Análise Sintática - Final

http://www.dcc.ufrj.br/~fabiom/comp

LR(k) vs LL(k)

Encontrando o próximo passo em uma derivação

LR(k) ⇒ Cada redução na análise detectável com

- → o contexto esquerdo completo,
- → a frase a reduzir e
- → os k símbolos terminais à sua direita

generalizações de LR(1) e LL(1) para lookaheads maiores

LL(k) ⇒ Parser deve escolher a expansão com

- → o contexto esquerdo completo (pilha LL(k))
- → os próximos k terminais

Logo, LR(k) examina mais contexto

A questão é, existem linguagens que são LR(k) mas não LL(k)?

LR(1) vs LL(1)

A gramática LR(1) a seguir não tem correspondente LL(1)

0	*Goal*	→	*S*
1	*S*	→	*A*
2		|	*B*
3	*A*	→	(*A*)
4		|	a
5	*B*	→	(*B* >
6		|	b

- Precisa de lookahead arbitrário para escolher entre A e B
- Um parser LR(1) pode carregar o contexto (os '(' s) até encontrar um a ou b e saber como reduzir
- Um parser LL(1) não pode decidir se deve expandir Goal por A ou por B, e não adianta fatorar a gramática
  - → Na verdade, essa gramática não é LL(k) para nenhum k
  - → Precisa de um analisador LL(*)

LR(k) vs LL(k)

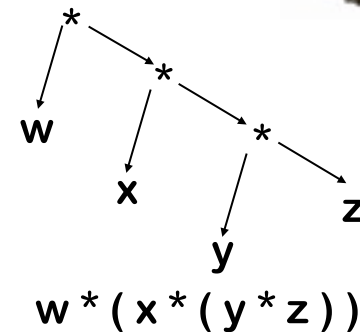
Outra gramática não-LL(k)

0	B	$\rightarrow$	R
1		$\mid$	$(\ B\ )$
2	R	$\rightarrow$	$E = E$
3	E	$\rightarrow$	$\underline{a}$
4		$\mid$	$\underline{b}$
5		$\mid$	$(\ E + E\ )$

Essa gramática é LR(0)!

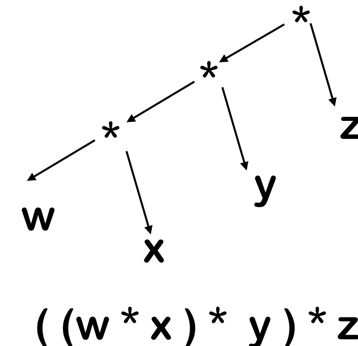
Recursão à Esquerda vs Recursão à Direita

- Recursão à direita
  - Necessária para terminação em parsers top-down
  - Usa mais espaço na pilha em parsers bottom-up
  - Associatividade à direita



w * (x * (y * z))

- Recursão à esquerda
  - Funciona em parsers bottom-up
  - Limita espaço necessário na pilha
  - Associatividade à esquerda



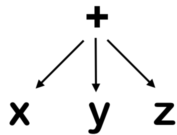
((w * x) * y) * z

- Regra geral
  - Recursão à esquerda para parsers bottom-up
  - Recursão à direita para parsers top-down, convertendo em laço onde possível

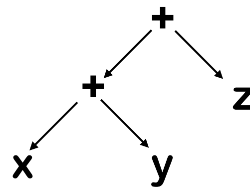
Associatividade

Que diferença faz em operações associativas?

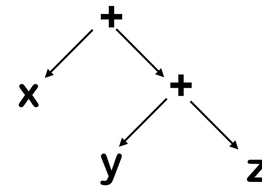
- Pode mudar respostas em ponto flutuante
- Oportunidades diferentes de otimização
- Considere x+y+z



Operador ideal



Associatividade esquerda



Associatividade direita

E se y+z aperece em outro lugar? Ou x+y? Ou x+z?

O compilador pode querer mudar a "forma" das expressões

- E se x = 2 & z = 17? Nem à esquerda nem à direita expõe 19.
- Melhor forma é função do contexto.

Detecção e Recuperação de Erros

Detecção de Erros

- Parser recursivo
  - Parser cai na última cláusula else ou default do switch
  - Projetista pode programar praticamente qualquer ação
- LL(1) de tabela
  - No estado s_i com token x, entrada é um erro
  - Reporta o erro, entradas na linha de s_i têm os possíveis tokens
- LR(1) de tabela
  - No estado s_i com token x, entrada é um erro
  - Reporta o erro, shifts na linha do estado têm possíveis tokens
  - Mensagens de erro podem ser pré-computadas para itens LR(1), aí é só consultar uma tabela

Detecção e Recuperação de Erros

Recuperação de Erros

- LL(1) de tabela
  - Trata como token faltante, p. ex. ')', ⇒ expande pelo símbolo desejado
  - Trata como token extra, p. ex., 'x - + y', ⇒ desempilha e segue adiante
- LR(1) de tabela
  - Trata como token faltante, p. ex. ')', ⇒ shift do token
  - Trata como token extra, p. ex., 'x - + y', ⇒ não faz shift

Detecção e Recuperação de Erros

Recuperação por token de sincronização

Avança na entrada até achar alguma "âncora", p. ex., ';'

- Resincroniza estado, pilha e entrada para ponto depois do token
  - → LL(1): desempilha até achar linha com entrada para ';'
  - → LR(1): desempilha até achar estado com redução em ';'
- Não corrige a entrada, mas deixa análise prosseguir

```
NT ← pop()
repeat until Tab[NT,';'] ≠ error
    NT ← pop()
token ← NextToken()
repeat until token = ';'
    token ← NextToken()
```

Resincronizando parser LL(1)

```
repeat until token = ';'
    shift token
    shift s_e
    token ← NextToken()
reduce por produção de erro
    // desempilha todo esse estado
```

Resincronizando parser LR(1)

Hierarquia das Linguagens Livres de Contexto

Linguagens Livres de Contexto

↓

Linguagens determinísticas (LR(k)) ← LR(k) ≡ LR(1)

↙ ↘

linguagens LL(k) → linguagens LL(1)

Linguagens de precedência simples → Linguagens de precedência de operadores

A hierarquia de inclusão para linguagens livres de contexto

Hierarquia das Gramáticas Livres de Contexto

Gramáticas Livres de Contexto

- Analisável Floyd-Evans
- CFGs não-ambíguas
  - LR(k)
    - LR(1)
      - LALR(1)
        - SLR(1)
          - LR(0)
          - LL(1)
    - LL(k)
      - LL(1)
- Precedência de Operadores

Existem gramáticas que não são ambíguas mas não são LR(k), mas toda linguagem para a qual existe uma gramática não ambígua tem uma gramática LR(1)

A hierarquia de inclusão para gramáticas livres de contexto

Resumo

	Vantagens	Desvantagens
Top-down Recursivo, LL(1)	Rápido Boa localidade Simples Bom trat. de erros	Escrito à mão Alta manutenção Assoc. direita
Bottom-up LR(1)	Rápido Linguagens determinísticas Automatizável Assoc. esquerda	Má localidade Trat. de erros difícil

Gerador de Parsers JACC

- Gerador de parsers LALR(1) para Java
- Sintaxe baseada na do YACC
- Tratamento de erros diferente do YACC, usa exemplos de erro ao invés de produções **error** e resincronização
  - Pode gerar analisadores com mensagens de erro muito boas, mas é bem mais trabalhoso
- Rápido, codifica a máquina de estado LALR(1) em código ao invés de usar uma tabela

Usando JACC

- Linha de comando
  - `jacc X.jacc`
  - Gera arquivos Java pro parser e pra interface de tokens
- Opções
  - -v: escreve saída da máquina de estados em arquivo X.output
  - -h: escreve máquina em formato HTML no arquivo XMachine.html
  - -fv, -fh: mostra conjuntos FIRST e FOLLOW para cada não-terminal em conjunto com as opções anteriores
  - -a, -s, -0: usa estratégia de parsing LALR(1), SLR(1), ou LR(0)
  - -e X.errs: lê exemplos de erros no arquivo X.errs

Usando JACC

- Arquivo de entrada

```
diretivas
%%
regras
%%
código extra
```

- Diretivas controlam a geração do parser
- Regras especificam a gramática e as ações durante a análise
- Código extra é inserido dentro da classe do parser

Diretivas JACC

- %class FooParser
  - Nome da classe (e nome do arquivo java gerado)
- %interface FooTokens
  - Nome da interface com códigos dos tokens (e do arquivo gerado)
- %next nextToken()
  - Código que carrega próximo token e retorna o tipo numérico dele
- %get tokenType
  - Código que pega tipo numérico do token corrente
- %semantic Node: tokenVal
  - Tipo do valor semântico dos símbolos da gramática, e código (depois do :) para pegar o valor semântico do token corrente
- %token FOO BAR BAZ
  - Tokens da linguagem

Regras JACC

- As regras JACC começam com o nome do não-terminal, seguido de : e das produções dele, separadas por |, e terminam com ;

```
expr : expr '+' term
     | expr '-' term
     | term
     ;

term : term '*' fact
     | term '/' fact
     | fact
     ;

fact : '(' expr ')'
     | NUM
     ;
```

Ações de Redução

- As ações que o parser deve executar quando reduzir uma produção vêm entre {} depois da produção
- O valor de cada elemento fica em pseudo-variáveis $1, $2, ...
- O valor da produção deve ser atribuído a $$

```
expr : expr '+' term    { $$ = $1 + $3; }
     | expr '-' term    { $$ = $1 - $3; }
     | term             // default { $$ = $1; }
     ;

term : term '*' fact    { $$ = $1 * $3; }
     | term '/' fact    { $$ = $1 / $3; }
     | fact
     ;

fact : '(' expr ')'
     | NUM
     ;
```

Operadores e Precedência

- As gramáticas JACC podem ser ambíguas na parte de expressões, usando diretivas de precedência/associatividade
  - %left, %right
- Ordem em que aperecem no arquivo dão precedência (menor para maior)

```
%left ‘+’ ‘-’
%left ‘*’ ‘/’
%right ‘^’

%%

expr : expr ‘+‘ expr  { $$ = $1 + $3; }
     | expr ‘-‘ expr  { $$ = $1 - $3; }
     | expr ‘*‘ expr  { $$ = $1 * $3; }
     | expr ‘/‘ expr  { $$ = $1 / $3; }
     | expr ‘^‘ expr  { $$ = Math.pow($1,$3); }
     | NUM
     ;
```

EBNF para JACC

- Repetição { ... } em JACC:
  - { ... } => a : a ... | ;
  - <1> { <2> } => a : a <2> | <1> ;
- Opcional [...] em JACC:
  - [...] => a : ... | ;
  - <1> [<2>] <3> => <1> <3> | <1> <2> <3>

Código Extra

- Tratamento de erros

```
void yyerror(String msg) {
    ...
}
```

- Dispare uma exceção para interromper a análise
- Outras coisas para incluir nessa seção: construtores, campos para guardar instância do analisador léxico e o token corrente