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. Analisador Léxico

2.1. Visão geral do projeto

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.

2.2. Tratamento de 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 *list_error_t*. A linha e coluna onde o token inválido é encontrado são adicionadas no início da mensagem. Nele, *make_error* recebe como parâmetro o tipo de erro (léxico, sintático, etc), o token que causou o erro e a linha e coluna onde o erro foi identificado.

```
1    comp_error_t* err = make_error(0, yytext, line, col);
2    add_error(&error_list_root, err);
```

As funções *make_error* e *add_error* são implementadas em um arquivo separado, chamado *list_error*. A primeira aloca uma struct do tipo comp_error_t e inicializa com os parâmetros, retornando o ponteiro, e a segunda adiciona um erro em uma lista dada no primeiro argumento.

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.

Exemplos de erros léxicos são números seguidos de letras, como “954asd”, identificadores iniciados com underscore, como “_func”, e caracteres não aceitos na linguagem, como “ç”.

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.

3. Sobre a gramática da linguagem

No final deste relatório encontra-se a gramática para a linguagem. Em relação à entrega anterior, houveram apenas duas mudanças. A primeira foi a adição das diretivas de entrada e saída. Mais especificamente, as seguintes regras foram adicionadas.

```
1 stmt:
2       | io_stmt ';'
3
4 io_stmt:
5       | 'readInt'
6       | 'readFloat'
7       | 'readBool'
8       | 'print' expr
```

A segunda foi a modificação das regras para constantes booleanas na gramática, juntando ambas em um único token. Seu valor vai ser retornado internamente no token. No código abaixo, onde existe “BOOLVAL”existiam duas regras, “TRUE”e “FALSE”.

```
1 basic_value:
2       INT
3       | FLOAT
4       | BOOLVAL
5       | ID
6       | '(' ')'
```

O resto da gramática continua exatamente como dito no primeiro relatório.

4. Descrição dos arquivos de teste

Existem 4 arquivos de teste contidos na pasta *testes*.

- O arquivo *correto1.txt* mostra um programa simples da linguagem. Observar os operadores e os identificadores com números.
- O arquivo *correto2.txt* contém mais alguns exemplos, como constantes booleanas, comentários e erros sendo ignorados dentro dos comentários, em especial, identificador começando com underscore.
- O arquivo *incorreto1.txt* já contém alguns erros. Identificadores começando com underscore e caracteres inválidos estão presentes.
- O arquivo *incorreto2.txt* também contém alguns erros. Em especial vale notar que # não inicia um comentário, *x - -* não retorna dois tokens, mas três. Identificadores não podem começar com números e nem underscores.

5. Dificuldades encontradas

Não foram encontradas muitas dificuldades no desenvolvimento do trabalho. A única que vale a pena ser mencionada é a identificação de erros que o analisador não percebe imediatamente. Um exemplo é identificar números seguidos de letras como um erro léxico. O analisador léxico retorna dois tokens, <T_NUM>e <T_ID>, ao invés de cair na regra de token não identificado. Para tratar este tipo de erro, uma regra foi adicionada para cada um.

6. 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.

7. Anexo: Gramática

```
program:
    line_elems

line_elems:
    line_elem line_elems
    | line_elem

line_elem:
    fundecl
    | procdecl
    | funtype_decl

fundecl:
    ID args '=' expr ';'
    | ID args '=' expr where_exp
    | ID '=' expr ';'
    | ID '=' expr where_exp

args:
    arg_value args
    | arg_value
    | WILDSCORE

arg_value:
    list_value
    | basic_value
    | '(' arg_value ')'

basic_value:
    INT
    | FLOAT
    | TRUE
    | FALSE
    | ID
    | '(' ')'

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

```
150     | basic_value '<-' io_stmt ';'
151     | expr ';'
152     | while_expr ';'
153     | io_stmt ';'
154
155 io_stmt:
156     | 'readInt'
157     | 'readFloat'
158     | 'readBool'
159     | 'print' expr
160
161 while_expr:
162     'while' '(' expr ')' '{' stmts '}'
```