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

AS1 Questão 1

A gramática X++ é recursiva à esquerda?

R: Não. A definição de uma produção com recursão à esquerda é: "expr -> expr + term", e a gramática do X++ não apresenta nenhuma produção desse tipo.

AS1 Questão 1

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.

AS1 Questão 2

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'.

AS1 Questão 2

Fatoramos ALOCEXPRESSION gerando as seguintes produções:

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

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

AS1 Questão 2

<program> -> <class_decl> | &

<class_list> -> <class_decl> | &

<class_decl> -> class ident <extend> <class_body> <class_list>

<extend> -> extends ident | &

<class_body> -> <class_list> <class_body2>

<class_body2> -> <type> <class_body3> | constructor
<method_body> <construct_decls> <method_decls> | &

<class_body3> -> ident <class_body4> | [ ] <vector> ident
<method_body> <method_decls>

<class_body4> -> <method_body> <method_decls> | <vector>
<extra_var> ; <class_body2>

<var_decls> -> <type> <var_decl> <var_decls> | &

<var_decl> -> ident <vector> <extra_var>

AS1 Questão 2

<extra_var> -> , ident <vector> | &

<construct_decls> -> <construct_decl> <construct_decls> | &

<method_decls> -> <method_decl> <method_decls> | &

<var_or_atrib> -> <var_decl> | <atrib_stat>

<type> -> <ctype> | ident

<ctype> -> int | string

<mdm_op> -> * | / |

<rel_op> -> > | < | <= | >= | == | !=

<pm_op> -> + | -

<pm_op?> -> <pm_op> | &

<vector> -> [ ] <vector> | &

<construct_decl> -> constructor <method_body>

AS1 Questão 2

<method_decl> -> <type> <vector> ident <method_body>

<method_body> -> (<param_list>) <statement>

<param_list> -> <type> ident <vector> <extra_param> | &

<extra_param> -> , <type> ident <vetor> <extra_param> | &

<statement> -> <ctype> <var_decl> ; | ident <var_or_atrib> ; | <print_stat> | <read_stat> | <if_stat> | <for_stat> | <stat_list> | break ; | ;

<read_stat> -> read ident <lvalue>

<print_stat> -> print <expression>

<atrib_stat> -> <lvalue> = <expr_or_aloc>

<expr_or_aloc> -> <expression> | <aloc_expression>

<expression?> -> <expression> | &

<return_stat> -> return <expression?>

AS1 Questão 2

<else> -> else <statement> | &

<if_stat> -> if (<expression>) <statement> <else>

<for_stat> -> for (<atrib_stat?> ; <expression?> ; <atrib_stat?>) <statement>

<stat_list> -> <statement> <stat_list> | &

<lvalue> -> [<expression>] <lvalue> | . ident <arg_list?> <lvalue> | &

<[expression]*> -> [<expression>] <[expression]*> | &

<aloc_expr2> -> ident <aloc_expr3> | <ctype> [<expression>] <[expression]*>

<aloc_expr3> -> (<arg_list>) | [<expression>] <[expression]*>

<aloc_expression> -> new <aloc_expr2>

<opt_expression> -> <rel_op> <num_expression> | &

AS1 Questão 2

<expression> -> <num_expression> <opt_expression>

<expression2> -> , <expression> <expression2> | &

<opt_term> -> <pm_op> <term> <opt_term> | &

<num_expression> -> <term> <opt_term>

<unary_expr> -> <pm_op?> <factor>

<unary_vazio> -> <mdm_op> <unary_expr> <unary_vazio> | &

<term> -> <unary_expr> <unary_vazio>

<factor> -> int-const | string-const | null | ident <lvalue> | (<expression>)

<arg_list> -> <expression> <expression2> | &

<atrib_stat?> -> <atrib_stat> | &

<arg_list?> -> (<arg_list>) | &

AS1 Questão 3

A gramática X++ é LL(3). Por quê?

R: Por causa das produções VARDECL e METHODDECL.

AS1 Questão 3

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 '('.

AS1 Questão 3

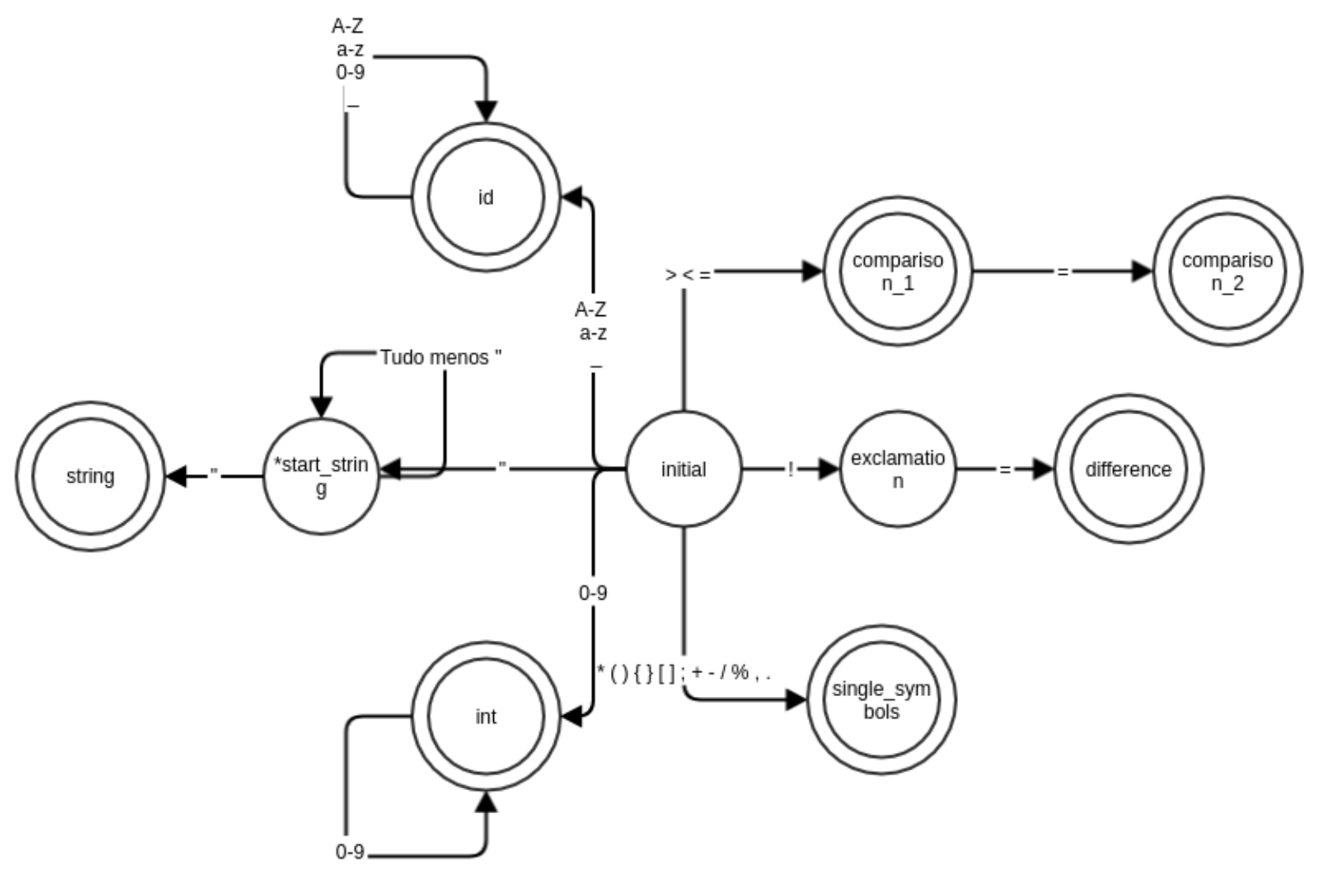
Por causa disso, para se ter certeza que a produção corresponde ou não a um VARDECL é preciso olhar 3 símbolos para frente. Por exemplo: Caso os primeiros dois símbolos sejam “int ident” não é possível saber se é um método com o nome ident ou uma variável. Mas ao olhar o terceiro símbolo se for ou ’[’,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

AL - AFD

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



AL - compiler.py

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.

AL - `lexical_analizer.py`

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 símbolosform 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.

AL - automata.py

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 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 pro 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.