Documentação do analisador léxico

Lucas Amaral [1]

[1]Departamento de Ciência da Computação - Universidade de Brasília

1. Descrição do trabalho

Este trabalho consiste na implementação do analisador léxico utilizando a ferramenta Flex. Uma pequena modificação da gramática foi feita em relação à anteriormente apresentada. Mais especificamente, as diretrizes de entrada e saída foram adicionadas, como vai ser exibido posteriormente.

2. Visão geral do projeto

Foi utilizada a linguagem C++ para o programa, principalmente pelo fornecimento de estruturas de dados prontas em suas bibliotecas.

O programa irá receber, por linha de comando, o arquivo a ser escaneado. Cada token será lido e terá seu nome correspondente impresso na tela. Quebras de linha no arquivo fonte são replicadas na saída. Caso algum erro ocorra, este será guardado e só será notificado no final da execução do programa. É utilizada uma estrutura em uma lista para armazenamento dos erros.

A função *handle_token(int token)* é utilizada para fazer o tratamento de um token lido, imprimindo na tela os adequados e adicionando à lista de erros caso algo de incorreto seja encontrado. Ela consiste, basicamente, de um vetor contendo os tokens como strings, e *if-then-else* com um tratamento apropriado para cada token. Quando um token não identificado é lido, o código seguinte é executado. Nele *comp_error_t* é uma estrutura que representa um erro, contendo um tipo de erro e uma mensagem a ser mostrada. A lista de erros é chamada de *error_list*. A linha e coluna onde o token inválido é encontrado são adicionadas no início da mensagem.

```
1     std::string aux = "\n" ;
2     aux += to_string(line);
3     aux += ":";
4     aux += to_string(col);
5     aux += " Token não reconhecido \"";
6     aux += yytext;
7     aux += "\"";
8     error_list.push_back(comp_error_t(0, aux));
```

Este tratamento foi escolhido por dois motivos:

- Organização da saída do programa
- Reaproveitamento da estrutura de tratamento de erros para os próximos trabalhos

Vale ressaltar que o único tipo de erro tratado nesta fase é o erro estritamente léxico. Apenas tokens inválidos são identificados como um erro por este analisador.

Foi incluida também uma função *main* para realização da do recebimento do nome do arquivo por linha de comando, chamada à função específica do parser e impressão dos resultados.

2.1. Segundo exemplo

```
sumBelow10 :: [Integer] -> Integer;
sumBelow10 [] = 0;
sumBelow10 xs = do {
    sum <- 0;
    while(xs /= []) {
        x:xs <- xs;
        aux  <- if (x < 10) then {
                    yield x;
                } else {
                    yield 0;
                };
        sum <- sum + aux;
    };
    sum;
}
```

Pontos a observar:

- Expressões possuem valores
- Função declarada imperativamente retorna o último valor computado.

Aqui, vemos que o valor da expressão *if-then-else* está sendo atribuída à uma variável *aux*, diferente da expressão while, que não tem seu valor atribuído a nada. Isto se deve ao uso da palavra-chave *yield*, que faz com que estas estruturas tenham retornem um valor no contexto imperativo. No caso do *if*, ambas as branches precisam de um *yield*, e o tipo deve ser compatível. Caso o *yield* não seja usado, como no caso do *while*, a expressão retorna o valor unitário, "()".

Vale ressaltar que o *yield* não funciona da mesma maneira que o *return* das linguagens tradicionais, pois não pode ser usado para terminar a execução no meio de um bloco. Diferentemente, ele é opcional, e só pode ser usado como última instrução de uma estrutura de loop/condição. Vemos também que a função retorna o valor de *sum*, pois é a última linha a ser executada.

O uso de estruturas imperativas e modificação de variáveis internas não afeta a *previsibilidade* da função. Para os mesmo valores de entrada, ela sempre retorna o mesmo resultado.

3. Gramática da linguagem

Abaixo encontra-se a gramática para a linguagem. Alguns detalhes importantes:

- O uso de ';' para delimitar extensão das declarações de função e statments no caso imperativo.
- O uso das variáveis do tipo *op_precN*. Nelas, *N* representa o nível de precedência do operador, em ordem crescente.
- Notar também a variável *procdecl*, referente à declaração procedural de função.

```
program:
    line_elems

line_elems:
    line_elem line_elems
```

```
6       | line_elem
7
8   line_elem:
9       fundecl
10      | procdecl
11      | funtype_decl
12
13  fundecl:
14      ID args '=' expr ';'
15      | ID args '=' expr where_exp
16      | ID '=' expr ';'
17      | ID '=' expr where_exp
18
19  args:
20      arg_value args
21      | arg_value
22      | WILDSCORE
23
24  arg_value:
25      list_value
26      | basic_value
27      | '(' arg_value ')'
28
29  basic_value:
30      INT
31      | FLOAT
32      | TRUE
33      | FALSE
34      | ID
35      | '(' ')'
36
37  list_value:
38      arg_value ':' list_value
39      | built_list_value
40
41  built_list_value:
42      '[' ']'
43      | '[' args ']'
44
45  funtype_decl:
46      ID '::' funtype ';'
47
48  funtype:
49      basic_type
50      | '(' funtype ')'
51      | basic_type '->' funtype
52
53  basic_type:
54      INTEGER
55      | FLOAT_T
56      | BOOL
57      | '[' types ']'
58      | ID
59      | '(' ')'
60
61  types:
```

```
62      basic_type
63      | basic_type ',' types
64
65  expr:
66      op_prec1
67      | appexp
68      | ifexpr
69      | yieldexpr
70
71  ifexpr:
72      'if' expr 'then' '{' expr '}' 'else' '{' expr '}'
73
74  yieldexpr:
75      "yield" ifexpr
76      | "yield" appexpr
77      | "yield" op_prec1
78
79
80  op_prec1:
81       op_prec2 '||' op_prec1
82       | op_prec2
83
84  op_prec2:
85       op_prec3 '&&' op_prec2
86       | op_prec3
87
88  op_prec3:
89       op_prec3 '==' op_prec4
90       | op_prec3 '/=' op_prec4
91       | op_prec3 '<' op_prec4
92       | op_prec3 '<=' op_prec4
93       | op_prec3 '>' op_prec4
94       | op_prec3 '>=' op_prec4
95       op_prec4
96
97  op_prec4:
98       op_prec5 ':' op_prec4
99       | op_prec5 '++' op_prec4
100      | op_prec5
101
102 op_prec5:
103     op_prec5 '+' op_prec6
104     | op_prec5 '-' op_prec6
105     | op_prec6
106
107 op_prec6:
108      op_prec6 '%' op_prec7
109      | op_prec7
110
111 op_prec7:
112     op_prec7 '*' op_prec8
113     | op_prec7 '/' op_prec8
114     | op_prec8
115
116 op_prec8:
117     basic_value
```

```
118     | list_expr
119     | '(' expr ')'
120     | '-' expr
121
122 list_expr:
123     | '[' exprs ']'
124     | '[' ']'
125
126 exprs:
127     expr
128     | expr ',' exprs
129
130 appexp:
131     ID expr
132
133 where_exp:
134     'where' '{' line_elems '}'
135
136 procdecl:
137     ID args '=' 'do' '{' stmts '}'
138     | ID args '=' 'do' '{' stmts '}' where_exp
139     | ID '=' 'do' '{' stmts '}'
140     | ID '=' 'do' '{' stmts '}' where_exp
141
142 stmts:
143     stmt
144     | stmt stmts
145
146 stmt:
147     basic_value '<-' expr ';'
148     | basic_value '<-' while_expr ';'
149     | expr ';'
150     | while_expr ';'
151
152 while_expr:
153     'while' '(' expr ')' '{' stmts '}'
```

4. Referências Bibliográficas

[1] The Haskell 98 Report - https://www.haskell.org/onlinereport/index.html [1]

[1] Utilizado como referência para estruturas da linguagem Haskell e predecências de operadores.