Trabalho Prático 3
Compilador para a Linguagem P
Parte 1 - Análise Léxica

Arthur Souto Lima

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

arthursl@ufmg.br

1. Introdução

O projeto consiste em implementar um compilador completo para a linguagem P descrita na documentação. Essa tarefa deve ser feita em partes, sendo a primeira delas implementar um analisador léxico com auxílio do Lex. Esse programa deve ser capaz de receber um programa fonte na linguagem P e dividi-lo em tokens. Nesse processo, são impressos na tela os tokens e algumas informações básicas acerca de cada um.

2. Instruções

2.1. Compilação

O analisador léxico foi desenvolvido usando o Lex (no caso o Flex) e as funções auxiliares, bem como o leitor da entrada, foram implementados em C. O programa acompanha um makefile para compilação e execução. Basta executar o comando *make* para compilar o programa. O executável final se chama *yylex* e estará na pasta raiz do programa (onde está também o makefile).

O makefile, nessa ordem, compila o analisador léxico por meio do Lex e então usa o scanner gerado por ele (arquivo *lex.yy.c*) para compilar o programa que lê o arquivo de entrada. Ato contínuo, ele já executa os três arquivos de teste.

2.2. Execução

Compilado o analisador léxico e seu scanner, ou seja, já foi gerado o arquivo *yylex*, deve-se rodar o executável *yylex* e o código-fonte, por exemplo, no terminal, via o comando

./yylex < teste1.in

Assim, o yylex lerá o programa fonte, no caso do exemplo, o *teste1.in*, da entrada padrão e encaminhará o que for lido para que o analisador léxico faça sua tarefa. A medida que forem encontrando tokens, será impresso na tela a linha deste, seu tipo e seu valor, se for o caso. O programa encerra sua execução automaticamente com o fim do arquivo de entrada.

3. Implementação

3.1. Resumo do Projeto

O analisador léxico foi implementado com base nas aulas acerca do assunto, bem como das seções relevantes do livro-texto [Aho et al. 2007] e também no tutorial [Niemann].

Assim, a criação do arquivo Lex foi bem direta. Um main foi necessário para podermos ler o programa fonte e este foi adaptado dos exemplos iniciais de [Niemann].

No arquivo *tokens.h* estão definidos os números referentes a cada token que será retornado pelo analisador léxico. Por agora, esses números não tem funcionalidade além de servir de retorno. O módulo *aux* contém funções auxiliares, que ora estariam na terceira parte do arquivo Lex, incluindo uma função para imprimir na tela o token encontrado, seu tipo e sua linha e funções para reconhecimento de identificadores.

Na implementação, o casamento de identificadores e palavras reservadas é um dos últimos a ser feito, para que, por exemplo, as constantes booleanas, embora possuam o padrão de um identificador, sejam casadas pelo analisador léxico do Lex com o padrão *true* ou *false* bem antes de tentar casar como um identificador de fato.

A gramática passada possui algumas palavras reservadas, seja por serem comandos, como *if* ou *while*, seja por serem funções-padrão, como *sin* e *eoln*. Assim, seguindo as orientações de [Niemann], temos um array com os comandos e outro com as funções. Para cada possível identificador, isto é, um token que casou com o padrão de identificador, verificamos se está em cada um dos arrays (via funções auxiliares e a função de C *strcmp*). Se estiver em algum deles, retornamos o índice deles em seus respectivos arrays. Se não, retornamos o token de identificador. Reiterando, esse retorno não tem função neste momento.

3.2. Questões não solucionadas

Tendo em vista que essa parte inicial do projeto lida apenas com a análise léxica, muitas questões seguem em aberto e serão resolvidas ao longo do desenvolvimento.

A primeira delas é que o analisador léxico não conhece não conhece as regras e produções da gramática, ele apenas consegue detectar se uma palavra é reservada ou se é uma função, mas não consegue (nem precisa) encaixar um padrão de um comando complexo como um *if-then-else*. Isso é tarefa do analisador sintático.

Outra é que por desconhecer essas regras, ele não sabe da existência do operador menos unário. Assim, para o analisador léxico, todo sinal ”-” é uma subtração (operador binário). O analisador sintático, novamente, é quem lida com essa situação.

Mais uma consequência desse fato é que a gramática permanece inalterada, já que o analisador léxico não sofre com eventuais conflitos e ambiguidades que podem existir na gramática proposta.

Finalmente, como é apenas um analisador léxico, etapas seguintes da compilação ainda estão por ser feitas nas próximas partes do projeto, como análise sintática, criação e preenchimento da tabela de símbolos, análise semântica e geração de código intermediário.

3.3. Suposições Feitas

Nessa etapa, passou-se a regra gramatical que identificava o que era uma constante booleana para as definições léxicas, já que todas as outras constantes ali estavam definidas. Isso propiciou que o analisador léxico também fosse capaz de identificar o token de uma constante desse tipo, além das outras colocadas nas convenções léxicas.

4. Exemplos e Testes

A seguir estão várias entradas para o analisador léxico, acompanhadas do ”hardcopy” do terminal, tendo em vista que a saída dele é ali. Os dois primeiros testes são testes sintéticos, ou seja, não são efetivamente um programa na linguagem P, apenas tokens para serem reconhecidos. O terceiro e último teste é um programa na linguagem referida.

Reiterando o que foi descrito nas instruções, esses três testes são automaticamente executados pelo makefile após a compilação do programa, ou seja, após o comando *make*.

4.1. Teste 1: Constantes e Identificadores

No arquivo *teste1.in* estão apenas constantes e identificadores, um por linha, podemos ver na saída cada um deles. O caractere inválido da linha 14 e na 15 é o ”_” (underline), pois tentou-se definir um identificador ”*_var2*” e outro ”*var1_*”. Como a convenção léxica da especificação não permite símbolos especiais nos identificadores, o analisador léxico descartou o *underline*, mas ainda assim reconheceu um identificador válido.

```
./yylex < teste1.in
[   1] Real:    1.1
[   2] Integer:         10
[   3] Real:    793.003E-0053
[   4] Real:    793.003E+0053
[   5] Char:    ' '
[   6] Char:    'a'
[   7] Identifier:      ttttrue
[   8] Bool:    true
[   9] Identifier:      trueaaaa
[  10] Identifier:      tttfalse
[  11] Bool:    false
[  12] Identifier:      falseaaa
[  13] Identifier:      var1
Caractere ou token inválido
[  14] Identifier:      var2
[  15] Identifier:      var1
Caractere ou token inválido
```

4.2. Teste 2: Operadores e Palavras Reservadas

No arquivo *teste2.in* temos algumas expressões, com operadores e operandos, além de algumas palavras reservadas nas linhas posteriores, como podemos ver logo abaixo:

```
12 >= 32E+2
31.01 != 12.001E-5
32 + 31
32 div 32
if then else
cos
program
sen cos log 1
sin
goto eoln
```

A saída do *teste2.in* está a seguir, com cada um dos tokens identificados, sua linha respectiva e seu valor (ou instância).

```
./yylex < teste2.in
[   1] Integer:         12
[   1] Relop:   >=
[   1] Real:    32E+2
[   2] Real:    31.01
[   2] Relop:   !=
[   2] Real:    12.001E-5
[   3] Integer:         32
[   3] Addop:   +
[   3] Integer:         31
[   4] Integer:         32
[   4] Mulop:   div
[   4] Integer:         32
[   5] Reserved:        if
[   5] Reserved:        then
[   5] Reserved:        else
[   6] Function:        cos
[   7] Reserved:        program
[   8] Identifier:      sen
[   8] Function:        cos
[   8] Function:        log
[   8] Integer:         1
[   9] Function:        sin
[  10] Reserved:        goto
[  10] Function:        eoln
```

4.3. Teste 3: Programa em P

Finalmente, no arquivo *teste3.in* temos um programa simples na linguagem P:

```
program mytest;

x: integer;
z: real

begin
    begin
        x := cos(0);
        z := 0;
        while (x <= 5) do
            z := z + x;
            x := x + 1
        end;
    end
end
```

Como esperado, o analisador léxico apenas identifica os tokens, como é possível ver na saída produzida, nada mais. A questão do reconhecimento das regras e das produções será feita no analisador sintático, próxima etapa do projeto.

```
./yylex < teste3.in
[   1] Reserved:        program
[   1] Identifier:      mytest
[   1] CharEsp:         ;
[   3] Identifier:      x
[   3] CharEsp:         :
[   3] Reserved:        integer
[   3] CharEsp:         ;
[   4] Identifier:      z
[   4] CharEsp:         :
[   4] Reserved:        real
[   6] Reserved:        begin
[   7] Reserved:        begin
[   8] Identifier:      x
[   8] Assign:  :=
[   8] Function:        cos
[   8] CharEsp:         (
[   8] Integer:         0
[   8] CharEsp:         )
[   8] CharEsp:         ;
[   9] Identifier:      z
[   9] Assign:  :=
[   9] Integer:         0
[   9] CharEsp:         ;
[  10] Reserved:        while
[  10] CharEsp:         (
[  10] Identifier:      x
[  10] Relop:   <=
[  10] Integer:         5
[  10] CharEsp:         )
[  10] Reserved:        do
[  11] Identifier:      z
[  11] Assign:  :=
[  11] Identifier:      z
[  11] Addop:   +
[  11] Identifier:      x
[  11] CharEsp:         ;
[  12] Identifier:      x
[  12] Assign:  :=
[  12] Identifier:      x
[  12] Addop:   +
[  12] Integer:         1
[  13] Reserved:        end
[  13] CharEsp:         ;
[  14] Reserved:        end
[  15] Reserved:        end
```

5. Conclusão

Com o fim desta etapa deste trabalho prático, pôde-se colocar em prática os conhecimentos adquiridos nas aulas acerca da construção de um analisador léxico através de uma ferramenta, no caso o Lex. O interessante nessa disciplina foi a possibilidade de experienciar as duas situações: implementar o analisador léxico feito sem tal ferramenta, como foi no Trabalho Prático 2, e, agora, utilizar uma ferramenta para automatizar uma boa parte dessa tarefa. Naturalmente, no contexto de todo o compilador da linguagem P que deve ser feito, ainda há um caminho considerável a ser trilhado até o fim.

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. (2007). *Compilers Principles,Techniques, Tools*. Addison Wesley.

Niemann, T. Lex & yacc tutorial. [1]. Acesso em: 27 jan 2021.

[1] `https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf`