



Judson Santos Santiago

# Gerador de Analizador Sintático

Compiladores

# Introdução

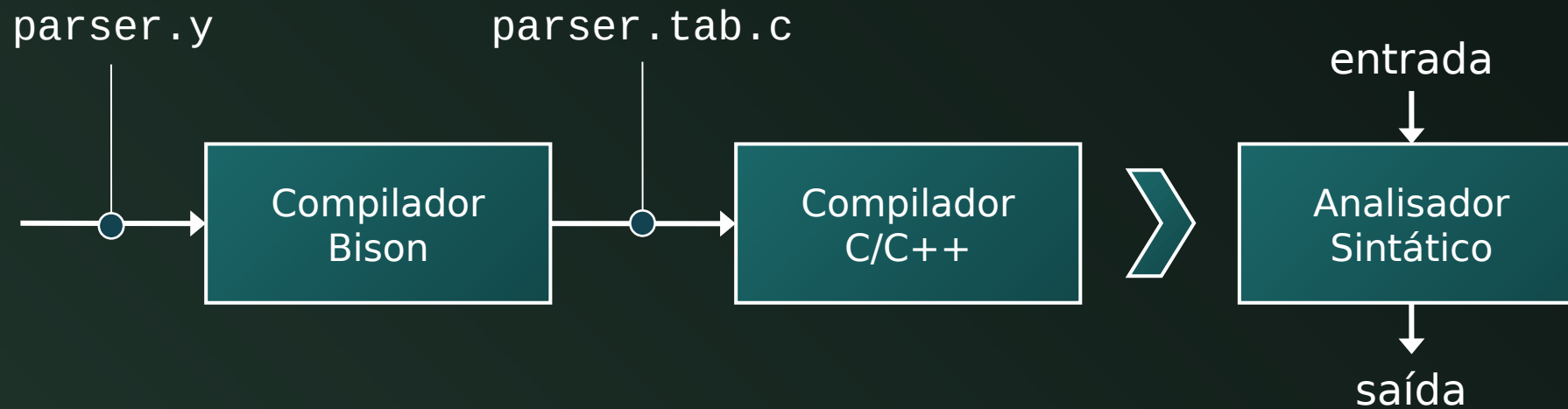
- Um **gerador de analisador sintático** simplifica e agiliza a construção de um compilador
  - Yacc
    - Foi **criado** por Stephen Johnson do Bell Labs **para o Unix nos anos 70**
    - Yacc significa "Yet another compiler-compiler"
    - Se destacou por combinar:
      - Uma sólida fundamentação teórica baseada no trabalho de **Donald E. Knuth**
      - Uma conveniente sintaxe de entrada

# Introdução

- Um **gerador de analisador sintático** simplifica e agiliza a construção de um compilador
  - Bison
    - Em 1985, um estudante de graduação chamado Bob Corbett criou uma **versão livre** e mais rápida do Yacc, chamada hoje de **Berkeley Yacc**
    - O GNU Project integrou e continuou a melhorar o projeto de Corbett, no que é hoje conhecido como GNU Bison
    - É uma **implementação mais recentes** do Yacc
    - Derivado do Berkeley Yacc

# Introdução

- Uso do Bison:
  - Um arquivo com uma especificação bison possui a extensão `.y`
  - O compilador Bison gera um arquivo com a extensão `.tab.c`
  - A saída do compilador C/C++ é o analisador sintático



# Estrutura do Programa

- Uma **especificação Bison** possui o seguinte **formato**:

Declarações

%%

regras de tradução

%%

rotinas de suporte

- Vamos exemplificar construindo uma calculadora:

```
expr  ::= expr + term
      | term
term   ::= term * factor
      | factor
factor ::= ( expr )
      | dígito
```

**dígito** é um único  
número de 0 a 9

# Estrutura do Programa

- Declarações

- São compostas por duas seções, ambas opcionais:
  - Declarações da linguagem C/C++ delimitadas por `%{` e `%}`
    - Inclusão de arquivos de cabeçalho, comentários
    - Declarações de constantes, variáveis e funções (protótipos)
  - Declarações dos tokens da gramática


```
%{  
#include <cctype>  
%}
```

```
%token DIGIT
```

```
%%
```

# Estrutura do Programa

- Regras de tradução
  - Cada regra consiste em:
    - Uma **produção** da gramática
    - Uma **ação semântica** associada

cabeça  corpo<sub>1</sub> | corpo<sub>2</sub> | ... |  
a corpo<sub>n</sub>

```
cabeça : corpo1 { ação semântica1 }  
      | corpo2 { ação semântica2 }  
      | ...  
      | corpo3 { ação semântica3 }  
      ;
```

# Estrutura do Programa

- Exemplo de regras de tradução:

```
%%  
calc : expr '\n'      { cout << $1 << '\n'; }  
      ;  
expr  : expr '+' term  { $$ = $1 + $3; }  
      | term  
      ;  
term  : term '*' fact   { $$ = $1 * $3; }  
      | fact  
      ;  
fact  : '(' expr ')'    { $$ = $2; }  
      | DIGIT  
      ;  
%%
```



# Estrutura do Programa

- Nas **especificações** de gramáticas:
  - Cadeias de letras e dígitos sem aspas são consideradas **não-terminais**
  - Um único caracteres entre aspas simples é considerado um **terminal**
- Em uma **ação semântica**:
  - O **símbolo** `$$` refere-se ao valor de atributo da cabeça
  - O **símbolo** `$i` refere-se ao valor do i-ésimo símbolo do corpo
- Podemos **omitir a ação** para produções com um único símbolo
  - A ação `{ $$ = $1; }` é a **ação padrão**

# Estrutura do Programa

- Exemplo de rotinas de suporte:

%%

```
int yylex() {           // implementação manual do analisador léxico
    char ch;
    ch = cin.get();
    if (isdigit(ch)) {
        yylval = ch - '0';
        return DIGIT;
    }
    return ch;
}

int main() {           // função principal chama o analisador sintático
    yyparse();
}
```

# Estrutura do Programa

- Nas **rotinas de suporte**:
  - Um **analisador léxico** com o nome `yyllex()` **precisa ser fornecido**
  - O **Flex** pode ser usada para produzir `yyllex()`
  - Rotinas de **recuperação de erros** são procedimentos comuns
  - A **função principal** chama o analisador sintático `yyparse()`
- O `yyllex()` do exemplo:
  - Lê um caractere por vez
  - Se for um dígito, retorna o **token DIGIT**, colocando o **atributo** em `yylval`
  - Caso contrário, retorna o **código do caractere** como token

# Gramáticas Ambíguas

- Vamos construir uma **calculadora melhorada**:
  - Avalia uma **sequência de expressões**
    - Uma expressão por linha
    - Permitindo linhas em branco
      - Uma alternativa vazia denota  $\epsilon$

```
calc : calc expr '\n'      { cout << $2 << endl; }
      | calc '\n'
      | /* vaziao */
      ;
```

# Gramáticas Ambíguas

- Ampliaremos a **classe de expressões**
  - Para incluir **números** e não apenas dígitos
  - Para incluir os **operadores aritméticos** +, -, \*, /, - unário
- O modo mais fácil é usando uma **gramática ambígua**

```
expr : expr '+' expr      { $$ = $1 + $3; }  
      | expr '-' expr      { $$ = $1 - $3; }  
      | expr '*' expr      { $$ = $1 * $3; }  
      | expr '/' expr      { $$ = $1 / $3; }  
      | '(' expr ')'        { $$ = $2; }  
      | '-' expr %prec UMINUS { $$ = -$2; }  
      | NUMBER  
      ;
```

# Gramáticas Ambíguas

- Como a gramática é ambígua haverá conflitos
  - O Bison informa o número de conflitos gerados
  - Uma descrição pode ser obtida rodando o Bison 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:
    - Um conflito `reduce/reduce` é resolvido escolhendo a produção listada primeiro
    - Um conflito `shift/reduce` é resolvido em favor da transferência (shift)

# Gramáticas Ambíguas

- O Bison oferece também um **mecanismo** para resolver conflitos
  - Podemos atribuir **associatividades** aos símbolos terminais
    - **left** faz com que os terminais sejam **associativos à esquerda**
    - **right** faz com que os terminais sejam **associativos à direita**
    - **nonassoc** faz com que os terminais **não sejam associativos**
  - Símbolos terminais recebem **precedência**:
    - Na **ordem em que aparecem** na declaração (mais baixa primeiro)
    - Terminais listados na mesma declaração possuem a **mesma precedência**

```
%left '+' '-'  
%left '*' '/'  
%nonassoc UMINUS
```

# Gramáticas Ambíguas

- O Bison resolve conflitos conectando uma precedência e uma associatividade a cada produção e terminal envolvidos
  - A precedência de uma produção é a mesma de seu terminal mais a direita
    - A produção  $E \rightarrow E + E$  possui a mesma precedência do  $+$
    - A produção  $E \rightarrow E * E$  possui a mesma precedência do  $*$
    - O operador  $*$  possui precedência maior que o operador  $+$

$E \rightarrow E + E$   
 $| E * E$

Pilha	Entrada
$\$E+E$	$+E... \$$

Reduzir  $E+E$  ou transferir  
 $+$  para a pilha?

Pilha	Entrada
$\$E+E$	$*E... \$$

Reduzir  $E+E$  ou transferir  
 $*$  para a pilha?



# Gramáticas Ambíguas

- O Bison reduz se:
  - A precedência da produção for maior que a do terminal
  - Ou as precedências forem iguais e a associatividade da produção for left
  - Caso contrário, a transferência (shift) é escolhida

E  E + E  
| E \* E

Pilha	Entrada
\$E+E	+E...\$

O Bison reduz E+E

Pilha	Entrada
\$E+E	*E...\$

O Bison transfere (shift) o terminal \* para a pilha

# Gramáticas Ambíguas

- Nas situações em que o terminal mais a direita não fornece a precedência apropriada, podemos anexar uma tag

```
%prec <terminal>
```

```
expr : '-' expr %prec UMINUS { $$ = -$2; }
```

- Supõem-se que o terminal foi definido na seção de declarações
- Ele pode ser um marcador de lugar, com o UMINUS

```
%left '+' '-'
```

```
%left '*' '/'
```

```
%nonassoc UMINUS
```

# Resumo

- O Bison é um **gerador de analisador sintático**
  - Derivado do Berkeley Yacc e compatível com o Yacc (Unix)
  - Gera um analisador sintático **em código C/C++**
  - Permite a criação de linguagens **de forma incremental**
  - Trata **gramáticas ambíguas**, com recursão à esquerda e não fatoradas
- Pode ser usado em conjunto com o Flex
  - Flex & Bison permitem uma implementação rápida do **front-end de um compilador**