UNIOESTE – Foz
Ciência da Computação

Compiladores

Ferramentas Lex/Yacc

Profa. Camile Bordini

Lex e Yacc / Flex e Bison

- **Papel do Lex**: toma um conjunto de expressões regulares e produz uma rotina C que irá executar a análise léxica identificando os tokens.

- **Papel do Yacc**: toma uma descrição concisa de uma gramática e produz uma rotina C que irá executar a análise sintática ou parsing

Yacc / Bison

- **yacc** (yet another compiler-compiler) ou **bison**
- É uma ferramenta similar e complementar ao gerador de analisador léxico.
- Utiliza algoritmo de análise sintática ascendente (*bottom-up* ou *shift-reduz*)
- Utiliza pilha

Yacc / Bison

- **Lex** gera a função **yylex()** que retorna o identificador de um item léxico reconhecido.

- **Yacc** gera a função **yyparse()** que analisa os itens léxicos e decide se eles formam ou não uma sentença válida.

Yacc/Bison

Yacc / Bison

1ª parte: Declarações

%%

2ª parte: Regras de tradução

%%

3ª parte: Rotinas de suporte

Yacc / Bison

- Supondo a gramática abaixo:

calc → exp **`\n`** | calc exp **`\n`** | **λ**
exp → exp **+** termo | termo
termo → termo ***** fator | fator
fator → **(** exp **)** | **NUM**

- Quais *tokens* que o Lex deve reconhecer?

- Criar as ER's no arquivo **.l** para o reconhecimento dos *tokens*

1ª Seção: Declarações

- Composta por 2 partes, ambas opcionais:

1. Declarações de C/C++ delimitadas por **%{** e **%}**
   - Arquivos de cabeçalho, comentários
   - Declarações de constantes, variáveis e protótipos de funções

2. Declaração dos tokens da gramática e de regras auxiliares para solução de ambiguidades

1ª Seção: Declarações

- Exemplo

```
%{
  #include <stdio.h>
  #include "exemplo.tab.h"   //nome do arquivo .y
  int yylex(void);
  int yyparse(void);
  void yyerror(char *);
%}

%token NUM
%left MAIS MENOS
%left VEZES DIVIDIR

%%
```

2ª Seção: Regras de tradução

- Cada regra consiste em:
  - Uma produção da gramática
  - Uma ação semântica associada

Uma gramática definida assim:

```
lado_esquerdo -> alt1 | alt2 |...| altN
```

Transforma-se da seguinte especificação Yacc:

```
lado_esquerdo : alt1   {/*ação 1 opcional*/}
              | alt2   {/*ação 2 opcional*/}
              ...
              | altN   {/*ação N opcional*/}
              ;
```

2ª Seção: Regras de tradução

- Cada símbolo (terminal ou não) tem associado a ele um atributo (uma pseudo variável);
- O símbolo do lado esquerdo tem associado a ele a pseudo variável **`$$`**;
- Cada símbolo **i** da direita tem associado a ele a variável **`$i`**;
- **`$i`** contém o valor do token (retornado por **yylex()**) para os símbolos **i** que estiverem à direita da regra, e **`$$`** para o não terminal à esquerda;
- Ações podem ser executadas sobre estas variáveis

2ª Seção: Regras de tradução

- Gramática:

calc → exp **\n** | calc exp **\n** | **λ**
exp → exp **+** termo | termo
termo → termo ***** fator | fator
fator → **(** exp **)** | **NUM**

2ª Seção: Regras de tradução

- Em Yacc ficará:

```
%%
calc: exp '\n'              { printf("%d\n",$1); }
    | calc exp '\n'         { printf("%d\n",$2); }
    |                       //Para a cadeia vazia
    ;
exp: exp '+' termo          { $$ = $1 + $3; }
   | termo                  //ação implícita pelo yacc: $$=$1;
   ;
termo: termo '*' fator      { $$ = $1 * $3; }
     | fator
     ;
fator: '(' exp ')'          { $$ = $2; }
     |  NUM                 { $$ = $1;
                              printf("%d\n",$1);
                            }
     ;
%%
```

2ª Seção: Observações

- Nas ações semânticas foram colocadas:
  - As propagações dos valores para o pai na árvore
  - Até chegar na raiz onde é feito a computação desejada com o resultado (ex: imprimir na tela, gerar código, gerar uma árvore, etc.)

2ª Seção: Observações

- As pseudo variáveis **`$i`** são alimentadas através da variável externa **yylval** (criada pela rotina léxica)
  - Por padrão **yylval** é do tipo inteiro
  - Caso deseja-se que o atributo do *token* tenha um tipo diferente de inteiro devemos colocar na 1ª seção:

```
//Forma de comunicar ao lex que será
//tipo double para os atributos em yylval
%define api.value.type {double}
```

  - Para atributos com mais de um tipo, deve-se usar **uniões**

3ª Seção: Rotinas de suporte

- Exemplo:
  - Rotinas de erros (não é obrigatório a utilização da função padrão yyerror, mas neste caso, é necessário criar uma função própria)
  - A função main chama o analisador sintático yyparse

```
%%
void yyerror(char *s){  //sempre que houver um erro
                        //no yacc é chamado yyerror
    printf("Erro: %s\n", s);
}

int main(void){
    yyparse(); //início do processo
    return 0;
}
%%
```

Gramáticas ambíguas

- Para gramáticas ambíguas, haverá conflitos, mas o yacc informa o número de conflitos gerados
  - Uma descrição pode ser obtida rodando o yacc com a opção –v
  - Essa opção gera um arquivo y.output contendo:
    - Uma descrição dos conflitos
    - Uma tabela mostrando como os conflitos foram resolvidos

- Por padrão, os conflitos são resolvidos com as seguintes regras pelo yacc:
  - Conflito **reduz-reduz**: é resolvido escolhendo a produção listada primeiro
  - Conflito **shift-reduz**: é resolvido escolhendo fazer o shift

Gramáticas ambíguas

- Mas, idealmente, é preferível que o yacc não “decida” automaticamente
- Para isso, há mecanismos para resolver conflitos

- Para atribuir **associatividade** aos símbolos terminais:
  - `%left  //terminais são associados à esquerda`
  - `%right  //terminais são associados à direita`
  - `%nonassoc  //terminais não são associativos`

- Para determinar **precedências**:
  - Na ordem em que aparecem na declaração (mais baixa primeiro)
  - Terminais listados na mesma declaração possuem a mesma precedência

Lex/Flex

3ª Seção: Rotinas de suporte

```
%{
#include<stdlib.h>
#include "exemplo.tab.h"//contém a definição dos tokens
%}
%%
[0-9]+  {    //yylval por padrão é do tipo inteiro
            yylval = atoi(yytext);
            return NUM;
        }
[+*()\n]  { return *yytext; }

[ \t]    ; /* skip whitespace */
.        { yyerror("Caracter invalido"); }
%%
int yywrap(void) {
    return 1;
}
```

3ª Seção: Rotinas de suporte

- Os tokens criados com o yacc são compartilhados para o lex

Compilação

- São 4 passos para criar um parser:
  - Escrever uma especificação de uma gramática no formato do yacc. O arquivo terá a extensão **.y**
  - Escrever uma especificação de um analisador léxico que pode produzir *tokens*; extensão **.l**
  - Executar o yacc sobre a especificação **.y** e o lex sobre a especificação **.l**
  - Compilar e linkar os códigos fontes do parser, do analisador léxico e suas rotinas auxiliares.

Compilação

- A saída do yacc, **yy.tab.c**, contém a rotina **yyparse()** que chama a rotina **yylex()** para obter tokens.
- Como o Lex, o Bison não gera programas completos.
- A **yyparse()** deve ser chamada a partir do main.
- A rotina léxica lê a entrada, e a cada token encontrado, retorna o número do token ao parser.
  - A rotina léxica pode também passar o valor do token usando a variável externa **yylval** (entre outras).
- Um programa completo também requer uma rotina de erro chamada **yyerror**

Compilação

- Na linha de prompt (Windows), deve-se entrar com os seguintes comandos:

```
> yacc -d exemplo.y
> lex exemplo.l
> gcc exemplo.tab.c lex.yy.c
> a.exe
```

obs: o parâmetro d é usado para se gerar arquivo .h

- O arquivo `exemplo.tab.c` é gerado pelo parser e o `lex.yy.c` pelo scanner. Ambos devem ser compilados com o gcc para gerar o executável

- Este executável é o analisador/tradutor desejado.