UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Análise Sintática

Relatório 2: Análise Sintática
Construção de Compiladores 2
Profª. Drª. Helena de Medeiros Caseli

César Roberto de Souza 298603
Leonardo Sameshima Taba 298816

São Carlos, 21 de outubro de 2009

1. Introdução

A análise sintática é a segunda etapa na elaboração de um compilador. Neste trabalho, foi desenvolvido um analisador sintático para a linguagem hipotética *ALG*, dando prosseguimento à construção do compilador **getToken();** iniciada no começo desta disciplina.

2. Desenvolvimento

2.1. Visão Geral

O analisador sintático, assim como o analisador léxico, também foi projetado utilizando-se a técnica de Orientação a Objeto. Assim, a classe **AnalisadorSintatico** tornou-se a principal classe responsável pela condução da análise sintática, atuando como um parser da linguagem ALG.

A classe AnalisadorSintatico possui apenas um método principal, denominado **parse()**, que dá inicio à análise sintática. Esse método faz chamadas ao método getToken() do analisador léxico, que retorna os tokens identificados no programa. Após sua execução, eventuais erros presentes e que puderam ser encontrados estarão disponíveis e poderão ser listados através do método **errors()**, que retorna a lista de erros encontrados.

Internamente, a classe AnalisadorSintatico utiliza o modo pânico para tratamento de erros, o que consiste basicamente em associar, a cada ponto de erro, um conjunto de símbolos de sincronização que dirão ao parser onde prosseguir a análise caso um erro seja encontrado. Com o andamento da análise sintática, marcas de sincronização são passadas para as demais regras no momento de sua ativação. Ao se deparar com um erro, o analisador varre a entrada, utilizando o método **varre()**, descartando simbolos até encontrar um dos símbolos de sincronização, de onde então a análise poderá ser continuada.

Para facilitar a programação do modo pânico, os procedimentos foram codificados de forma a evitar estruturas condicionais aninhadas. Tal estratégia possibilita que a análise continue, na maioria das vezes, por mais que vários erros sejam encontrados em cascata.

O ponto de entrada principal nesta segunda versão do programa é a função **mainSintatico**. Nela, criamos uma nova instância da classe AnalisadorSintatico e passamos o texto de entrada como parâmetro de seu construtor. Assim, basta chamar o método **parse()** para efetuar a análise e coletar os erros encontrados iterando sobre a lista de erros disponibilizada através do método **errors()**.

2.2 Decisões de Projeto

Como dito anteriormente, o análisador sintático utiliza o modo pânico. Para facilitar a programação do modo pânico, os procedimentos foram codificados de forma a evitar estruturas condicionais aninhadas, o que é feito utilizando-se uma lógica invertida para construir cada procedimento. Abaixo exemplificamos como isto é feito:

Figura 1. Grafo sintático para regra "programa". O símbolo @ indica que o valor especificado indica a categoria do símbolo esperado.

```
function programa() {

    if (simbolo == "programa") {
        obterSimbolo();

        if (simbolo == "@ident") {
            obterSimbolo();

            if (simbolo == ";") {
                obterSimbolo();

                // Chama a regra "corpo"
                corpo(Seguidores["programa"]);

                if (simbolo == ".") {
                    obterSimbolo();
                }
                else error("Esperado '.' mas encontrado '" + cadeia + "'");
             }
             else error("Esperado ';' mas encontrado '" + cadeia + "'");
        }
        else error("Esperado identificador mas encontrado '" + cadeia + "'");
    }
    else error("Esperado 'programa' mas encontrado '" + cadeia + "'");
}
```

Listagem 1. Exemplo de procedimento (sem varredura de sincronização) para regra "programa"

```
function programa() {

    if (simbolo != "programa") {
        error("Esperado 'programa' mas encontrado '" + cadeia + "'");
    }
    else obterSimbolo();

    if (simbolo != "@ident") {
        error("Esperado identificador mas encontrado '" + cadeia + "'");
    }
    else obterSimbolo();

    if (simbolo != ";") {
        error("Esperado ';' mas encontrado '" + cadeia + "'");
    }
    else obterSimbolo();

    // Chama a regra "corpo"
    corpo(Seguidores["programa"]);

    if (simbolo != ".") {
        error("Esperado '.' mas encontrado '" + cadeia + "'");
    }
    else obterSimbolo();
}
```

Listagem 2. Exemplo de procedimento (sem varredura de sincronização) para regra "programa" evitando aninhamentos condicionais.

Adicionalmente, quando existe uma transição vazia, a primeira coisa a ser feita ao adentrar um procedimento sintatico é verificar se o símbolo atual está contido no conjunto de primeiros ou seguidores da regra. Caso esteja no conjunto de seguidores, retornarmos do procedimento imediatamente.

É importante notar que algumas regras foram agrupadas de forma a melhor aproveitar o código e otimizar a execução. Também foi necessário realizar alterações na gramática original da linguagem ALG para torna-la LL(1). Mais especificamente, a regra 18 foi alterada para:

```
<cmd> ::= le (<variaveis>) |
   escreve (<variaveis>) |
   enquanto <condicao> faca <cmd> |
   se <condicao> entao <cmd> <cont_se> |
   ident <cont_ident> |
   inicio <comandos> fim
```

Além dessa modificação, duas novas produções tiveram de ser adicionadas:

```
<cont_se> ::= fim | senão <cmd>
<cont_ident> ::= := <expressão> | <lista_arg>
```

A seguir, apresentamos o conjunto de símbolos seguidores para cada não terminal:

Não terminal	Seguidores
programa	$
corpo	“.”
dc	“inicio”
dc_v	“procedimento”, “inicio”
tipo_var	“;”, “)”
Variaveis	“:”, “)”
mais_var	“:”, “)”
dc_p	“inicio”
parametros	“;”
lista_par	“)”
mais_par	“)”
corpo_p	“procedimento”, “inicio”
dc_loc	“inicio”
lista_arg	“;”, “fim”, “senao”
argumentos	“)”
mais_ident	“)”
comandos	“fim”
cmd	“;”, “fim”, “senao”
cont_se	“;”, “fim”, “senao”
cont_ident	“;”, “fim”, “senao”
condicao	“faca”, “entao”
relacao	“+”, “-”, identificador, numero inteiro, numero real, “(“
expressao	“faca”, “entao”, “=“, “<>“, “>=“, “<=“, “>“, “<“, “;”, “fim”, “senao”, “)”
op_un	identificador, numero inteiro, numero real, “(“
outros_termos	“faca”, “entao”, “=“, “<>“, “>=“, “<=“, “>“, “<“, “;”, “fim”, “senao”, “)”
op_ad	“+”, “-”, identificador, numero inteiro, numero real, “(“
termo	“faca”, “entao”, “=“, “<>“, “>=“, “<=“, “>“, “<“, “;”, “fim”, “senao”, “+”, “-”, “)”
mais_fatores	“faca”, “entao”, “=“, “<>“, “>=“, “<=“, “>“, “<“, “;”, “fim”, “senao”, “+”, “-”, “)”
op_mul	identificador, numero inteiro, numero real, “(“
fator	“faca”, “entao”, “=“, “<>“, “>=“, “<=“, “>“, “<“, “;”, “fim”, “senao”, “+”, “-”, “*”, “/”, “)”

Esses símbolos seguidores são usados como sincronizadores quando algum erro é encontrado, possibilitando a recuperação e continuação da análise. No entanto, em certos momentos também são usados os conjuntos de primeiros de cada não terminal.

Também são utilizados outros símbolos sincronizadores em pontos específicos para maximizar o número de erros encontrados e prevenir a cascata de erros e perda de informações importantes. Não serão listados todos os símbolos de sincronização extras, pois cada ponto de verificação de terminal checa por símbolos distintos. Por exemplo, em

listas de parâmetros e argumentos, os tokens ",", ";" e identificador são utilizados como sincronizadores, para que declarações de variáveis não sejam perdidas.

Quanto a erros léxicos, após serem relatados como cadeias inválidas, estes são sumariamente descartados para que a análise possa prosseguir a partir do próximo token válido.

2.3 Erros

Em nosso analisador os erros sintáticos encontrados são relatados tão logo ocorram, prevenindo assim problemas como perda do local do erro e maximizando o número de erros encontrados. Os erros detectados em nossa implementação incluem:

- **Falta de tokens**, quando deveria haver um token de determinado tipo em determinada posição. Nesta categoria se encaixam diversos erros diferentes, como falta de "programa", falta de identificador, falta de operador aritmético ou operador relacional, etc;
- **Token supérfluo**, quando há um token que não deveria estar em determinada posição;
- **Erros léxicos**, relatados como caracteres inválidos contidos na entrada.

No entanto, ainda existem alguns erros complicados demais para serem tratados na análise sintática. Estes erros ocorrem quando demasiados tokens são esquecidos ou escritos de forma errada em seguida e torna-se difícil adivinhar o que o usuário realmente queria dizer em seu programa.

3. Compilação e execução

3.1. Compilando

Por JavaScript ser uma linguagem interpretada, não há necessidade de compilação. No entanto, alguns browsers modernos, como por exemplo o Google Chrome, realizam uma compilação Just-in-Time (JIT) do código para tornar sua execução mais ágil. Isso ocorre de maneira transparente para o usuário.

3.2. Executando

A execução do analisador é feita através de um navegador. Portanto, basta clicar duas vezes no arquivo index.html contido na pasta raiz do compilador para iniciá-lo.

Sua interface consiste em apenas duas caixas de texto, uma delas destinada ao código fonte de entrada e a outra dedicada a mostrar a saída desejada. Através de uma caixa drop-down localizada no canto superior esquerdo, é possível escolher o tipo da análise desejada, seja léxica ou sintática. Neste caso, deve ser escolhida a opção **análise sintática** (já selecionada por padrão).

O código fonte a ser processado deve ser colocado na caixa de texto superior. Em seguida, deve-se clicar no botão "Processar" para iniciar a análise. Seus resultados serão então escritos na caixa inferior.

Note que também é possível optar pela execução de alguns exempos pré-determinados, alguns bem formados, segundo a linguagem ALG, e outros contendo erros sintáticos e/ou léxicos.

3.2.1. Exemplos de execução

Entrada	Saída

programa fibonacci;
{ imprime os n primeiros numeros de fibonacci }
 	
var n1, n2, n3, qtde, i : inteiro;
inicio
  n1 := 0;
  n2 := 1;
  i := 0;
  le(qtde);
  enquanto i < qtde faca
  inicio
  escreve(n2);
  n3 := n1 + n2;
  n1 := n2;
  n2 := n3;
  i := i + 1;
  fim;
fim.
	Analise concluida com sucesso

Entrada	Saída

programa exemplo2;
{ testa erros no uso de condicionais }
 	
var a: real;
var b: inteiro;
procedimento nomep(x: real);
  var a, c: inteiro;
  inicio
  le(c, a);
  se a<x+c enta
  inicio
  a:= c+x;
  escreve(a);
  fim
  senao c:= a+x;
  fim;
 	
inicio {programa principal}
  le(b);
  nomep(b);
fim.
	Analise terminada com erros
	Erro na linha 10: Esperado operador matematico mas encontrado 'enta'
	Erro na linha 11: Esperado operador matematico mas encontrado 'inicio'
	Erro na linha 11: Esperado 'entao' mas encontrado 'inicio'

Entrada	Saída

#programa exemplo4;
{ testa inicio de programa, declaracao de
  variaveis e expressoes contendo erros }
 	
var @: real;
var b: inteiro;
procedimento nomep(x: real);
  var a, c: inteiro;
  inicio
  le(c, a);
  se a<x+c entao
  inicio
  a:= c+x;
  escreve(a);
  fim
  senao c:=_ a+x;
  fim;
 	
inicio {programa principal}
  le(b);
  nomep(b);
fim.
	Analise terminada com erros
	Erro na linha 1: Caractere '#' nao reconhecido
	Erro na linha 5: Caractere '@' nao reconhecido
	Erro na linha 5: Esperado identificador mas encontrado ':'
	Erro na linha 16: Caractere '_' nao reconhecido