ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Departamento de Engenharia de Computação e Sistemas Digitais

RELATÓRIO DA P2

LINGUAGENS E COMPILADORES - PCS 2056

Professor: Ricardo Rocha

Pedro Vicente Martinez 7630826

Vinícius Adaime Alves de Melo 6849971

SÃO PAULO
2016

1. Considerações Iniciais

Durante o quadrimestre tivemos a experiência de desenvolver um compilador simples para uma linguagem que foi totalmente especificada por nós. Tal projeto nos possibilitou reconhecer aspectos das etapas de desenvolvimento de um compilador que podem se tornar armadilhas para as etapas subsequentes. Sendo assim procuramos utilizar a experiência adquirida para nos ajudar neste novo projeto.

Primeiramente percebemos que uma linguagem com estruturas de dados mais complexas poderia diminuir o tempo de desenvolvimento e tornar algumas tarefas mais fáceis. Desta forma escolhemos a linguagem Ruby.

Adicionalmente aprendemos que excesso de simplificação das submáquinas do Autômato de Pilha Estruturado responsável por fazer o reconhecimento sintático, pode dificultar muito associar ações semânticas às transições dos autômatos, desta forma é necessário equilibrar as coisas.

2. Análise léxica

Um dos principais motivos para a escolha da linguagem Ruby foi a possibilidade de utilização de expressões regulares para identificar palavras, o que facilitou muito a marcação de tokens durante a etapa de análise léxica.

Nesta etapa contudo há uma diferença de algoritmos entre aquele que utilizamos durante o projeto desenvolvido ao longo do quadrimestre e este compilador. Como a linguagem lambda permite rótulos que podem apontar para posições de memória posteriores (análogo a utilizar uma variável antes de declarar ou atribuir um valor ela), se fez necessário primeiramente percorrer todo o código identificando os rótulos e armazenando-os em um mapa antes que o analisador sintático pudesse solicitar tokens.

Adicionalmente é necessário que se saiba a posição de memória do rótulo com antecedência. Desta forma sempre que identificamos um rótulo armazenamos também sua posição na memória, conforme pode ser visto na Figura 1.

Figura 1 - Código responsável por armazenar

Pode-se perceber também da Figura 1 que além do `base_address` (256) foi utilizada uma variável `address_deslocation` para o cálculo dessa variável. A cada token ela pode ser incrementada de um ou dois de acordo com o número de posições de memória necessárias para representar a instrução.

O código responsável pela análise léxica pode ser encontrado no arquivo **lexico.rb.**

2.1 Tipos de token

Para a utilização posterior dos tokens nas etapas de análise sintática e semântica foi necessário classificá-los, esta classificação foi feita baseada na definição formal da linguagem. Todas as submáquinas que poderiam ser simplificadas, para uma expressão regular em cima de uma string, tornaram-se então um tipo de token.

O código responsável por classificar tokens pode ser visto no arquivo **token.rb** e os tipos de token podem ser vistos na Tabela 1.

Tabela 1 - Tipos de tokens

TYPE_LABEL	`[a-zA-Z]([a-zA-Z]|[0-9])*`
TYPE_NUM	`[0-9]+`
TYPE_OPERATOR_NO_EXP	`nop`
TYPE_OPERATOR_ONE_EXP	`read, print`

TYPE_OPERATOR_TWO_EXP	load, add
TYPE_REGISTER	instruction, data, result
TYPE_EOL	EOL
TYPE_ESPECIAL_CHAR	!, :

3. Definição formal da linguagem

A definição foi feita a partir da descrição da linguagem lambda, que pode ser encontrada em (http://web.archive.org/web/20101127063243/http:/www.veling.nl/anne/lang/lambda/) contudo a descrição feita pelo autor em alguns aspectos é vaga, de forma que alguns pressupostos foram assumidos, por exemplo:

- Na declaração de um rótulo, o único tipo de argumento permitido é um número, de forma que construções do tipo `label: add data 1`, não são permitidas.
- Operações possuem zero, um ou dois argumentos que podem ser: uma operação, um registrador, um rótulo ou um número.

3.1 Descrição da linguagem em EBNF

```
_PROG = _INST { _INST } .
_INST = ( _OPERATION | TYPE_LABEL [ ":" [TYPE_NUM] | [TYPE_LABEL] ) "EOL" .
_OPERATION = _OPERATOR [ _EXP ] [ _EXP ] .
```

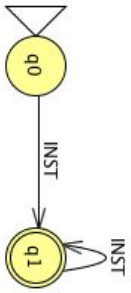
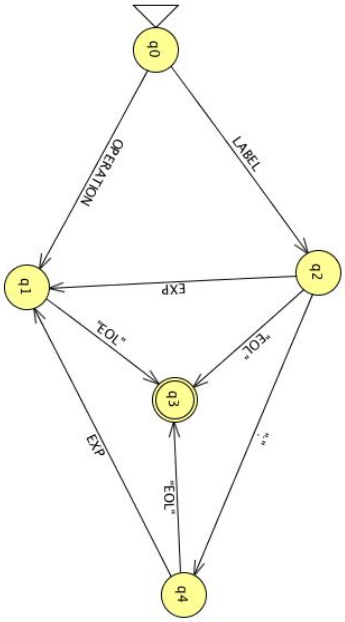
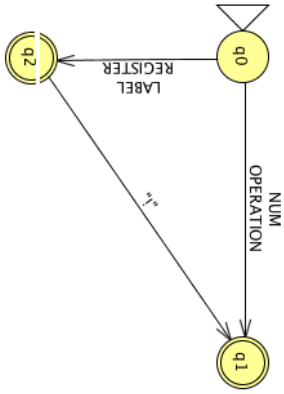
```
_EXP = _OPERATION | ( TYPE_REGISTER | TYPE_LABEL ) [ "!" ] | TYPE_NUM .


OPERATOR = ( "nop" | "read" | "load" | "add" | "print" ) .
REGISTER = ( "instruction" | "data" | "result" ) .
```

4. Analisador Sintático

A partir da descrição da linguagem feita na sessão anterior utilizando a ferramenta online disponível em (http://mc-barau.herokuapp.com/) foi gerado o Automato de Pilha Estruturado AEP, responsável por realizar o reconhecemento sintático das cadeias de entrada

PROG	**INST**	**OPERATION**	**EXP**
initial: 0 final: 1 (0, _INST) -> 1 (1, _INST) -> 1	initial: 0 final: 3 (0, _OPERATION) -> 1 (0, TYPE_LABEL) -> 2 (1, "EOL") -> 3 (2, TYPE_LABEL) -> 1 (2, ":") -> 4 (2, "EOL") -> 3 (4, TYPE_NUM) -> 1 (4, "EOL") -> 3	initial: 0 final: 1, 2, 3 (0, _OPERATOR) -> 1 (1, _EXP) -> 2 (2, _EXP) -> 3	initial: 0 final: 1, 2 (0, _OPERATION) -> 1 (0, TYPE_REGISTER) -> 2 (0, TYPE_LABEL) -> 2 (0, TYPE_NUM) -> 1 (2, "!") -> 1

PROG	INST	OPERATION	EXP
			

A primeira parte da análise sintático foi ler o arquivo de entrada **automatos.txt** e construir dentro do programa cada um dos autômatos descritos anteriormente. Cada autômato é representado por um objeto da classe `Automata` (código no arquivo **automata.rb**) que guarda informações acerca do estado atual, estado inicial, estado final, não terminais à esquerda das regras e terminais à esquerda das regras.

O analisador conta com uma pilha de máquinas, e cada entrada ou saída de submáquina ela é empilhada e desempilhada respectivamente.

Cada token lido é submetido à submáquina presente que então decide a ação devida. Para lidar com transições de não terminais foi criada a função `current_state_subautomatas_terminals()` que recursivamente percorre todas as máquinas que podem ser acessadas do estado atual e devolve os terminais que podem ser tratados por elas.

O código do analisador sintático pode ser visto em **sintatico.rb**.

5. Analisador semântico

O Analisador semântico do presente compilador tem a função de popular a memória da máquina lambda que foi criada e cuja descrição encontra-se na próxima sessão, desta forma foram feitas algumas modificações em relação ao fluxo do compilador que foi feito ao longo do quadrimestre.

Cada token recebido pela função `semantic_action()` é convertido em uma instrução válida e adicionado à memória, sequencialmente a partir da posição 256:

- **Operações e Registradores (nop, read, load, add, print, instruction, data, result):** Opcode correspondente é inserido na memória.
- **Número inteiro:** É colocado diretamente na memória seguido por 255 na posição de memória seguinte.
- **Rótulo:** Caso o lookahead(1) seja diferente de ':' (não seja uma declaração de rótulo) a posição do rótulo (obtida durante a análise léxica) é inserida na memória.
- **'!'**: O valor 255 é inserido na memória indicando que deve-se ser utilizado o endereço do rótulo ou registrador e não o valor apontado por ele.

O código do analisador semântico pode ser visto no arquivo **semantico.rb**.

6. Máquina Lambda

O compilador produzido pelo projeto é simulado por uma máquina lambda criada em Ruby. Para o funcionamento desta máquina, fora criada a estrutura `@mem` um Array referente a memória do programa, e as rotinas `create_mem()`, `add_token()`, e `execute()`.

```
def create_mem
  @mem = []
  @mem[0..@@mem_size] = 0

end

def add_token(value)
  @mem[@@mem_size] = value
  @@mem_size += 1
end

def execute
  resetPointers

  puts 'read ' + (@@mem_size - 255).to_s + ' tokens'
  puts 'executing'

  while get(INSTRUCTION) > 0 && get(INSTRUCTION) < @@mem_size
    @printString = @mem[INSTRUCTION].to_s + ':'
    readInstruction
    puts @printString
  end

  if get(INSTRUCTION) >= @@mem_size
    puts 'INFINITY; ready.'
  else
    puts 'ready.'
  end
end
```

A geração de código começa pela função `create_mem()`, que popula a array `@mem` com 256 valores 0 (equivalentes à função NOP). Conforme os tokens são lidos no sintático, chamadas `add_token()` são realizadas, colocando o valor do token (definido pelo semântico) numa nova posição `@mem`.

O `execute()` é a função chamada após todo o código a ser compilado, sendo então a função que realmente simula uma máquina lambda. Essa função possui como lógica 2 passos.

1. Inicialização dos registradores `INSTRUCTION`, `DATA` e `RESULT`, que são as posições 32, 34 e 36 da `@mem`.
2. Um loop então se inicia, existindo enquanto o registrador `INSTRUCTION` apontar para um endereço de `@mem` válido (maior que 0 e menor que a última posição da memória). A estratégia nesse loop é de chamar a rotina interna `readInstruction`, que lê a instrução atual (apontada por `@mem[INSTRUCTION]`) e recursivamente pega todos os parâmetros necessários até resolvê-la. Basicamente pode-se dizer que essa função cria uma recursão que só termina no

token EOL, podendo se então falar que o loop é a leitura de cada linha do código lambda, e o `readInstruction()` é a leitura de cada token, e sua interpretação.

Durante runtime (função `execute`) a interface de manipulação da memória (`@mem`) é por meio do get e set (com ponteiros). O que vale comentar, que não é detalhado no artigo que explica a linguagem (http://web.archive.org/web/20101127063243/http:/www.veling.nl/anne/lang/lambda/), é a utilização interna da máquina lambda do valor 255, que é salvo na memória sempre que se tem interesse em dizer que a posição de memória anterior significa um número, e não uma função/jump/variável. Essa informação se mostra relevante não só para simular a máquina lambda, mas para a própria geração de código, uma vez que a proposta da linguagem Lambda é de se alterar o código em runtime, e para que isso seja possível/planejado, é necessário que o programador entenda o posicionamento dos tokens na memória, incluindo então o uso do valor 255 e seu significado.

7. Execução e testes

Para a execução do compilador e comparação de seu funcionamento com o compilador oficial em java a dupla criou o script compilar.sh, que imprime o programa analisado, e então imprime o compile e runtime do projeto da dupla em ruby, e do compilador oficial em java.

```
#!/bin/sh
clear

echo "Prova 2 - Compiladores - Dezembro de 2016"
echo "   Pedro Martinez e Vinicius Adaime"

echo "\n\nPrograma exemplo (ENTRADA.txt)"
echo "------------------------------"
more ENTRADA.txt

echo "\n\nCompilando programa exemplo"
echo "---------------------------"
compilador/main.rb

echo "\n\nComparando com o compilador exemplo!"
echo "-----------------------------------"
java Lambda ENTRADA.txt
```

Por conta dessa comparação, os prints do compilador em ruby foram bolados para replicar os prints do compilador em java, imprimindo 1º a tokenização do léxico, 2º a conversão dos tokens em código pelo semântico, e então em 3º o runtime do código. Vale ressaltar que foi feita uma mínima alteração ao código do compilador java original para que esse imprimisse o código gerado pelo semântico, para facilitar a depuração do compilador.

7.1 Execução do programa add5

```
Prova 2 - Compiladores - Dezembro de 2016
   Pedro Martinez e Vinicius Adaime


Programa exemplo (ENTRADA.txt)
------------------------------
load x! 50
add5 x
print x

exit

x: 0


add5:
add read data 5
load instruction! add data! 1

exit:
```

```
Compilando programa exemplo
---------------------------
tokenizing 256 [load x! 50]
tokenizing 261 [add5 x]
tokenizing 263 [print x]
tokenizing 265 [exit]
tokenizing 266 [x: 0]
tokenizing 268 [add5:]
tokenizing 268 [add read data 5]
tokenizing 273 [load instruction! add data! 1]
tokenizing 281 [exit:]
compiling
256: 2
257: 266
258: 255
259: 50
260: 255
261: 268
262: 266
263: 4
264: 266
265: 281
266: 0
267: 255
268: 3
269: 1
270: 34
271: 5
272: 255
273: 2
274: 32
275: 255
276: 3
277: 34
278: 255
279: 1
280: 255
compiled
read 26 tokens
executing
[266]<--50
256: load ( 266) ( 50)
[266]+=5
261: jump 268 ( add ( (read read 34)) ( 5))
[34]+=1
[32]<--263
273: load ( 32) ( add ( 34) ( 1))
-->55
263: print read 266
265: jump 281 ( nop)
INFINITY; ready.
```

```
Comparando com o compilador exemplo!
------------------------------------
tokenizing 256 [load x! 50]
tokenizing 261 [add5 x]
tokenizing 263 [print x]
tokenizing 265 [exit]
tokenizing 266 [x: 0]
tokenizing 268 [add5:]
tokenizing 268 [add read data 5]
tokenizing 273 [load instruction! add data! 1]
tokenizing 281 [exit:]
compiling
256: 2
257: 266
258: 255
259: 50
260: 255
261: 268
262: 266
263: 4
264: 266
265: 281
266: 0
267: 255
268: 3
269: 1
270: 34
271: 5
272: 255
273: 2
274: 32
275: 255
276: 3
277: 34
278: 255
279: 1
280: 255
compiled
read 26 tokens
executing
[266]<--50
256: load ( 266) ( 50)
[266]+=5
261: jump 268 ( add ( (read read 34)) ( 5))
[34]+=1
[32]<--263
273: load ( 32) ( add ( 34) ( 1))
-->55
263: print read 266
265: jump 281 ( nop)
INFINITY; ready.
```

7.2 Execução do programa print999

```
Prova 2 - Compiladores - Dezembro de 2016
   Pedro Martinez e Vinicius Adaime


Programa exemplo (ENTRADA.txt)
------------------------------
//start of program
print999

back:
exit

//the declaration of the procedure
print999:
print 999
back

//end of program
exit:
```

```
Compilando programa exemplo
---------------------------
tokenizing 256 [print999]
tokenizing 257 [back:]
tokenizing 257 [exit]
tokenizing 258 [print999:]
tokenizing 258 [print 999]
tokenizing 261 [back]
tokenizing 262 [exit:]
compiling
256: 258
257: 262
258: 4
259: 999
260: 255
261: 257
compiled
read 7 tokens
executing
-->999
256: jump 258 ( print 999)
261: jump 257 ( jump 262 ( nop))
INFINITY; ready.
```

```
Comparando com o compilador exemplo!
------------------------------------
tokenizing 256 [print999]
tokenizing 257 [back:]
tokenizing 257 [exit]
tokenizing 258 [print999:]
tokenizing 258 [print 999]
tokenizing 261 [back]
tokenizing 262 [exit:]
compiling
256: 258
257: 262
258: 4
259: 999
260: 255
261: 257
compiled
read 7 tokens
executing
-->999
256: jump 258 ( print 999)
261: jump 257 ( jump 262 ( nop))
INFINITY; ready.
```