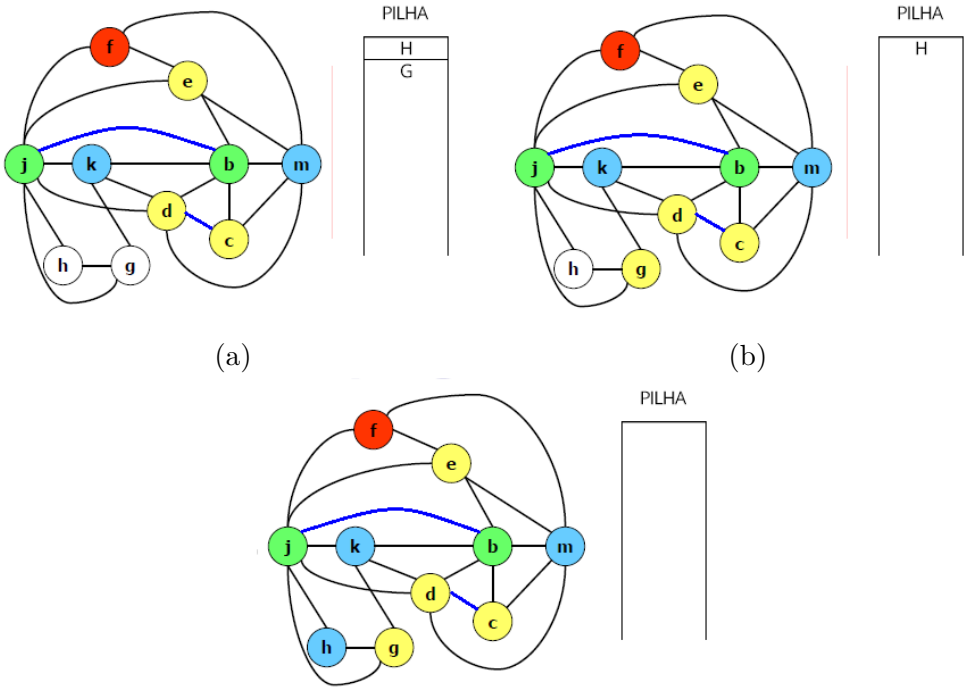


Figura 13 – (13a) *Select* Nó M (13b) *Select* (13c) grafo k colorido

Considerações finais

Com este trabalho, foi realizado o desenvolvimento das análises léxica e sintática de um compilador para a linguagem C Mais-ou-Menos, uma linguagem imperativa, fortemente tipada, mais ou menos parecida com C, sendo esta uma variação aproximada e muito restrita da linguagem em questão.

A implementação foi realizada utilizando a biblioteca PLY (*Python Lex-Yacc*) para a linguagem de programação Python. Na análise os *tokens* foram definidos com o uso de expressões regulares e na análise sintática, uma árvore sintática abstrata foi construída.

Foi realizada também, pesquisa sobre seleção de instruções e alocação de registradores através dos métodos descritos em Appel e Palsberg (2003), sendo a primeira baseada em *Maximal Munch* e Programação Dinâmica e, a segunda apresentando o processo de análise de longevidade de variáveis e o problema de coloração de grafos.

Referências

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0-201-10088-6. Citado 2 vezes nas páginas 3 e 4.

APPEL, A. W.; PALSBERG, J. *Modern Compiler Implementation in Java*. 2nd. ed. New York, NY, USA: Cambridge University Press, 2003. ISBN 052182060X. Citado 3 vezes nas páginas 3, 19 e 25.

BEAZLEY, D. *PLY (Python Lex-Yacc)*. 2001–2017. <http://www.dabeaz.com/ply/>. Acesso em: 31 de julho de 2017. Citado na página 4.