

INE5426 AL e AS1

João Gabriel Trombeta

Mathias Olivion Reolon

Otto Menegasso Pires

Wagner Braga dos Santos

30 de Abril de 2019

Universidade Federal de Santa Catarina - UFSC

A gramática $X++$ é recursiva à esquerda?

R: Não. A definição de uma produção com recursão à esquerda é: " $\text{expr} \rightarrow \text{expr} + \text{term}$ ", e a gramática do $X++$ não apresenta nenhuma produção desse tipo.

Nota-se de maneira trivial que as produções CLASSDECL, CLASSBODY, VARDECL, CONSTRUCTDECL, METHODDECL, METHODBODY, PARAMLIST, PRINTSTAT, READSTAT, RETURNSTAT, SUPERSTAT, IFSTAT, FORSTAT, LVALUE, ALOCEXPRESSION, FACTOR não são recursivas à esquerda porque em todos os casos a produção mais a esquerda é um terminal.

Para os outros casos é necessário avaliar se não existem recursões indiretas.

AS1 Questão 1

PROGRAM não é recursivo pois sua produção leva para CLASSLIST, que leva para CLASSDECL, que não é recursiva.

ATRIBSTAT não é recursivo pois sua produção leva para LVALUE, que não é recursiva.

STATEMENT não é recursivo pois suas produções levam para não-terminais que também não possuem recursão.

STATLIST não é recursivo pois sua produção leva para STATEMENT, que não é recursivo.

EXPRESSION não é recursivo pois sua produção leva para NUMEXPRESSION, que leva para TERM, que leva para UNARYEXPR, que leva para FACTOR, que produz (EXPRESSION), o que não caracteriza como recursão já que existe o terminal (antes de EXPRESSION.

ARGLIST não é recursivo pois sua produção leva para EXPRESSION.

A Gramática X++ está fatorada a esquerda? Se não, fatore.

R: Não, pois existem produções como ALOCEXPRESSION, que ao serem expandidas geram coisas como 'new ident (ARGLIST)' | 'new int ...' | 'new string ...' | 'new ident ...'.

Pode-se perceber que todas as transições a partir de ALOCEXPRESSION se iniciam com 'new', sendo que duas delas se iniciam com 'new ident'.

Fatoramos ALOCEXPRESSION gerando as seguintes produções:

ALOCEXPRESSION \rightarrow new ALOCEXPRESSION2 ALOCEXPRESSION2
 \rightarrow ident ALOCEXPRESSION3 | CTYPE [EXPRESSION]
([EXPRESSION])* ALOCEXPRESSION3 \rightarrow (ARGLIST) | [
EXPRESSION] ([EXPRESSION])* CTYPE \rightarrow string | int

Desse modo, para cada produção, não existem duas transições que se iniciam com os mesmos terminais ou não terminais.

A gramática X++ é LL(3). Por quê?

R: Por causa das produções VARDECL e METHODDECL.

VARDECL: Essa produção se inicia com o tipo da variável (int/string/ident). Depois tem um identificador que representa o nome da variável e depois pode-se ter ou um '[' ou uma ',' ou um ';'. Dependendo do terceiro símbolo pode-se ter um ']', completando o '[]', um identificador, declarando uma nova variável de mesmo tipo, ou o fim da produção.

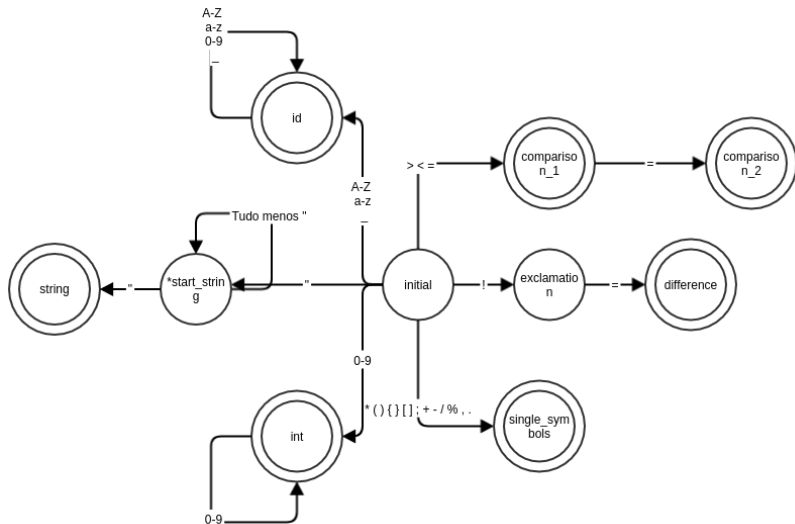
METHODDECL: inicia-se com o tipo do método (int/string/ident), depois pode ou não vir uma sequência de '[]' e depois vem o nome do método, em forma de ident e finalmente o corpo do método que começa com um '('.

Por causa disso, para se ter certeza que a produção corresponde ou não a um VARDECL é preciso olhar 3 símbolos para frente. Por exemplo: Caso os primeiros dois símbolos sejam “int ident” não é possível saber se é um método com o nome ident ou uma variável. Mas ao olhar o terceiro símbolo se for ou '[', ou ',', ou ';' então pode-se concluir que se trata de uma variável.

A análise é feita utilizando um autômato finito terminístico que reconhece os padrões no código. O autômato é descrito em `automata.txt`, a primeira linha são os estados, a segunda os símbolos, a terceira o estado inicial, a quarta o final, e as demais as transições, que devem obedecer o padrão estado -> destino símbolo.

Para facilitar a implementação do lexema pertencente ao token `string_const`, caso o estado comece com o caracter * o autômato irá permanecer no mesmo estado caso ele não possua uma transição pelo símbolo lido, ao invés retornar um erro.

AL - Descrição dos estados



Instancia o analisador léxico com a descrição do autômato e pede o arquivo a ser analisado. O analisador também recebe o conjunto de palavras que são palavras reservadas e tipos básicos da linguagem. O analisador então é utilizado para gerar todos os tokens.

O construtor abre o arquivo que descreve o AFD e cria o objeto Automata. Ele também abre o código a ser lido e deixa como seu atributo.

Os tokens são gerados ao chamar `get_next_token`, cada chamada ele retorna o próximo token e quando o código acabar ele retorna `$`. O método ignora brancos e espaçamentos e faz com que o autômato rode sobre o arquivo que tem como atributo. Como retorno, a função `run` do autômato informa o estado em que parou, se é de aceitação, e quantos símbolos foram lidos antes de parar nesse estado. Tendo a informação de quantos símbolos foram lidos e o código, o analisador consegue ver o que foi lido e cria o token apropriado, colocando-o na tabela de símbolos caso necessário. Depois ele atualiza o código de forma a remover o que já foi lido e retorna o token.

Representa o autômato finito. O método `run` recebe uma string de entrada e roda o AFD sobre ela. Caso exista uma transição pelo símbolo lido o autômato faz a transição e continua o processo até o momento em que lê algo inválido. Quando algo inválido é lido o AFD ignora essa leitura e retorna o estado atual, se ele é final e quantos símbolos foram lidos. Isso acontece porque, por exemplo, tomando a entrada " ";. " " é uma entrada válida que leva o autômato ao estado de aceitação `string`, mas quando o símbolo ; for lido isso resultaria em um erro, uma vez que do estado `string` não existe mais transições. Então ; é ignorado e o autômato retorna que leu uma string. Quando o método for chamado novamente o AFD estará no estado inicial e conseguirá reconhecer ; como um padrão válido.