Ferramentas para Construção de Compiladores

Profa. Thatyana de Faria Piola Seraphim

thatyana@unifei.edu.br

Gerador de Analisador Léxico

A ferramenta `LEX` ou `FLEX`:

- Permite especificar um analisador léxico definindo expressões regulares para descrever padrões para os *tokens*.
- A notação de entrada para a ferramenta `LEX` é chamada de linguagem *Lex* e a ferramenta em si é o compilador *Lex*.
- O compilador *Lex*:
  - Transforma os padrões de entrada em um diagrama de transição.
  - Gera o código em um arquivo chamado `lex.yy.c`, que simula o diagrama de transição.

Gerador de Analisador Léxico

O uso do LEX

- Um arquivo de entrada chamado `lex.l` é escrito na linguagem *Lex* e descreve o analisador léxico a ser gerado.
- O compilador `Lex` transforma o `lex.l` em um programa C e o armazena em um arquivo que sempre se chama `lex.yy.c`.

`lex.yy.c`:

- É compilado pelo compilador C em um arquivo chamado `a.out`.
- Consiste em uma representação de um diagrama de transição construído a partir de expressões regulares do `lex.l`.
- Utiliza de uma rotina padrão que utiliza a tabela de símbolos para reconhecer os *tokens*.
- As ações associadas a cada expressão são trechos de códigos escritos em que são carregados diretamente no `lex.yy.c`.

`a.out`: é a saída do compilador C.

- É o próprio analisador léxico gerado.
- Pode receber como entrada um fluxo de caracteres e produzir como saída um fluxo de *tokens*.

Formato de um programa Lex

declarações
%%
regras de tradução
%%
funções auxiliares

- **Declarações**: inclui declaração de variáveis, identificadores que significam uma constante (nome de um *token*).
- **Regras de tradução** possuem o seguinte formato:

  `Padrão { Ação }`

  - `Padrão`: é uma expressão regular que pode usar as definições regulares da seção de declaração.
  - `Ação`: são pedaços de código escritos em C.

- **Funções auxiliares**: contém quaisquer funções adicionais usadas nas ações. Essas funções podem ser compiladas separadamente e carregadas com o analisador léxico.
- Quando o analisador léxico é chamado pelo analisador sintático:
  - Começa a ler da entrada um caracter de cada vez, até encontrar um caracter que case com um dos padrões P_i.
  - Depois executa a ação associada A_i, e a retorna ao analisador sintático.
  - Caso a ação A_i não seja retornada, o analisador léxico prossegue a leitura para encontrar lexemas adicionais, até que uma das ações resulte em retorno ao analisador sintático.

- O analisador léxico retorna um único valor (nome do *token*) ao analisador sintático.
- Para retornar o valor, é utilizada uma variável compartilhada inteira chamada `yylval` para passar informações adicionais sobre o lexema encontrado.
- `yylval` pode conter o valor do atributo, um código numérico, ou um apontador para a tabela de símbolos.

Exemplo de um programa LEX - Declarações

```
%{
   /* comentarios */
   #include<stdio.h>
   #include<stdlib.h>
%}
```

- Qualquer código em C inserido antes do primeiro %% deve ser inserido nesta seção.
- qualquer declaração feita entre %{ e %} é copiada diretamente no arquivo gerado na compilação `lex.yy.c`.

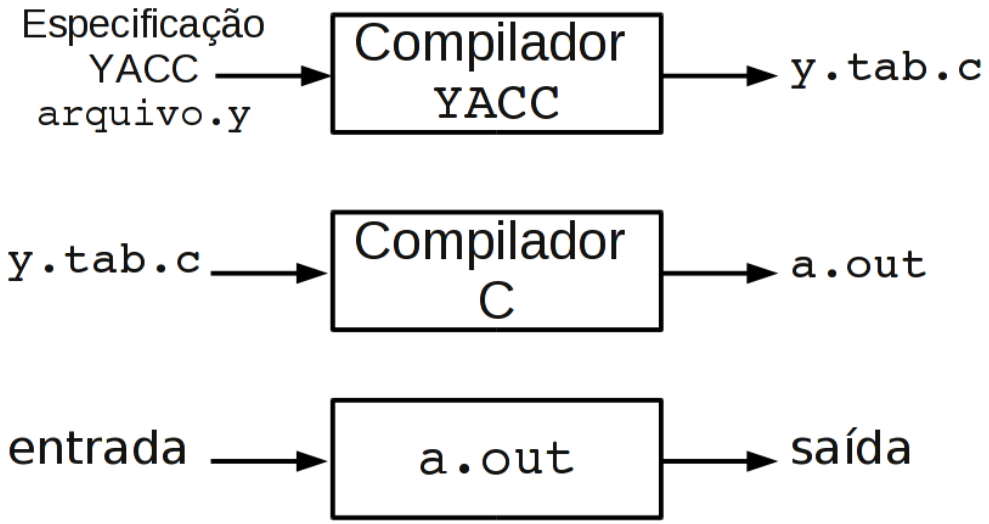
Exemplo de um programa LEX - Regras de Tradução

```
%%
"if"        { return (IF); }
"else"      { return (ELSE); }
"=="        { return (EQ); }
// yytext aponta para o primeiro caracter do token
"="|"+"|"-"|"*"|"/"|"%"  { return *yytext; }
// yylval retorna o valor do token armazenado na tabela de símbolos
[-+]?[0-9]+("."[0-9]*)?([eE]"-"?[0-9]*)? {
            yylval.pont→val = atof(yytext);
            return (NUM); }
[a-zA-Z][a-zA-z0-9]* {
            strncpy(yylval.pont→nome, yytext, 256);
            return (IDENT); }
```

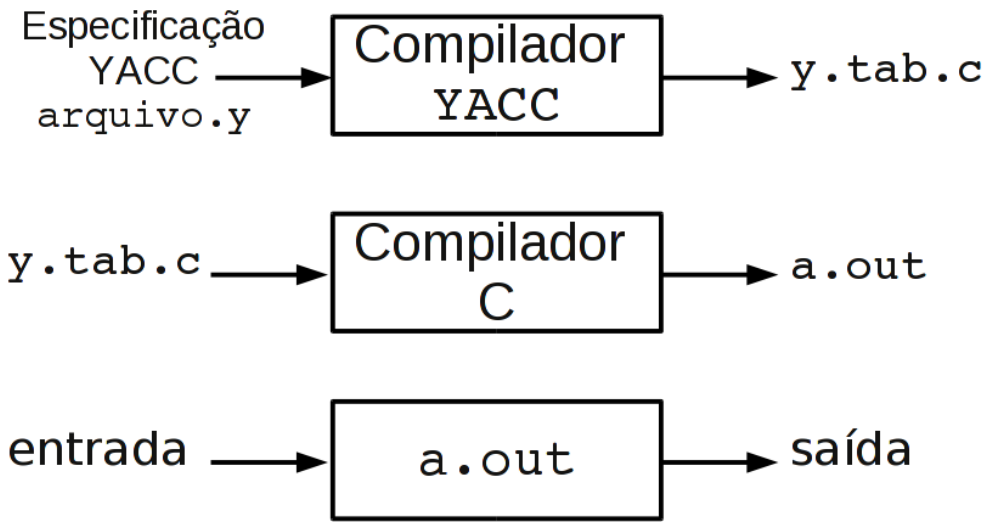
Exemplo de um programa LEX - Funções Auxiliares

```
%%
void main(){
        yylex(); // rotina que retorna o token
}
```

- Contém o código C para rotinas auxiliares.
- Pode conter um programa principal.
- Pode ser omitida e o segundo %% não precisa ser escrito.



- O comando do sistema UNIX `yacc arquivo.y` transforma o arquivo `arquivo.y` em um programa C chamado `y.tab.c`
- O programa `y.tab.c` é uma representação de um analisador LALR escrito em C, junto com outras rotinas C que o usuário pode ter preparado



- Compilando o programa `y.tab.c` obtém-se o programa objeto `a.out`
- `a.out` traduz o programa YACC
- se outros procedimentos forem necessários, eles podem ser compilados ou carregados com `y.tab.c`

- Um programa fonte *YACC* possui três partes

Partes de um programa YACC

declarações
%%
regras de tradução
%%
rotinas de suporte em C

- Para construir uma calculadora simples que lê e avalia uma expressão aritmética e imprime seu valor numérico

Gramática para a calculadora

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \textbf{digito}$

Declarações

- Existem duas seções na parte de declarações de um programa *YACC*, onde ambas são opcionais
- **Primeira seção**: coloca-se declarações C comuns que são delimitadas por %{ e %}; são colocadas declarações de quaisquer temporários usados pelas regras de tradução ou procedimentos da segunda e terceira seções
- Para o exemplo da calculadora, esta seção contém apenas o comando `#include <ctype.h>` que faz com que o pré-processador C inclua o arquivo de cabeçalho padrão

Seção de Declaração

```
%{
#include <stdio.h>
#include <type.h>
}%
```

- A declaração dos *tokens* usados são definidos pelo comando

  %token DIGITO

- Os *tokens* declarados nesta seção podem ser usados na segunda e na terceira parte da especificação

Regras de Tradução

- Após o primeiro par %%, são colocadas as regras de tradução
- Cada regra consiste em uma produção da gramática e a ação semântica associada

- O conjunto de produções (regras) são descritos da seguinte forma

Regras da Linguagem

<head> : <body>$_1$ { <ação semântica>$_1$ } |
<body>$_2$ { <ação semântica>$_2$ } |
...
<body>$_n$ { <ação semântica>$_n$ };

- Em uma produção (regra) *YACC*
  - cadeias de letras e dígitos sem aspas, não declaradas são consideradas como não-terminais
  - lados direitos alternativos podem ser separados por uma barra vertical

Regras da Linguagem

<head> : <body>$_1$ { <ação semântica>$_1$ } |
<body>$_2$ { <ação semântica>$_2$ } |
...
<body>$_n$ { <ação semântica>$_n$ };

- Em uma produção (regra) *YACC*
  - um ponto-e-vírgula vem após cada lado esquerdo com suas alternativas e suas ações semânticas
  - um único caractere entre apóstrofos ('c') é considerado o símbolo terminal **c**, assim como o código inteiro para o *token* representado por esse caractere
  - *head* é considerado o símbolo inicial
- Uma ação semântica do *YACC* é uma sequência de instruções C

- Em uma ação semântica
  - o símbolo $$ refere-se ao valor do atributo associado ao símbolo não-terminal *head*
  - o símbolo $\$_i$ refere-se ao valor do atributo associado ao i-ésimo símbolo da gramática (terminal ou não-terminal) de *body*
  - a ação semântica é efetuada sempre que reduzimos pela produção associada, de modo que a ação semântica calcula um valor para $$ em termos dos $\$_{is}$
- Por exemplo,na produção-E

$$E \rightarrow E + T \mid T$$

- As ações semânticas para a produção são escritas da seguinte forma

```
exp : exp '+' termo { $$ = $1 + $3;} |
      termo { $$ = $1;}
      ;
```

Rotinas de suporte em C

- A terceira parte de uma especificação *YACC* consiste em rotinas de suporte em C
- Um analisador léxico com o nome `yylex()` precisa ser fornecido
- Outros procedimentos, como rotinas de recuperação de erros, podem ser acrescentadas
- O analisador léxico `yylex()` produz *tokens* consistindo em um nome de *token* e seu valor de atributo associado
- Se um nome de *token* como `DIGIT` for retornado
  - o nome do *token* deve ser declarado na primeira seção da especificação *YACC*
  - o valor do atributo associado a um *token* é passado ao analisador sintático por meio de uma variável chamada `yylval` definida pelo *YACC*

- Pode-se atribuir precedências e associatividades aos símbolos terminais
- A declaração `%left '+' '-'`, faz com que + e - tenham a mesma precedência e sejam associativos à esquerda
- A declaração `%right '^'` define um operador como associativo à direita
- É possível forçar um operador a ser um operador binário não associativo, ou seja, duas ocorrências do operador não podem ser combinadas de forma alguma, escrevendo `%nonassoc '<'`

- Os *tokens* recebem precedências na ordem em que aparecem na parte das declarações, a mais baixa primeiro. Os *tokens* na mesma declaração, tem a mesma precedência
- Quando o símbolo terminal não possui a precedência apropriada a uma produção, é possível forçar a precedência da seguinte forma: `%prec <terminal>`

Análise Sintática Ascendente

Exemplo da calculadora – YACC

```
%{
#include<stdio.h>
#define YYSTYPE double
%}
%token DIGITO
%left '+' '-'
%left '*' '/'
%right UMINUS
lines : lines exp '\n' { printf("%f \n", $2); }
      | lines '\n'
      | /* vazio */
      ;
exp : exp '+' exp                { $$ = $1 + $3; }
      | exp '-' exp              { $$ = $1 - $3; }
      | exp '*' exp              { $$ = $1 * $3; }
      | '(' exp ')'              { $$ = $2; }
      | '-' exp %prec UMINUS     { $$ = - $2; }
      | DIGITO
      ;
```

Exemplo da calculadora – LEX

```
%{
#include<stdio.h>
#include<stdlib.h>
#include "calculadora.tab.h"
%}
%%
[-+]?[0-9]+("."[0-9]*)?([eE]"-"?[0-9]*)? {
                yylval.pont->val = atof(yytext);
                return DIGITO;
}
"+" | "-" | "*" | "/" { return *yytext; }
[ ] { /* pula espacos */ }
```