UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CURSO DE CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA: CONSTRUÇÃO DE COMPILADORES
PERÍODO: 2013.2

Analisador sintático do Ptgrila

ALEX REIMANN CUNHA LIMA
ANDREY VINICIUS FAGUNDES
GUSTAVO KNOB
RAMON PERONDI

Chapecó – SC, Janeiro de 2014

Resumo

Análise sintática (parsing) é o processo de analisar uma sequência de entrada (lida de um arquivo ou teclado) para determinar sua estrutura gramatical segundo uma determinada gramática formal. Essa análise faz parte do compilador, sendo executada logo após com a análise léxica e anterior à análise semântica. Existem duas formas de fazer esta análise: Descendente, implementado pelo analisador sintático LL ou Ascendente, implementado pelo analisador LR, o qual será desenvolvido neste trabalho.

Introdução

O objetivo deste trabalho é conhecer um pouco mais a fundo o analisador sintático de um compilador: sua estrutura e seu funcionamento. Mas para isso, foi preciso entender mais a fundo sobre linguagem de programação: suas gramáticas, regras e sintaxes. Para tanto, foi desenvolvido uma linguagem hipotética: a linguagem Ptgrila, que foi construída unicamente para aplicar os conceitos vistos em sala de aula afim de compreender o funcionamento das partes de um compilador, bem como noções de gramática e linguagens formais.

Usando como entrada do analisador sintático a saída do analisador léxico (implementado no trabalho anterior), para que o analisador sintático possa usar um conjunto de regras para construir uma árvore sintática. Ou seja, sua tarefa é basicamente determinar se uma entrada de dados pode ser derivada de um símbolo inicial com as regras de uma gramática formal, podendo ser feita de duas formas:

- Descendente (top-down) - um analisador pode iniciar com o símbolo inicial e tentar transformá-lo na entrada de dados. Intuitivamente, o analisador inicia dos maiores elementos e os quebra em elementos menores. Exemplo: analisador sintático LL.

- Ascendente (bottom-up) - um analisador pode iniciar com um entrada de dados e tentar reescrevê-la até o símbolo inicial. Intuitivamente, o analisador tentar localizar os elementos mais básicos, e então elementos maiores que contêm os elementos mais básicos, e assim por diante. Exemplo: analisador sintático LR.

Analisador LR

Um analisador sintático LR, também chamado de parser LR, é um algoritmo de análise sintática para gramáticas livres de contexto. Ele lê a entrada de texto da esquerda para a direita e produz uma derivação mais à direita, por isso LR, do termo em inglês left-right (diferente do analisador sintático LL). Um analisador sintático LR é ascendente, pois tenta deduzir as produções da gramática a partir dos nós folhas da árvore, chegando até a raiz.

Para implementaçao deste analisador ascendente é nescessario primeiro criar as regras sintaticas, após isso, enumera-las, o proximo passo é criar os estados e os itens completos, indentificar o conjunto de transiçoes, fazer o conjunto first-follow e por fim, criar a tabela de parse.

Tokens da linguagem

A linguagem Ptgrila é composta basicamente de palavras reservadas: palavras que devem aparecer literalmente na linguagem, sem variações (e.g. SI, SINON, PUIS, etc). Constantes: literais TRUE e FALSE, símbolos especiais: sequências de um ou mais símbolos que não podem aparecer em identificadores nem palavras reservadas. São utilizados para composição de expressões aritméticas ou lógicas, comando de atribuição, etc. São exemplos de símbolos especiais: “<” (menor), “:” (dois pontos), “:” (atribuição), etc. Identificadores/variáveis: palavras que seguem algumas regras de escrita, porém podem assumir diversos valores. São definidos através da expressão regular: “[\$]+[a-b-c]{3}”, ou seja, palavras que começam obrigatoriamente com o caractere ‘$’ onde o restante pode ser composto pelas letras a,b ou c e

devem ter tamanho igual a 3, exemplo: $aaa, $abc, $cbc, etc. O objetivo de usar esta regra para a criação das variáveis foi apenas para facilitar na construção dos autômatos e, consequentemente, na criação do analisador léxico.

As tabelas abaixo mostram todos os tokens desta linguagem:

Palavras reservadas	
se	si
então	puis
senao	sinon
enquanto	pendant
faça	faire
para	paire
inteiro	ensemble
flutuante	flottant
caractere	lettre
palavra	mat
retornar	retour

Constantes	
verdadeiro	true
falso	nalse

Variáveis / Identificadores
[\$]+[a-b-c]{3}

símbolos especiais	
Multiplicaçao	*
divisão	/
adiçao	+
subtraçao	-
igualdade	=
Desigualdade	!=

mod	%
menor que	<
maior que	>
menor ou igual	<=
maior ou igual	>=
Delimitador de linha	//
atribuição	:
abre parêntese	(
fecha parêntese	)
operador e	^
operador ou	v

E do alfabeto definido por L = {S,I,P,U,N,O,E,D,A,T,F,R,P,M,B,L,C,$,*,/,+,-,=,!,%,<,>,:,(,),^,V}.

Estados finais

A seguinte tabela traz os estados finais da linguagem, bem como seus respectivos nomes. Ao todo, são 34 estados: um para cada token da linguagem e o estado de erro (0). Para identificar quais são os estados finais foram usados números negativos, apenas para simplificar a implementação do algoritmo escrito em linguagem C.

Estado final	Nome estado
0	Fim da entrada
-1	Estado de erro
-7	si
-6	puis
-10	sinon
-16	pendant
-20	faire
-24	paire
-31	ensemble
-38	flottant

-43	lettre
-45	mat
-53	coprendre
-62	principale
-67	retour
-70	true
-74	nalse
-77	Identificadores
-78	*
-79104	/
-80	+
-90	-

-91	=
-93	!=
-94	%
-9597	<
-9698	>
-99	:
-100	(
-101	)
-102	^
-103	V
-105	//
-106	<=
-107	>=

Código fonte

```
#include <stack>
#include <ctype.h>
#include <vector>
#include <cstdio>
#include <cstring>
#include <string>

//#define DEBUG

typedef std::vector<int> ivec;

std::vector<std::string> listanomes;
std::vector<int> tamregras;
std::vector<int> regraesq;

#define MAX_ESTADOS 256
#define MAX_COLUNAS 256

int tabela_n[MAX_ESTADOS][MAX_COLUNAS];
char tabela_a[MAX_ESTADOS][MAX_COLUNAS];

void poetab_n(int estado, int coluna, int valor) {
    tabela_n[estado][coluna] = valor;
}

void poetab_a(int estado, int coluna, char valor) {
    tabela_a[estado][coluna] = valor;
}

int pegarnomenum(const char *nome) {
    for (int i = 0; i < (int)listanomes.size(); i++) {
        if (listanomes[i] == nome) {
            return i;
        }
    }
    return -1;
}

int pegar_a(int estado, int coluna) {
    char nome[25];

    if (coluna != 0) {
        sprintf(nome, "'%d'", coluna);
        coluna = pegarnomenum(nome);
    }
    if (estado < 0 || estado >= MAX_ESTADOS) {
        return '?';
    }
    if (coluna < 0 || coluna >= MAX_COLUNAS) {
        return '?';
    }
```

```
    return tabela_a[estado][coluna];
}

int pegar_n(int estado, int coluna) {
    char nome[25];

    if (coluna != 0) {
        sprintf(nome, "'%d'", coluna);
        coluna = pegarnomenum(nome);
    }
    if (estado < 0 || estado >= MAX_ESTADOS) {
        return -1;
    }
    if (coluna < 0 || coluna >= MAX_COLUNAS) {
        return -1;
    }
    return tabela_n[estado][coluna];
}


void abrarq(void) {
    {
        for (int i = 0; i < MAX_ESTADOS; i++) {
            for (int j = 0; j < MAX_COLUNAS; j++) {
                tabela_n[i][j] = -1;
            }
        }
    }
    memset(tabela_a, 0, sizeof(tabela_a));

    char linha[256];
    enum {
        NENHUM,
        LENDO_TERMINAIS,
        DEPOIS_TERMINAIS,
        LENDO_NAOTERMINAIS,
        DEPOIS_NAOTERMINAIS,
        LENDO_REGRAS,
        DEPOIS_REGRAS,
        LENDO_ESTADOS,
        LER_INI_ESTADO,
        LENDO_MEIO_ESTADO,
        LENDO_FIM_ESTADO,
        ACABOU
    } estado;

    estado = NENHUM;
    //FILE *f = fopen("/media/sf_vshared/ptgrila.txt", "rt");
    FILE *f = fopen("data/ptgrila.txt", "rt");
    int pular_linhas = 0;
    int state = -1;
    while (fgets(linha, sizeof(linha), f)) {
        std::string l = linha;

        switch(estado) {
            case NENHUM:
```

```
    if (l == "Terminals\r\n") {
        estado = LENDO_TERMINAIS;
        pular_linhas = 2;
    }
    break;
case LENDO_TERMINAIS:
    if (pular_linhas > 0) {
        pular_linhas--;
    } else {
        if (l == "\r\n") {
            estado = DEPOIS_TERMINAIS;
        } else {
            int id;
            char nome[100];
            sscanf(linha, "%d %s\r", &id, nome);
            #ifdef DEBUG
            printf("terminal <%d> <%s>\n", id, nome);
            #endif

            listanomes.push_back(nome);
        }
    }
    break;
case DEPOIS_TERMINAIS:
    if (l == "Nonterminals\r\n") {
        estado = LENDO_NAOTERMINAIS;
        pular_linhas = 2;
    }
    break;
case LENDO_NAOTERMINAIS:
    if (pular_linhas > 0) {
        pular_linhas--;
    } else {
        if (l == "\r\n") {
            estado = DEPOIS_NAOTERMINAIS;
        } else {
            int id;
            char nome[100];
            sscanf(linha, "%d %s\r", &id, nome);
            #ifdef DEBUG
            printf("naoterminal <%d> <%s>\n", id, nome);
            #endif

            listanomes.push_back(nome);
        }
    }
    break;
case DEPOIS_NAOTERMINAIS:
    if (l == "Rules\r\n") {
        estado = LENDO_REGRAS;
        pular_linhas = 2;
    }
    break;
case LENDO_REGRAS:
    if (pular_linhas > 0) {
        pular_linhas--;
```

```
            } else {
                if (l == "\r\n") {
                    estado = DEPOIS_REGRAS;
                } else {
                    bool tentar = false;
                    int tamanho  = 0;
                    int numregra;
                    sscanf(linha, "%d", &numregra);
                    const char *s = linha;
                    while(!isspace(*s)) {
                        s++;
                    }
                    while(isspace(*s)) {
                        s++;
                    }
                    char nome[30];
                    int i= 0;
                    while(!isspace(*s)) {
                        nome[i++] = *s;
                        s++;
                    }
                    nome[i] = '\0';
                    int ladoesq = pegarnomenum(nome);

                    for (s = linha; *s != '\r'; s++) {
                        if (*s == ':') {
                            tentar = true;
                        }
                        if (!tentar) {
                            continue;
                        }
                        if (*s == '-') {
                            tamanho++;
                        } else if (*s == '<') {
                            tamanho++;
                        }
                    }
                    #ifdef DEBUG
                    printf("REGRA %d tamanho=%d ladoesq=%d <%s>\n", numregra,
tamanho, ladoesq, nome);
                    #endif
                    tamregras.push_back(tamanho);
                    regraesq.push_back(ladoesq);
                }
            }
            break;
        case DEPOIS_REGRAS:
            if (l == "LALR States\r\n") {
                estado = LENDO_ESTADOS;
                pular_linhas = 2;
            }
            break;


        case LENDO_ESTADOS:
            if (pular_linhas > 0) {
```

```
                pular_linhas--;
            } else {
                if (sscanf(linha, "State %d\r\n", &state) == 1) {
                    #ifdef DEBUG
                    printf("estado <%d>\n", state);
                    #endif
                    estado = LER_INI_ESTADO;
                }
                if (strncmp(linha, "====", 4) == 0) {
                    estado = ACABOU;
                }
            }
            break;
        case LER_INI_ESTADO:
            if (strncmp(linha, "        Prior States:", 21) == 0) {
                //printf("AAA [%s]\n", linha);
                pular_linhas = 2;
            } else {
                pular_linhas = 0;
            }
            estado = LENDO_MEIO_ESTADO;
            break;
        case LENDO_MEIO_ESTADO:
            if (pular_linhas > 0) {
                pular_linhas--;
            } else {
                if (strncmp(linha, "\r\n",2) == 0) {
                    estado = LENDO_FIM_ESTADO;
                }
            }
            break;
        case LENDO_FIM_ESTADO:
            if (strncmp(linha, "\r\n",2) == 0) {
                estado = LENDO_ESTADOS;
            } else {
                char nome[100];
                char comando;
                int prox;

                if (sscanf(linha, "%s %c %d", nome, &comando, &prox) == 3) {
                    int nnum = pegarnomenum(nome);
                    #ifdef DEBUG
                    printf("normal %s %c %d :: (%d,%d)=%c\n", nome, comando, prox,
state, nnum, comando);
                    #endif
                    poetab_n(state, nnum, prox);
                    poetab_a(state, nnum, comando);
                } else if (sscanf(linha, "%s %c", nome, &comando) == 2) {
                    int nnum = pegarnomenum(nome);
                    #ifdef DEBUG
                    printf("aceite %s %c\n", nome, comando);
                    #endif
                    poetab_a(state, nnum, comando);
                } else {
                    printf("erro\n");
                }
```

```
                }
                break;
            case ACABOU:
                break;
        }
        //printf("%s", linha);
    }


    fclose(f);
}

char entrada[] = "-77 -99 -77 -105 0";
const char *sentrada;
int look = -1;

int lerentrada() {
    if (look == 0) {
        return look;
    }
    if (scanf("%d", &look) != 1) {
        look = 0;
    }
    return look;
#if 0
    while(isspace(*sentrada)) {
        sentrada++;
    }
    if (*sentrada == '\0') {
        look = 0;
        return look;
    }
    int num;
    sscanf(sentrada, "%d", &num);
    while(*sentrada == '-' || (*sentrada >= '0' && *sentrada <= '9')) {
        sentrada++;
    }
    look = num;
    return look;
#endif
}

void vai() {
    std::stack<int> pilha;

    sentrada = entrada;
    int num;
    pilha.push(0);
    lerentrada();


    int vezes = 11;

    while(true) {
        #ifdef DEBUG
        printf("%d\n", look);
```

```
#endif

if (look == -1) {
    printf("erro no lex\n");
    break;
}

int topo = pilha.top();

#ifdef DEBUG
printf("tenho pilha = %d fita = %d\n", topo, look);
#endif

int comando = pegar_a(topo, look);
int numtab = pegar_n(topo, look);

#ifdef DEBUG
printf("achei %c (%d)\n", comando, comando);
printf("achei %d\n", numtab);
#endif

if (comando == 's') {
    pilha.push(look);
    lerentrada();
    pilha.push(numtab);
} else if (comando == 'r') {
    int popcount = tamregras[numtab] * 2;
    #ifdef DEBUG
    printf("vou dar pop de %d\n", popcount);
    printf("tamanho pilha = %d\n", (int)pilha.size());
    #endif

    for (int i = 0; i < popcount; i++) {
        pilha.pop();
    }

    int estadoolhar = pilha.top();
    int ladoesq = regraesq[numtab];
    //pilha.push(
    int comando2 = tabela_a[estadoolhar][ladoesq];
    int numtab2 = tabela_n[estadoolhar][ladoesq];


    #ifdef DEBUG
    printf("vou olhar em linha=%d coluna=%d\n", estadoolhar, ladoesq);
    printf("2achei %c (%d)\n", comando2, comando2);
    printf("2achei %d\n", numtab2);
    #endif


    pilha.push(ladoesq);
    pilha.push(numtab2);
} else {
    if (comando == 'a') {
        printf("aceitou\n");
    } else {
```

```
                printf("erro\n");
            }
            break;
        }
        //vezes--;
        //if (vezes <= 0) {
        //    break;
        //}
    }

}

int main(void) {
    abrarq();
    vai();

    return 0;
}
```

Gramática

```
"Start Symbol" = <Linhas>

!==== Regras ====
<TK_si>          ::= -7
<TK_puis>        ::= -6
<TK_sinon>       ::= -10
<TK_pendant>     ::= -16
<TK_faire>       ::= -20
<TK_paire>       ::= -24
<TK_ensemble>    ::= -31
<TK_flottant>    ::= -38
<TK_lettre>      ::= -43
<TK_mat>         ::= -45
<TK_retour>      ::= -67
<TK_true>        ::= -70
<TK_nalse>       ::= -74
<TK_id>          ::= -77
<TK_vezes>       ::= -78
<TK_div>         ::= -79104
<TK_mais>        ::= -80
<TK_menos>       ::= -90
<TK_igual>       ::= -91
<TK_diferente>   ::= -93
<TK_resto>       ::= -94
<TK_menor>       ::= -9597
<TK_maior>       ::= -9698
<TK_doispontos>  ::= -99
<TK_abreparen>   ::= -100
<TK_fechaparen>  ::= -101
<TK_and>         ::= -102
<TK_or>          ::= -103
<TK_comentario>  ::= -105
<TK_menorigual>  ::= -106
<TK_maiorigual>  ::= -107

<Linhas>         ::= <Afirmacao> <TK_comentario> <Linhas>
                   | <Afirmacao> <TK_comentario>

<Afirmacao>      ::= <TK_id> <TK_doispontos> <Expression>
                   | <TK_si> <Expression> <TK_puis> <Linhas> <TK_paire>
```

```
                | <TK_si> <Expression> <TK_puis> <Linhas> <TK_sinon> <Linhas>
<TK_paire>
                | <TK_pendant> <Expression> <TK_faire> <Linhas> <TK_paire>
                | <TK_ensemble> <TK_id>
                | <TK_flottant> <TK_id>
                | <TK_lettre> <TK_id>
                | <TK_mat> <TK_id>
                | <TK_retour>
                | <TK_retour> <Expression>

<Expression>  ::= <And_Exp> <TK_or> <Expression>
                | <And_Exp>

<And_Exp>     ::= <Compare_Exp> <TK_and> <And_Exp>
                | <Compare_Exp>

<Compare_Exp> ::= <Add_Exp> <TK_igual>  <Compare_Exp>
                | <Add_Exp> <TK_diferente> <Compare_Exp>
                | <Add_Exp> <TK_maior>  <Compare_Exp>
                | <Add_Exp> <TK_maiorigual> <Compare_Exp>
                | <Add_Exp> <TK_menor>  <Compare_Exp>
                | <Add_Exp> <TK_menorigual> <Compare_Exp>
                | <Add_Exp>

<Add_Exp>     ::= <Mult_Exp> <TK_mais> <Add_Exp>
                | <Mult_Exp> <TK_menos> <Add_Exp>
                | <Mult_Exp>

<Mult_Exp>    ::= <Negate_Exp> <TK_vezes> <Mult_Exp>
                | <Negate_Exp> <TK_div> <Mult_Exp>
                | <Negate_Exp> <TK_resto> <Mult_Exp>
                | <Negate_Exp>

<Negate_Exp>  ::= <TK_menos> <Value>
                | <Value>

<Value>       ::= <TK_abreparen> <Expression> <TK_fechaparen>
                | <TK_id>
                | <TK_true>
                | <TK_nalse>
```

Instruções

Existem duas maneiras distintas de verificar a sequência de palavras

- make test: via arquivo de texto (data/infile)
- make run: aonde o usuário informa os dados pela linha de comando.
- make syn: executa os programas em sequência, primeiro o analisador léxico e depois o sintático.

Para analisar lexicamente uma palavra ou uma sequência delas, o usuário deve digitar os tokens separados por um espaço e apertar a tecla enter. O programa então verifica se cada token pertence ou não à linguagem Ptgrila, caso o token for aceito, o programa escreve na fita de saída o seu estado final correspondente (vide a sessão “Nomenclatura de estados finais”), estes estados são todos negativos a fim de se distinguir dos estados intermediários. Caso o token não for aceito, o estado 0 (estado de erro) será mostrado na fita.

Já para analisar sintaticamente, o programa usa a fita de saída do analisador léxico como entrada de processamento, se afirmação não for aceita, o estado -1 (estado de erro) será mostrado na saída. O estado 0 aqui é usado para marcar o final da entrada.

A entrada pode ser dada das seguintes formas:

```
<afirmação 1> // <afirmação 2> //
```

Ou

```
<afirmação 1> //

<afirmação 2> //
```

As duas barras paralelas (//) delimitam o final da linha da linguagem, ou seja, são utilizadas para separar as instruções que constituem o programa (como o ponto e vírgula em outras linguagens, como o C e C++ por exemplo).

Após analisar os tokens e afirmações, o programa devolve “aceitou” se passar nas duas análises, “erro no lex” se estiver algum erro léxico ou “erro” se ocorrer algum erro na análise sintática.

Para marcar o fim de um bloco de execução, o comando “paire” é usado. Por exemplo:

```
Si (condição)
      <afirmação 1> //
            .
            .
            .
puis (condição)
      <afirmação 2> //
            .
            .
            .

paire //
```

Conclusão

Um analisador precisa se basear em uma linguagem, e esta linguagem pode ter um ou mais scanners diferentes, além disso, cada analisador pode verificar apenas uma única linguagem. Foi preciso então construir primeiramente a linguagem de programação e um analisador léxico, para que assim o analisador sintático fosse implementado.

O analisador sintático é na verdade uma máquina de estado finito que consiste de um buffer de entrada, uma pilha no qual é armazenada uma lista de estados anteriores, uma tabela de próximo estado que indica para onde o novo estado deve se mover e uma tabela de ação que indica uma regra gramatical a aplicar no estado e símbolo atual na entrada de dados.

Esta análise é a segunda etapa do processo de compilação de um código fonte e seu objetivo é derivar uma expressão em uma arvore da expressão, preparando-o para a próxima fase de compilação: a Análise Semântica.