Capítulo 8 - Recuperação de Erros e Outros tipos de Analisadores Sintáticos

Edna Ayako Hoshino
eah@dct.ufms.br

1 de outubro de 2004

Este texto refere-se a notas de aulas da disciplina de Projeto e Implementação de Linguagens. As Seções 4.4 e 4.5, 4.6 e 4.7 do livro do Aho, Sethi e Ullman discutem em detalhes os assuntos mencionados superficialmente neste texto.

1 Recuperação de erros e aspectos de implementação

Nas implementações dos analisadores sintáticos discutidas até então, o analisador parava a análise ao encontrar o primeiro erro no programa fonte. É interessante que o compilador seja capaz de identificar o maior números de erros no programa em uma única execução. Dizemos que o compilador deve ser capaz de se recuperar do erro. Para tanto, o analisador sintático deve identificar a causa do erro e prosseguir a análise como se o erro tivesse sido corrigido. Mas, nem sempre é possível identificar, a partir do erro encontrado, qual era a intenção do programador. Por exemplo, considere o seguinte trecho de programa fonte:

```
int a,b c=10;
```

O analisador iria detectar um erro sintático ao encontrar um ID (c) após o outro ID (b) na declaração. Um possível erro é a falta de vírgula separando os dois ID. Então, o analisador prosseguiria a análise a partir do ID como se fosse parte da declaração de variáveis. Por outro lado, o erro poderia ser conseqüência da falta de ponto-e-vírgula e, portanto o trecho $c = 10$; faria parte do corpo do programa e não das declarações. Com este exemplo, podemos notar que dependendo do tratamento de erro realizado, podemos estar introduzindo outros erros erroneamente no programa fonte. Por exemplo, se o programador esqueceu de colocar um ponto-e-vírgula após o ID b, mas o compilador interpretar que há falta de uma vírgula, o compilador estaria introduzindo um erro (o compilador detectaria que o identificador c já havia sido declarado e identificará um erro de duplicidade de declaração de variáveis).

Dentre os erros que podem ocorrer dentro de um programa, temos os erros léxicos (por exemplo, erro na grafia de algum identificador), erros sintáticos (por exemplo, parênteses não balanceados), erros semânticos (por exemplo, tipo de argumento inválido na chamada de uma função) e os erros lógicos (como uma chamada recursiva infinita). A detecção dos erros semânticos e lógicos são mais difíceis e serão tratados em outra aula. Vamos discutir como detectar e se recuperar de erros sintáticos. Em geral, os erros léxicos que não foram identificados pelo analisador léxico são detectados pelo analisador sintático. Por exemplo, a grafia errônea "fi" da palavra-chave "if" não será detectada pelo analisador léxico, que irá reconhecê-lo como um ID. Desta forma, o analisador sintático perceberia o erro, apenas ao verificar o próximo token depois do ID, o qual deveria ser um operador de atribuição ou um parênteses para a chamada da função "fi".

Existem diferentes estratégias para a recuperação de erros, como por exemplo, a recuperação a nível de frase, recuperação na modalidade do desespero e a recuperação através de produções de erro. Na recuperação a nível de frase, o analisador tenta corrigir a entrada remanescente (a parte do programa fonte que segue o local em que ocorreu o erro) através da inserção e/ou remoção e/ou troca de caracteres até obter uma entrada sem erros, por exemplo, inserindo uma vírgula em int a,b c=10;. A idéia da recuperação através de produções de erro consiste em introduzir novas produções na gramática da linguagem de modo a gerar um trecho de programa com erro. Essa estratégia é útil para identificar erros comuns no programa fonte, como a falta de um ponto-e-vírgula ou ainda, um ponto-e-vírgula que não deveria ocorrer antes de um ELSE em estruturas condicionais na linguagem PASCAL. Nesta estratégia,

as produções de erro são implementadas da mesma forma que as demais produções da gramática. Na modalidade do desespero, a idéia é ignorar parte da entrada (a parte do programa fonte que segue o local em que ocorreu um erro) até que o analisador consiga prosseguir a análise de maneira segura (ou seja, sem introduzir novos erros). Nós utilizaremos a modalidade do desespero em nosso trabalho.

2 Modalidade do desespero

Na modalidade do desespero, quando o analisador encontra um erro, ele começa a descartar os tokens de entrada até que um token especial, chamado *token de sincronização* seja encontrado. Em geral, para cada tipo de erro encontrado no programa, um conjunto de tokens de sincronização diferente precisa ser definido para determinada linguagem fonte.

Vamos considerar a análise sintática preditiva recursiva. Considere que um erro sintático ocorreu na derivação de uma variável A, ou seja, nenhuma das produções de A pode ser usada na derivação (o lookahead não faz parte do início (FIRST) de nenhuma das produções de A). Uma maneira de retomar a análise sintática, é deixar de realizar a derivação de A, assumir que a derivação de A ocorreu com sucesso e prosseguir a análise retornando à função que chamou a função A(). Para prosseguir corretamente com a análise é necessário descartar os tokens da entrada que fazem parte da derivação de A. Isso pode ser feito, considerando os tokens em FOLLOW(A) como tokens de sincronização. Essa estratégia pode não ser muito boa, pois pode "pular" grande parte do programa fonte.

Por exemplo, considere que ponto-e-virgula é um token em FOLLOW(comando) e que ocorreu um erro dentro de um comando do programa fonte. Então, o analisador iria pular a entrada até encontrar o ponto-e-virgula. Mas, a omissão de ponto-e-vírgula ao final do comando errado fará com que o próximo comando também seja ignorado pelo analisador.

Uma alternativa, é retomar a análise quando aparecer um token que possa ser o início de uma das produções de A. Em outras palavras, podemos adotar os tokens em FIRST(A) como tokens de sincronização. No exemplo anterior, o primeiro token que ocorrer no início do próximo comando será usado como token de sincronização e, então o analisador iria pular apenas o comando contendo erros. Note que esta estratégia pode desencadear outros erros sintáticos.

Uma outra boa opção é postergar o erro sintático até que uma mensagem mais adequada possa ser reportada pelo analisador. Por exemplo, se nenhuma das produções de uma variável A é iniciada pelo lookahead, podemos derivar A usando a produção $A \rightarrow \varepsilon$, se esta produção existir.

Quando um erro sintático ocorre dentro da função CONSOME(), uma opção é mostrar uma mensagem de erro adequada e continuar a análise como se o token esperado tivesse sido encontrado (ou seja, qualquer token é usado como token de sincronização).

Considere o exemplo da aula passada:

S → a A e S | B d | d

A → a A | ε

B → b B | c

Com a recuperação de erros, temos a seguinte implementação para a função B():

```
Função B ()
    se lookahead = b entao
      consome (b)
      B()
    senao
      se lookahead = c entao
        consome (c)
      senao
        escreva "erro sintatico"
        repita
          lookahead → Lex()
        ate lookahead = b OU lookahead = c OU lookahead = d OU lookahead = FIM
        se lookahead = b OU lookahead = c entao
          B()
        fim se{caso contrário, retorna à função chamadora}
```

fim se
fim se

Fim

3 Outros tipos de analisadores

Vamos introduzir a idéia dos demais tipos de analisadores sintáticos.

3.1 Analisadores sintáticos Bottom-Up

A idéia dos analisadores sintáticos bottom-up é construir uma árvore gramatical para uma cadeia de entrada começando pelas folhas (do fundo) e trabalhando em direção à raiz (ao topo). Em outras palavras, os analisadores sintáticos bottom-up procuram reduzir a cadeia de entrada à variável inicial da gramática. Essa redução consiste em um processo inverso à derivação. A cada passo de redução, obtemos uma cadeia de símbolos gramaticais (variáveis e terminais). Seja α a cadeia de símbolos gramaticais obtida até o momento após vários passos de redução. O próximo passo de redução é aplicado sobre uma subcadeia particular de α, a qual coincide com o lado direito de alguma produção de uma variável A da gramática. A redução consiste em substituir essa subcadeia particular pela variável A em α, obtendo assim outra cadeia de símbolos gramaticais. Os analisadores sintáticos aplicam sucessivas reduções até obter um único símbolo gramatical, a variável inicial da gramática, ou até que uma redução não possa ser realizada (ocorra um erro).

Considere o seguinte exemplo:

S → a A B e

A → A b c | b

B → d

Os passos de redução para a palavra abbcde são:

```
a b b c d e => a A b c d e => a A d e => a A B e => S
```

Uma vez que a redução obteve a variável inicial, a palavra pertence à linguagem gerada pela gramática.

Note que os passos da redução em ordem inversa consiste em uma derivação mais à direita de abbcde.

As implementações dos analisadores sintáticos bottom-up giram em torno das seguintes idéias. Mantém-se uma pilha, na qual empilhamos os tokens da entrada até que uma seqüência de símbolos ao topo da pilha coincida com o lado direito de uma das produções. Quando isso ocorre, desempilhamos essa seqüência de símbolos e empilhamos a variável do lado esquerdo da produção que gerou a redução. Desta forma, os símbolos na pilha mais os tokens remanescentes na entrada formam uma cadeia de símbolos gramaticais em algum passo de redução. Assim, os analisadores consistem em realizar duas ações elementares: empilhar ou reduzir. Quando o único símbolo que resta na pilha é a variável inicial e toda a cadeia de entrada foi analisada temos obtido sucesso na análise.

Para que os analisadores sintáticos bottom-up possam ser implementados precisamos manter uma tabela que indique que ações devem ser realizadas com base na informação do símbolo que está ao topo da pilha e do lookahead. Basicamente, os diferentes tipos de analisadores sintáticos bottom-up se diferem na forma com que a tabela é construída.

São exemplos de analisadores sintáticos bottom-up: analisadores de precedência de operadores, analisadores sintáticos SLR, LALR, LR canônico. Sendo que cada um deles se aplica a classes de linguagens diferentes.

	id	+	*	FIM
id		>	>	>
+	<	>	<	>
*	<	>	>	>
FIM	<	<	<	

Tabela 1: Exemplo de tabela de precedência de operadores.

estado	id	+	*	(	)	FIM	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Tabela 2: Exemplo de tabela sintática SLR

4 Exercícios

1. Altere o analisador léxico construído no Exercício 2 do Capítulo 6 para incluir a especificação dos tokens MAIS, MENOS, VEZES, DIVISAO, ABREPARENT e FECHAPARENT.

2. Escreva um analisador sintático preditivo recursivo para aceitar uma seqüência de expressões aritméticas terminadas com OP_ATRIBUICAO. O seu programa deve usar o analisador léxico adaptado no Exercício 1.