Construção de Compiladores
Aula 2 - Analisador Léxico

Bruno Müller Junior

Departamento de Informática
UFPR

8 de Agosto de 2014

1 Analisador Léxico
2
3 Método 1: Programa
4 Código Analisador Léxico
5 Método 2: Flex
6 Flex - Funcionamento
7 Flex - Estrutura do arquivo .l
8 Flex: Definições
9 Flex: Regras
10 Parte 3 - Subrotinas
11 Parte 3 - Subrotinas
12 Parte 3 - Subrotinas
13 Compilador: flex
14 Bibliografia

Analisador Léxico

- Função: Isolar palavras-chave, símbolos especiais, etc., transformando-os em códigos mais convenientes para outras fases, por exemplo, analisador sintático.
- Quando isoladas, estas palavras-chave são chamados **Tokens**.
- Normalmente é mais conveniente referenciá-los com um nome significativo, chamado **Símbolo**.

```
char token[100];
typedef enum simbolos {  simb_program, simb_identificador,
  simb_numero,  simb_abre_parenteses,  simb_virgula, ... };
simbolos simbolo;
```

- Exemplo. Dada a entrada abaixo, o analisador léxico irá dividir os tokens de entrada em símbolos.

```
program p1 (input, output);
```

Token	Símbolo
"program"	simb_program
"p1"	simb_identificador
"("	simb_abre_parenteses
"input"	simb_identificador
...	

Método 1: Programa

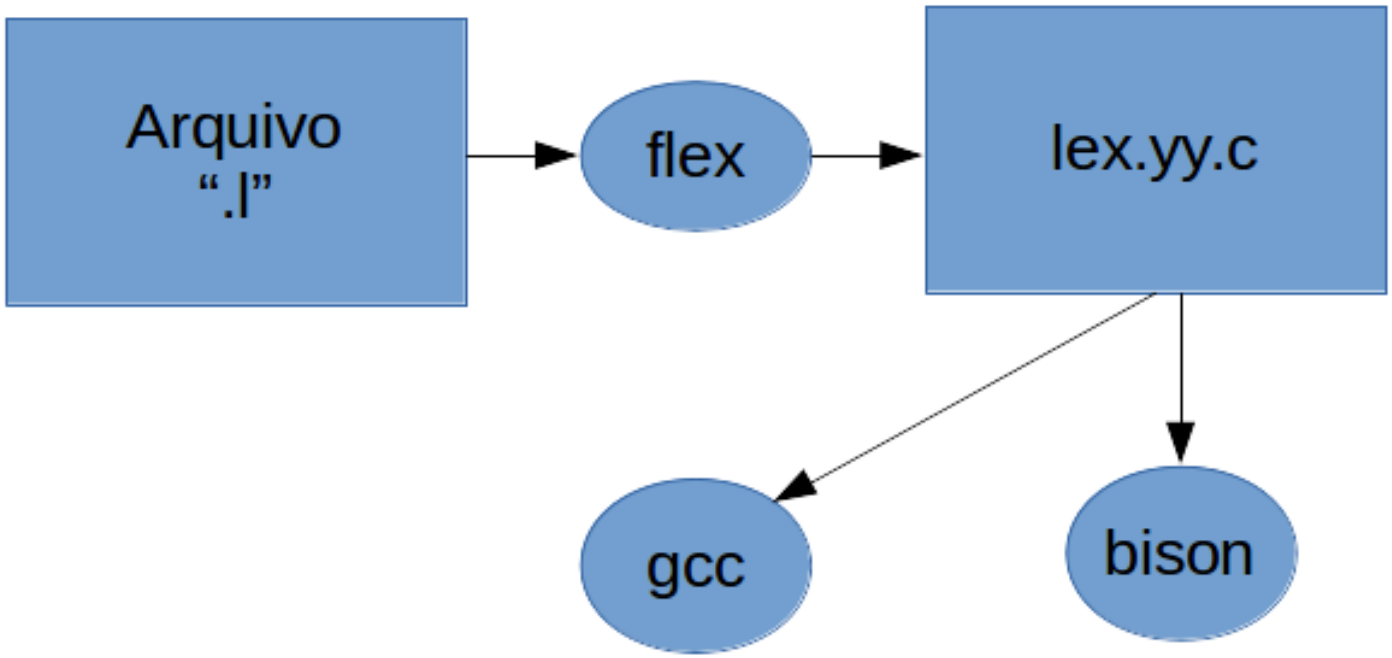
- O A.L. é um "agente passivo", chamado sempre que alguém quiser o próximo token da entrada.
- mantém a posição corrente de leitura;
- um token é definido como o conjunto de caracteres entre dois separadorese (vírgula, ponto, branco, etc.).
- uma forma "natural" de entendê-lo é implementando uma subrotina com assinatura do tipo:
  `Analex(char *token, simbolos *simbolo)`
- A próxima página contém um exemplo de implementação (livro do Tomasz, página 79), onde "átomo" corresponde ao que chamamos de "token".

```
procedimento ANALISADOR_LÉXICO;
início
  átomo:=cadeia_vazia;
  enquanto próximo='□' faça PRÓXIMO;
  se próximo ∈ símbolos_especiais
   então {s:=próximo; PRÓXIMO;
          caso s de
            ':' : se próximo='='
                    então {s:=':=' ; PRÓXIMO};
            '.' : se próximo='.'
                    então {s:='..' ; PRÓXIMO};
            outros: nada
          fim do caso;
          símbolo:=CÓDIGO (s)}
   senão
  se próximo ∈ letras
   então {repita
            átomo:=átomo & próximo; PRÓXIMO
          até próximo ∉ letras_e_dígitos;
          se átomo ∈ palavras_chave
            então símbolo:=CÓDIGO(átomo)
            senão símbolo:=código_de_identificador}
   senão
  se próximo ∈ dígitos
   então {repita
            átomo:=átomo & próximo; PRÓXIMO
          até próximo ∉ dígitos;
          se próximo ∈ letras então ERRO;
          símbolo:=código_de_número}
   senão ERRO
fim
```

Método 2: Flex

- A tarefa que um A.L. faz é essencialmente a de um autômato finito determinístico.
- Existe uma ferramenta, chamada flex, que é capaz de reconhecer os tokens criando um AFD para as palavras indicadas.
- A entrada da ferramenta é um arquivo (normalmente extensão .l) com as regras e gera como saída um programa fonte na linguagem C.
- Ao compilar este programa C, o programa executável será um analisador léxico.

Flex - Funcionamento



Flex - Estrutura do arquivo .l

- Um arquivo de entrada do flex é dividido em três partes.

```
...      } Definições
%%
...      } Regras
%%
...      } subrotinas
```

Flex: Definições

- O que for colocado entre %{ ...%} aqui será copiado no começo do arquivo `lex.yy.c`.
- O que vier a seguir são tratados como #define da linguagem C.

```
%{
#define PROGRAM 255
int conta_linhas=1;
%}
pulo_linha [\n]
ident [a-zAZ][a-zAZ1-9]*
numero [0-9]+
```

Flex: Regras

- Expressões regulares que devem ser comparados com os caracteres de entrada.
- Assim que encontrar o primeiro "match", executa o código associado (entre { ...}) e volta ao início.

```
%%
\+    { return MAIS; }
{pulo_linha} {conta_linhas++;}
program { simbolo  = simb_program;
          strncpy (token, yytext, TAM_TOKEN);
          IMPRIME("program ");
          return PROGRAM;
 }
```

Parte 3 - Subrotinas

- Trecho de código que será copiado para o arquivo `lex.yy.c`.
- Por exemplo, uma subrotina que deve ser executada em várias regras.
- É importante destacar que o arquivo `lex.yy.c` não contém um `main`, pois ele é normalmente usado junto com o bison.
- Para usá-lo "sozinho", é necessário incluir um `main`.

```
%%
\+    { return MAIS; }
{pulo_linha} {conta_linhas++;}
program { simbolo  = simb_program;
          strncpy (token, yytext, TAM_TOKEN);
          IMPRIME("program ");
          return PROGRAM;
 }
```

Parte 3 - Subrotinas

- Trecho de código que será copiado para o arquivo `lex.yy.c`.
- Por exemplo, uma subrotina que deve ser executada em várias regras.
- É importante destacar que o arquivo `lex.yy.c` não contém um `main`, pois ele é normalmente usado junto com o bison.
- Para usá-lo "sozinho", é necessário incluir um `main`.

```
%%
\+    { return MAIS; }
{pulo_linha} {conta_linhas++;}
program { simbolo  = simb_program;
          strncpy (token, yytext, TAM_TOKEN);
          IMPRIME("program ");
          return PROGRAM;
 }
```

Parte 3 - Subrotinas

- Trecho de código que será copiado para o arquivo `lex.yy.c`.
- Por exemplo, uma subrotina que deve ser executada em várias regras.
- É importante destacar que o arquivo `lex.yy.c` não contém um `main`, pois ele é normalmente usado junto com o bison.
- Para usá-lo "sozinho", é necessário incluir um `main`.

```
int main () {
  yyin = fopen (argv[1], "r");
  yylex();
  fclose(yyin);
}
```

Compilador: flex

- Acesse o endereço `http://www.inf.ufpr.br/bmuller`.
- Em "Encargos Didáticos", acesse `CI211`.
- Baixe o arquivo `Projeto base`.
- Complemente o arquivo `compilador.l`, acrescentando todos os tokens válidos para a linguagem Pascal. Consulte o livro do Tomasz, apêndice 1. Palavras em negrito.

Bibliografia

- Praticamente todos os links retornados com a busca `"tutorial lex"` podem ser usados.
- Um livro completo é `"Lex & Yacc", John R. Levine, Tony Mason, Doug Brown - O'Reilly Media`