Escola Politécnica da Universidade de São Paulo

Linguagens e Compiladores

Relatório P2

Henrique Sussumu Matsui Kano
Mi Che Li Lee

4 de dezembro de 2015

Sumário

Capítulo 1

Construção do reconhecedor

Nessa etapa foi feita a construção do reconhecedor para a linguagem LazyComb baseado no autômato de pilha estruturado.

1.1 Notação BNF

Linguagem em notação BNF:

```
<Program>  ::= <CCExpr>

<CCExpr>   ::= <CCExpr> <Expr> | epsilon

<Expr>     ::= i | <Expr'>

<IotaExpr> ::= i | <Expr'>

<Expr'>    ::= I
             | K | k
             | S | s
             | <NonemptyJotExpr
             | ` <Expr1> <Expr2>
             | * <IotaExpr1> <IotaExpr2>
             | ( <CCExpr> )

<NonemptyJotExpr> ::= <JotExpr> 0
                    | <JotExpr> 1

<JotExpr>  ::= <NonemptyJotExpr> | epsilon
```

1.2 Notação Wirth

A partir da notação em BNF, foi obtida a notação em Wirth:

```
Program      = CCExpr.

CCExpr       = { Expr }.

Expr         = "i" | Expr'

IotaExpr     = "i" | Expr'

Expr'     ::= "I"
            | "K" | "k"
            | "S" | "s"
            | NonemptyJotExpr
            | "`" Expr Expr
            | "*" IotaExpr IotaExpr
            | "(" CCExpr ")"

NonemptyJotExpr ::= JotExpr "0"
                  | JotExpr "1"

JotExpr = NonemptyJotExpr | epsilon.
```

1.3 Análise Léxica

Como todos os terminais da linguagem são compostos apenas por um caracter, a tarefa do analisador léxico se torna simples, basta verificar se a entrada está contida no conjunto de caracteres válidos.

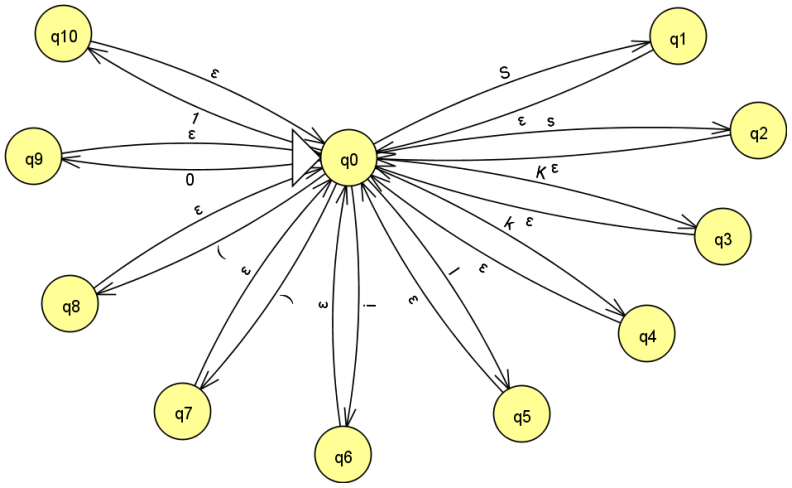

Figura 1.1: Transdutor - Analisador léxico

1.4 Análise Sintática

A partir da notação em Wirth, foi feita uma simplificação:

```
Program = { Expr }.

Expr = "i" | "I" | "K" | "k" | "S" | "s" | { "0" | "1" }
    | "`" Expr Expr | "*" IotaExpr IotaExpr | "(" { Expr }
     ")".

IotaExpr = "i" | "I" | "K" | "k" | "S" | "s" | { "0" |
    "1" } | "`" Expr Expr | "*" IotaExpr IotaExpr | "(" {
    Expr } ")".
```

Para cada não terminal foi gerada uma máquina sintática, representada pelos seu estado inicial, estados finais e transições entre estados. Cada transição pode se manifestar na forma de chamada ou retorno para uma submáquina, ou execução de uma ação semântica, tudo isso usando autômato de pilha estruturado (empilhando ou desempilhando máquinas).

Utilizando o site http://mc-barau.herokuapp.com/ e o programa JFLAP, a descrição da linguagem em notação de Wirth resultou nas seguintes máquinas:

Program

```
Program = 0 { 1 Expr 2 } 1 .
```

```
Program:
Minimized DFA:
initial: 0
final: 1
(0, Expr) -> 1
(1, Expr) -> 1
```

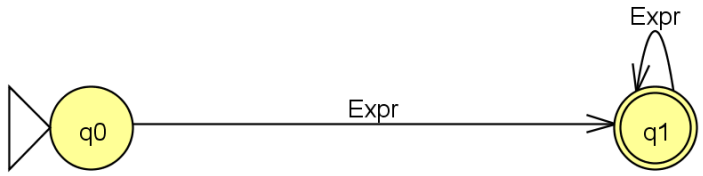

Figura 1.2: Program

Expr

```
Expr = 0 "i" 1 | 0 "I" 2 | 0 "K" 3 | 0 "k" 4 | 0 "S" 5 | 0 "s
   " 6 | 0 { 7 "0" 8 | 7 "1" 9 } 7 | 0 "‘" 10 Expr 11 Expr 12
    | 0 "*" 13 IotaExpr 14 IotaExpr 15 | 0 "(" 16 { 17 Expr
   18 } 17 ")" 19 .

Sub-maquina Expr:
Minimized DFA:
initial: 0
final: 1, 2
(0, "i") -> 1
(0, "I") -> 1
(0, "K") -> 1
(0, "k") -> 1
(0, "S") -> 1
(0, "s") -> 1
(0, "0") -> 2
(0, "1") -> 2
(0, "‘") -> 3
(0, "*") -> 4
```

```
(0, "(") -> 5
(2, "0") -> 2
(2, "1") -> 2
(3, Expr) -> 7
(4, IotaExpr) -> 6
(5, Expr) -> 5
(5, ")") -> 1
(6, IotaExpr) -> 1
(7, Expr) -> 1
```

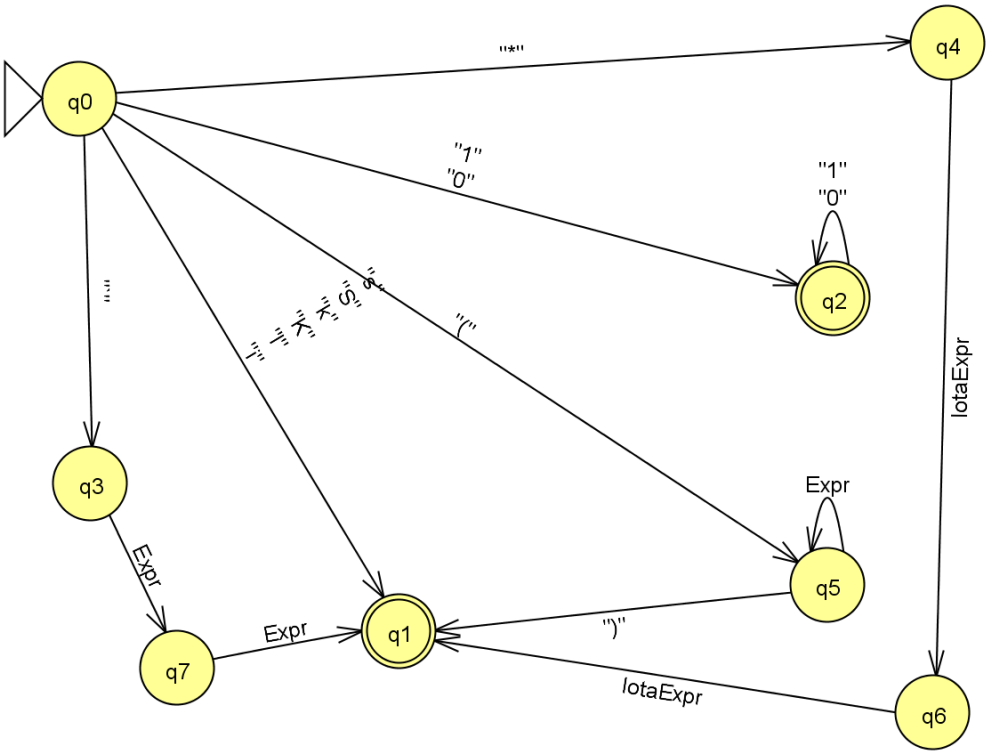

Figura 1.3: Expr

IotaExpr

```
IotaExpr = 0 "i" 1 | 0 "I" 2 | 0 "K" 3 | 0 "k" 4 | 0 "S" 5 |
   0 "s" 6 | 0 { 7 "0" 8 | 7 "1" 9 } 7 | 0 "`" 10 Expr 11
   Expr 12 | 0 "*" 13 IotaExpr 14 IotaExpr 15 | 0 "(" 16 { 17
    Expr 18 } 17 ")" 19 .
```

```
Sub-maquina IotaExpr:
Minimized DFA:
initial: 0
final: 1, 2
(0, "i") -> 1
(0, "I") -> 1
(0, "K") -> 1
(0, "k") -> 1
(0, "S") -> 1
(0, "s") -> 1
(0, "0") -> 2
(0, "1") -> 2
(0, "`") -> 3
(0, "*") -> 4
(0, "(") -> 5
(2, "0") -> 2
(2, "1") -> 2
(3, Expr) -> 7
(4, IotaExpr) -> 6
(5, Expr) -> 5
(5, ")") -> 1
(6, IotaExpr) -> 1
(7, Expr) -> 1
```

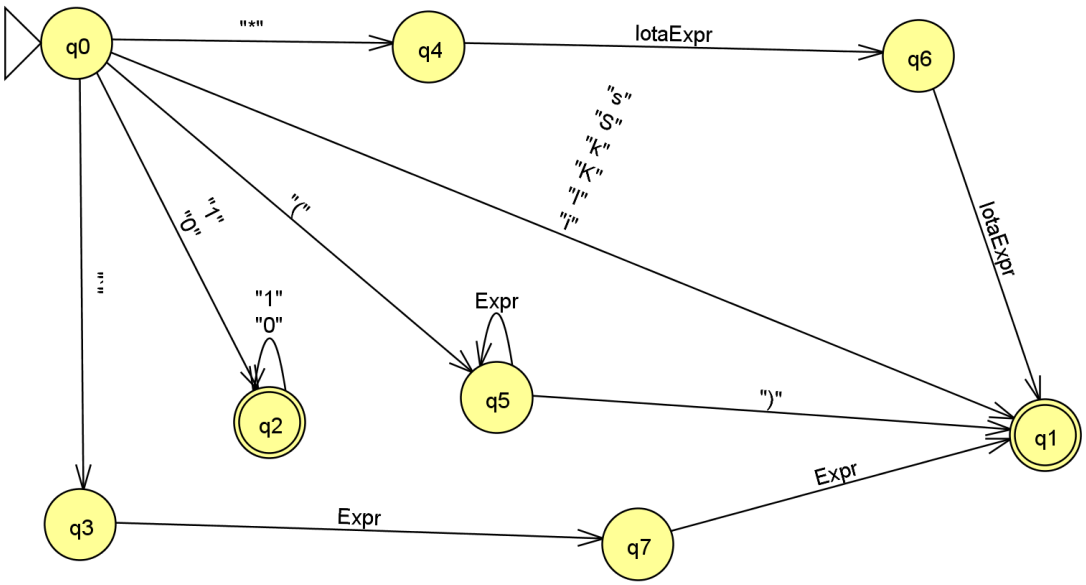

Figura 1.4: IotaExpr

Capítulo 2

Rotinas

2.1 Implementação

Com os automatos de pilha prontos, bastou fazer a implementação. Para isso, foi criado uma struct (AutomataPE) que representa um tipo de monitor. Esse monitor é responsável por gerenciar a pilha (empilhando e desempilhando quando necessário os sub-autômatos), criar os sub-autômatos chamados, e gererenciar as transições dos autômatos. Para isso, são guardadas várias tabelas auxiliares que informam pontos importantes dos autômatos, que foram identificados com um id próprio cada um:

- transitionTables: Um array de tabelas indexado pelo id da máquina em questão que informa, para cada estado da máquina e token recebido, qual o proximo estado dela;
- subMachineCall: Um array de tabelas indexado pelo id da máquina em questão que informa, para o estado atual dessa máquina, qual o id da máquina deve ser chamada. Essas tabelas acabam por ser matrizes-coluna;
- afterCallStates: Um array de tabelas indexado pelo id da máquina em questão que informa, para a máquina que acabou de ser desempilhada, para qual estado ela deve ir. São tabelas id x id;
- finalStates: Um array de tabelas indexado pelo id da máquina em questão que informa, para a máquina em questão quais estados são finais;

Com essas informações, falta apenas a lógica para fazer as transições: primeiramente, deve ser chamada a função initMachines para que sejam inicializadas as tabelas descritas anteriormente. Dessa chamada será retornado um AutomatoPE. Esse deve então ser usado para chamar a função automataPERun, que

recebe também o arquivo que contém o programa e um token que será usado de buffer e deve, de inicio, conter o token inicial. Com essas informações, é possível calcular o próximo estado da máquina inicial. Desse ponto são várias as situações possíveis:

- Se esse próximo estado não existir na tabela de transição, deve-se investigar se nesse estado é possível chamar alguma máquina;
- Se na tabela de chamada de submaquinas existir uma submáquina a ser chamada, basta empilhar a submaquina atual e começar a nova submáquina;
- Se na tabela de chamada de submaquinas não existir um id de máquina válido, ainda é possível que o estado em que o autômato reside é final
  - Se o estado for final, é necessário desempilhar uma máquina e recomeçar o processo para verificar se o token é reconhecido por máquinas "a cima;
  - Se o estado não for final, isso significa que o reconhecimento falhou;

Por causa dessa implementação preditiva (qual deve ser o próximo estado), deve ser dado o primeiro token para começar a operação. E como uma ação de desempilhar e empilhar não envolvem o consumo de token, mas são considerados como um passo do algoritmo, deve-se usar um buffer para não perder tokens já lidos.

Referências Bibliográficas

[1] R. Ricardo S. Jaime A. Reginaldo, B. Anarosa. Introdução máquina de von neumann.

[2] João José Neto. *Introdução à Compilação*. Escola Politécnica da USP, 1 edition, 1986.