

– INF01147 –
Compiladores

Análise Sintática 2/5
Descendente Recursivo com Retrocesso
Descendente Recursivo Preditivo

Prof. Lucas M. Schnorr
– Universidade Federal do Rio Grande do Sul –



Introdução a Análise Sintática

(revisão da aula anterior)

Análise Sintática – Tratamento de Erros

- ▶ Objetivos da recuperação de erros
 - ▶ Informar o erro de forma clara e precisa
 - ▶ Recuperar-se rapidamente (para detectar mais erros)
 - ▶ Custo baixo quando a entrada é correta
- ▶ Várias estratégias
 - ▶ **Modo Pânico**
 - Descarta tokens até um token de sincronismo
 - ▶ **Recuperação em nível de frase**
 - Realizar a correção local
 - ▶ Substituição de tokens
 - ▶ Exclusão
 - ▶ Inserção
 - ▶ **Produções de erro**
 - Prever erros com produções gramaticais

GLC – Estratégias de Análise

- ▶ Duas possibilidades
 - ▶ **Descendente** *top-down*
 - ▶ **Ascendente** *bottom-up*
- ▶ Exemplo para a entrada **ccbca** e a gramática
$$S \rightarrow AB$$
$$A \rightarrow c|\epsilon$$
$$B \rightarrow cbB|ca$$
 - ▶ Descendente parte de S e tenta chegar a **ccbca**
 - ▶ $S \Rightarrow AB \Rightarrow cB \Rightarrow ccbB \Rightarrow ccbca$
 - ▶ Ascendente parte de **ccbca** e tenta chegar a S
 - ▶ $ccbca \Rightarrow ccbB \Rightarrow cB \Rightarrow AB \Rightarrow S$

GLC – Outras Definições

- ▶ Gramática **sem ciclos** – inexistência de produções tipo $A \Rightarrow^+ A$ para algum $A \in NT$
- ▶ Gramática **ϵ -Livre** – inexistência de produções tipo $A \rightarrow \epsilon$, salvo $S \rightarrow \epsilon$ onde S é o símbolo inicial
- ▶ Gramática **Fatorada à Esquerda** – sem produções tipo $A \rightarrow \alpha\beta_1|\alpha\beta_2$, sendo α uma forma sentencial
- ▶ Gramática **Recursiva à Esquerda** – com a produção $A \Rightarrow^+ A\alpha$ para algum $A \in NT$
 - ▶ Recursão direta ou indireta
 - ▶ Impossibilita uma análise descendente (*top-down*)
 - ▶ Consumo do *token* é feito após a escolha da produção
 - ▶ Exemplo: $A \rightarrow Aa | b$ e a entrada **ba**

Análise Descendente x Recursão à Esquerda

- ▶ Recursão à esquerda **impossibilita** a análise descendente
- ▶ Considerando a gramática recursiva à esquerda

$S \rightarrow expr$

$expr \rightarrow expr + term \mid term$

$term \rightarrow a \mid b$

- ▶ E considerando a entrada **$a + b$**
- ▶ Na implementação da análise descendente
 - ▶ Três funções: $S()$, $expr()$ e $term()$
 - ▶ A execução chega na função $expr()$
 - ▶ Devido ao símbolo $+$, esta função escolhe a produção
 $expr \rightarrow expr + term$
 - ▶ Essa escolha implica em uma chamada recursiva a $expr()$
 - ▶ Nenhum símbolo da entrada é reconhecido
 - ▶ Recursão infinita

Análise Descendente x Recursão à Esquerda

- Movendo a recursão da esquerda para a direita, temos

$S \rightarrow \text{expr}$

$\text{expr} \rightarrow \text{term } R$

$R \rightarrow +\text{term } R \mid \epsilon$

$\text{term} \rightarrow a \mid b$

- Considerando a entrada $a + b$

- Considerando a gramática $A \rightarrow Aa \mid b$ e a entrada **ba**

Revisão

- ▶ O que é análise sintática? Para que serve?
- ▶ Podemos usar ER como formalismo reconhecedor na análise sintática?
- ▶ O que é uma gramática? Qual classe de gramáticas vamos usar?
- ▶ Quais as três características que uma gramática deve ter para que nossos algoritmos de reconhecimento funcionem?
 - ▶ ϵ -Livre (sem produções $A \rightarrow \epsilon$)
 - ▶ Fatorada à esquerda
 - ▶ Sem recursividade à esquerda
- ▶ Qualquer gramática livre de contexto pode apresentar as características acima?

Plano da Aula de Hoje

- ▶ Como reconhecer se a sentença pertence a gramática?
- ▶ Análise Sintática Descendente (*top-down*)
 - ▶ Recursiva com retrocesso
 - ▶ Recursiva Preditiva
 - ▶ Noção de *lookahead*
 - ▶ Introdução das primitivas *First* e *Follow*
- ▶ Yacc/Bison

Análise Sintática Descendente

- ▶ Como implementar um **reconhecedor** (*parser*) para uma GLC?
 - ▶ Constrói-se a árvore de derivação
 - ▶ A sentença da entrada é lida **da esquerda para a direita**
 - ▶ Derivação **mais à esquerda** de não-terminais
 - ▶ Tipos
 - ▶ Recursivo com retrocesso
 - ▶ Recursivo preditivo
 - ▶ Tabular preditivo
- Autômato de pilha guiado por uma tabela de análise

Recursiva com Retrocesso

Algoritmo Recursivo com Retrocesso

- Considerando a gramática

$$S \rightarrow A B$$

$$A \rightarrow c \mid \epsilon$$

$$B \rightarrow cbB \mid ca$$

- Ela gera $S \Rightarrow^* cbca$?

- Solução

S	cbca	escolhe $S \rightarrow AB$
AB	cbca	escolhe $A \rightarrow c$
cB	cbca	consome c
B	bca	sem saída, retrocede para a última escolha
AB	cbca	escolhe $A \rightarrow \epsilon$
B	cbca	escolhe $B \rightarrow cbB$
cbB	cbca	consome c
bB	bca	consome b
B	ca	escolhe $B \rightarrow cbB$
cbB	ca	consome c
bB	ca	sem saída, retrocede para a última escolha
B	ca	escolhe $B \rightarrow ca$
ca	ca	consome c
a	a	consome a
\emptyset	\emptyset	entrada reconhecida

Algoritmo Recursivo com Retrocesso

- ▶ É fácil de implementar, mas analisa por força bruta
 - ▶ Ineficiente e com custo computacional alto
- ▶ Qual o requisito fundamental?
 - ▶ Gramática **não seja recursiva à esquerda**
 $A \rightarrow Aa|b$
 - ▶ Considerando a entrada **ba** que pertence a essa gramática
- ▶ A gramática precisa ser fatorada a esquerda?
 - ▶ Ambiguidade na escolha da derivação influencia?

Recursiva Preditiva

Recursiva Preditivo (Introdução)

- ▶ Analisadores recursivos sem retrocesso
→ **Recursivos Preditivos**
- ▶ Um único símbolo terminal é suficiente para avançar
 - ▶ Escolhendo assim a próxima produção a ser aplicada
- ▶ Eles exigem
 - ▶ Que a gramática **não** possua recursão à esquerda
 - ▶ Que a gramática esteja **totalmente** fatorada à esquerda
 - ▶ Se houver não-terminais com mais de uma produção
Exemplo: $B \rightarrow bB \mid ca$
 - ▶ Os primeiros terminais deriváveis devem identificar a produção a ser escolhida de forma única e direta
- ▶ Com **a** na entrada e o não-terminal **A** a ser derivado
 - ▶ Determinar qual das produções
 $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
inicia por **a**

Recursivo Preditivo (Exemplo)

- ▶ Estamos tratando o não-terminal Comando
- ▶ Considerando as produções

Comando \rightarrow **if** Expr **then** Comando |
while Expr **do** Comando |
repeat Lista **until** Expr |
id := Expr

- ▶ É possível escolher qual produção com um único token?
 - ▶ Sim

Recursivo Preditivo (Exemplo)

- ▶ Estamos tratando o não-terminal Comando
- ▶ Considerando as produções

Comando	→	Condicional Iterativo Atribuição
Condicional	→	if Expr then Comando
Iterativo	→	while Expr do Comando repeat Lista until Expr
Atribuição	→	id := Expr

- ▶ Ainda é possível fazer a escolha com um único token?
- ▶ **Sim!**
 - ▶ Mas precisamos determinar o conjunto de terminais deriváveis a partir de Condicional, Iterativo e Atribuição
 - ▶ Isto é, o conjunto **First** de cada uma dessas produções

Yacc / Bison

Yacc – Introdução

- ▶ yacc -Yet Another Compiler Compiler
- ▶ Produz um analisador ascendente para uma gramática
- ▶ Usado para produzir compiladores para várias linguagens
 - ▶ Pascal, C, C++, ...
- ▶ Outros usos
 - ▶ bc (calculadora)
 - ▶ eqn & pic (formatadores para troff)
 - ▶ Verificar a sintaxe SQL
 - ▶ Lex
- ▶ **bison** – versão livre da GNU

Yacc – Especificação de Entrada

- ▶ Contém três seções
 - ▶ Definições (em C, incluído no início da saída)
 - ▶ Regras (Especificação da Gramática)
 - ▶ Código (em C, incluído no fim da saída)

- ▶ Sintaxe

Definições

%%

Regras

%%

Código Suplementar

Yacc – Seção de Regras (seção principal)

- ▶ Contém a gramática
- ▶ Exemplo

```
expr : expr '+' term | term;  
term : term '*' factor | factor;  
factor : '(' expr ')' | ID | NUM;
```

Yacc – Seção de Definições (seção auxiliar)

- ▶ Contém a definição de tokens, símbolo inicial
- ▶ Exemplo

```
%{  
    #include <stdio.h>  
    #include <stdlib.h>  
%}  
%token ID NUM /* notar a declaração dos tokens */  
%start expr
```

Yacc – Interação com o analisador léxico

- ▶ Flex produz uma função `yylex()`
- ▶ Bison produz uma função `yyparse()`

- ▶ Flex e Bison: concebidos para interagirem
 - ▶ `yyparse()` chama `yylex()` para obter um token

- ▶ Duas opções
 - ▶ Ou implementamos manualmente `yylex()`
 - ▶ Ou **utilizamos Flex diretamente**

Yacc – Sequência básica operacional

- ▶ Supondo os arquivos
 - ▶ **scanner.l** com as especificações de tokens em lex
 - ▶ **parser.y** com a gramática em yacc
- ▶ Ordem de passos possível para construir o analisador

```
bison -d parser.y
```

```
flex scanner.l
```

```
gcc -c lex.yy.c y.tab.c
```

```
gcc -o parser lex.yy.o y.tab.o -lfl
```

- ▶ **scanner.l** deve incluir na seção de definições

```
#include "y.tab.h"
```

Yacc – Miscelânea

- ▶ As regras da gramática
 - ▶ Podem ser recursivas tanto a esquerda quanto a direita
 - ▶ Não podem ser ambíguas
- ▶ Usa um parser ascendente LALR(1)
 - ▶ Solicita um token
 - ▶ Empilha
 - ▶ Redução?
 - ▶ Sim \leadsto reduz usando a regra correspondente
 - ▶ Não \leadsto lê outro token na entrada
- ▶ Não pode olhar mais que um token de *lookahead*
- ▶ `bison -v gramatica.y` gera a tabela de estados

Exemplo (Lex) – arquivo scanner.l

```
%{  
#include <stdio.h>  
#include "y.tab.h"  
%}  
id [_a-zA-Z][_a-zA-Z0-9]*  
wspc [ \t\n]+  
semi [;]  
comma [,]  
%%  
int { return INT; }  
char { return CHAR; }  
float { return FLOAT; }  
{comma} { return COMMA; }  
{semi} { return SEMI; }  
{id} { return ID; }  
{wspc} {;
```

Exemplo (Yacc) – arquivo parser.y

```
%{  
#include <stdio.h>  
#include <stdlib.h>  
%}  
%start line  
%token CHAR, COMMA, FLOAT, ID, INT, SEMI  
%%  
decl : type ID list { printf("Success!\n"); } ;  
list : COMMA ID list | SEMI;  
type : INT | CHAR | FLOAT;  
%%
```

- Notem uma ação para a regra decl

Yacc – Ações e Atributos

- ▶ Cada regra pode ter **ações** (semânticas)
- ▶ Exemplo

```
E: E '+' E      { $$ = $1 + $3; }  
  | INT_LIT      { $$ = INT_VAL; };
```

- ▶ \$n é o atributo do n-ésimo símbolo na regra
- ▶ O default é que os atributos sejam do tipo inteiro
- ▶ Pode-se mudar o tipo através da diretiva

```
%token<...> /* com o tipo do token */  
%type<...>  /* tipo do não-terminal, com %union */
```

Yacc – Ações e Atributos (Exemplo)

```
%union {  
    char* nome;  
    int inteiro;  
    node* no;  
}  
%token<nome> IDF /* IDF terá atributo de tipo char* */  
%type<no> E      /* E terá atributo de tipo node* */  
%%  
E: E + E { $$ = create_node($1, $3, "plus"); }  
  | IDF { $$ = create_leaf($1); };
```

Conclusão

- ▶ Leituras Recomendadas
 - ▶ Livro do Dragão
 - ▶ Seções 2.4 e 4.4
 - ▶ Série Didática
 - ▶ Seções do Capítulo 3.2 até 3.2.1 inclusive
- ▶ Próxima Aula
 - ▶ Conjuntos First/Follow
 - ▶ Análise Sintática Preditiva Tabular