NÁLISE SINTÁTICA INCREMENTAL PARA LINGUAGENS . 1)

OBERTO DA SILVA BIGONHA*, ROSILENE TEREZINHA MARTINS**

UMARIO

escreve-se uma estratégia para construção de um analisador sintático ncremental para linguagens LR(1). São propostos uma forma intermediária ara armazenamento de programas-fonte e modificações no reconhecedor R(1) para permitir a análise sintática incremental. A ênfase é a reaná- ise do ponto de vista sintático. Entretanto, esta estratégia pode ser ambém estendida às fases de análise semântica e geração de código.

BSTRACT

his paper describes an strategy for implementing incremental parsers for R(1) languages. In order to achieve incremental capabilities, a new ntermediate representation for source programs and a modified version of he LR(1) parser are proposed. The emphasis is on the reparsing problem. lowever, the proposed method can be extended and applied to other ompiler phases.

* PhD in Computer Science (UCLA,1981); Professor Adjunto da UFMG; Linguagens de Programação e Compiladores; DCC - ICEx - UFMG.

Bacharel em Ciência da Computação (UFMG,1984); Mestranda em Ciência da Computação; Linguagens de Programação e Compiladores; DCC - ICEx - UFMG.

MG - ICEx - DCC; CP 702; Belo Horizonte - MG; Tel.:(031) 443-4088.

1. INTRODUÇÃO

Atualmente, grande importância tem sido dada a projetos de sistemas de programação visando proporcionar ao usuário um ambiente de programação eficiente e confortável durante as fases de criação, depuração, manipulação e documentação de programas [1 a 17].

Ambientes de programação tradicionais envolvem, entre outras ferramentas, a utilização de um editor para criar e modificar programas, e um compilador para traduzir o código fonte em código objeto. Nestes casos, sempre que o código fonte sofre qualquer alteração, a versão de código objeto gerada anteriormente é substituída por um novo código modificado, gerado a partir da completa recompilação do fonte. Este processo deve se repetir após cada alteração no programa.

Compilação Incremental, entretanto, visa prover a facilidade de submeter à recompilação, unicamente os trechos de código alterados.

A chave para a compilação incremental é a idéia de se armazenar uma representação intermediária do código fonte, capaz de manter um relacionamento entre este e o código objeto, de forma que se possa a cada alteração no programa fonte efetuar as alterações necessárias no código objeto existente.

Para possibilitar compilação incremental, as soluções existentes, dentre elas as propostas por Teitelbaum [6], Crowe [3] [7], Medina-Mora [8], Budinsky [4], Atkinson [11], Ghezzi [1] e Kahrs [17], se baseiam na manutenção dinâmica da árvore sintática correspondente ao programa em questão. Isto equivale a dizer que a árvore sintática deve ser atualizada sempre que o código fonte for alterado. Convém ressaltar, entretanto, que tal representação do programa fonte apresenta uma série de desvantagens, como veremos posteriormente.

Neste trabalho, apresentamos uma outra forma de representação intermediária,que pode ser mantida atualizada sem os inconvenientes da árvore sintática, por ser mais econômica em termos de espaço de armazenamento e mais eficiente em termos de processamento (manutenção e utilização da estrutura).

2. A ABORDAGEM DE GHEZZI & MANDRIOLI

O método de compilação incremental proposto neste trabalho baseia-se nos resultados de Ghezzi & Mandrioli [1].

Ghezzi & Mandrioli estendem as técnicas de análise LR de maneira a dar suporte à construção de um analisador sintático incremental e utilizam o conceito de gramáticas RL.

Segundo Ghezzi e Mandrioli, uma gramática G é RL(k) se e somente se sua reversa Gr, isto é, a gramática cujos lados direitos das produções correspondem aos reversos das de G, for LR(k).

Seja LR(k) ^ RL(k) o reconhecedor simultaneamente LR(k) e RL(k) das sentenças da gramática G. Durante a análise LR(k) (RL(k)), a sentença é lida da esquerda para a direita (dá direita para a esquerda) e, somente os próximos k símbolos a serem lidos ("lookahead") afetam a decisão do analisador, qualquer que seja o seu estado corrente. Portanto, a porção de uma sentença que antecede (sucede) os k símbolos anteriores (posteriores) a uma modificação não é afetada durante a reanálise LR(k) (RL(k)).

Aos k símbolos que antecedem a alteração, chamaremos de I. Analogamente, os k símbolos que sucedem essa modificação chamaremos de F. Se considerarmos que a sentença em questão pertence à classe de linguagens LR(k) ^ RL(k), I e F desempenham o papel de delimitadores da porção a ser reanalisada, ou seja, garantem a correção da porção à direita e à esquerda da modificação, respectivamente. Logo, se uma gramática for simultaneamente LR(k) e RL(k) para algum k, é possível definir um reconhecedor sintático incremental que economiza passos de análise tanto do lado esquerdo quanto do lado direito da modificação realizada no fonte.

Em termos práticos, Ghezzi & Mandrioli propõem um método de se determinar a porção da árvore de derivação que é afetada por uma modificação no programa fonte. Assim, a reanálise do programa fonte consistiria somente em substituir a sub-árvore que corresponde ao trecho modificado, preservando-se o restante da árvore.

A abordagem de Ghezzi & Mandrioli é, sem dúvida alguma, uma contribuição de grande relevância. Contudo, baseia-se na manutenção dinâmica da árvore sintática associada ao programa, o que apresenta as seguintes desvantagens:

a) O método pressupõe um editor dirigido por sintaxe, o que se torna

indesejável, tendo em vista os seguintes aspectos:

- a maioria destes editores, tais como Cornell [6], SDE [3] e [9], caracteriza-se por um método de entrada e modificação programas, restritivo e altamente estruturado;
- o usuário manipula o código fonte em termos de unid sintáticas, o que não é natural, principalmente no que respeito aos movimentos de cursor efetuados pelo usuário s completamente inesperados tendo em vista o que foi por requisitado. Desde que programadores usam a representação tual para compreender seus programas, um editor deveria tr lhar também com texto, ou dar ao usuário esta sensação.

b) O custo de armazenamento da árvore é alto porque:

- devem ser mantidos os ponteiros que caracterizam a estru como árvore;
- para possibilitar a manutenção dinâmica da estrutura necessário que seja mantido um relacionamento entre a posiçã edição, como o usuário a vê, e a posição do nodo correspond na árvore sintática;
- para possibilitar as correspondentes modificações no có objeto, informações adicionais também devem ser mantidas árvore sintática, tais como a posição e o comprim associado a cada nodo nos arquivos fonte e objeto.

c) O custo de manutenção da árvore é caro porque:

- após cada modificação do código fonte deve-se gerar um fragme de árvore sintática, o qual deverá ser conectado à árvore cor pondente ao texto antigo, substituindo a sub-árvore que sofre alteração;
- ao conectar o novo fragmento de árvore, é necessário atuali informações contidas em todos os nodos ancestrais da nova s árvore (comprimento relativo ao nodo no texto fonte e, possiv mente no código objeto) e em todos os nodos à direita da r sub-árvore (posição relativa no texto fonte e no cód objeto), o que envolve um "overhead" considerável;

d) O caminhamento em árvore é mais complicado do que em uma estrut linear, o que encarece tanto a manutenção da estrutura quanto a r

nálise semântica e a geração de código em cada recompilação, visto que a árvore sintática resultante deve ser percorrida para consistir os aspectos semânticos da linguagem e gerar o código referente à nova modificação.

3. A ABORDAGEM PROPOSTA

3.1. REPRESENTAÇÃO DA FORMA INTERMEDIÁRIA

A forma intermediária proposta é uma lista de "tokens" de dois tipos: "tokens" normais e "tokens" de redução.

"Tokens" normais representam as unidades léxicas constituintes do programa fonte e contêm as seguintes informações:
- identificação do "token" compreendendo seu tipo e seu valor;
- estado do analisador LR(1) imediatamente antes de ler o "token".

"Tokens" de redução indicam pontos de ocorrência de reduções e contêm as seguintes informações:
- número da produção associada, que corresponde a um dos índices de uma tabela de produções. Cada entrada desta tabela contém informações relevantes acerca de uma produção, tais como comprimento do lado direito, não-terminal do lado esquerdo etc;
- ponteiro de reconstrução sintática, que é um apontador para o primeiro elemento da seqüência de "tokens" a ser reduzida utilizando-se a produção associada ao "token" de redução, isto é, um apontador para o início do "handle" [5] correspondente à redução. O próprio "token" de redução indica o final do "handle".

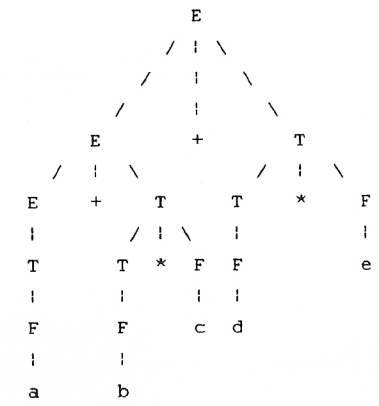
Antes da primeira compilação, o programa fonte deve ser convertido pelo editor (orientado por "token") para a forma intermediária, a qual inicialmente contém apenas "tokens" normais com informação sobre os seus tipos e valores apenas. Após a primeira compilação, os "tokens" de redução seriam incluídos e as informações contidas nos "tokens" normais seriam complementadas.

EXEMPLO:

Considere a gramática G com o seguinte conjunto de produçoes:

(1) F -> id (3) E -> T (5) T -> F
(2) E -> E + T (4) T -> T * F

Para a sentença a + b * c + d * e temos a seguinte árvore de derivação



A representação intermediária para esta sentença é a seguinte:

(id,a)	[r1,0]	[r5,1]	[r3,2]	(+,)	(id,b)	[r1,5]	[r5,6]	(*,)
0	1	2	3	4	5	6	7	8

(id,c)	[r1,9]	[r4,7]	[r2,3]	(+,)	(id,d)	[r1,14]	[r5,15]	(*,)
9	10	11	12	13	14	15	16	17

(id,e)	[r1,18]	[r4,16]	[r2,12]
18	19	20	21

Os termos entre parênteses são os "tokens" normais e contêm o tipo token e, quando necessário, o seu valor. Os termos entre colchetes são "tokens" de redução e contêm o número da produção a ser utilizada e ponteiro de reconstrução sintática que endereça um token da lista. endereços dos "tokens" são os números indicados abaixo de cada um.

3.2. SUPOSIÇÃO DE GRAMÁTICAS LR(1) ^ RL(1)

Em nosso trabalho, a análise sintática incremental se aplica às gramát cas LR(1) ^ RL(1) dentro da mesma filosofia da abordagem de Ghezzi Mandrioli [1]. O teorema apresentado a seguir é uma particularização pa k = 1 do teorema de Ghezzi & Mandrioli para gramáticas LR(k) ^ RL(k sendo de fundamental importância para garantir a validade do método.

TEOREMA

Seja G = (N,T,P,S) uma gramática LR(1) ^ RL(1). Seja G' a gramát geradora do conjunto de formas sentenciais de G.

ntão, para todo x, y, z em (N U T)*; I, F em T; A, B em N tal que:

a) |I| = |F| = 1

b) $AI \underset{Grm}{\overset{*}{\Rightarrow}} xI$

c) $FB \underset{Glm}{\overset{*}{\Rightarrow}} Fy$

IzFy pertence a L(G) se e somente se AIzFB pertence a L(G') ■

al teorema garante que, relativamente a um "string" xIzFy, todas as eduções efetuadas pelo analisador LR(1) (RL(1)) durante a aná- ise de x (y) não são afetadas pela modificação de z, ou seja, os símbo- os I e F delimitam o escopo das modificações efetuadas.

abemos que as gramáticas LR(k) são também LR(1). Analogamente e por efinição, gramáticas RL(k) são também RL(1). Tendo em vista tais consi- erações, em termos práticos, não se restringe a classe de gramáticas à ual o método proposto se aplica, relativamente à abordagem de Ghezzi & andrioli.

.3. DETERMINAÇAO DE I E F

e F são os delimitadores da porção a ser refeita, sendo o critério para sua determinação descrito a seguir:

seja v o fragmento da lista de "tokens" correspondente ao trecho do programa que sofreu alteração;

sejam a e b , respectivamente, o primeiro e o último "token" normal de v;

seja c o primeiro "token" de redução à direita de b cujo ponteiro de reconstrução sintática aponte para um "token" qualquer à esquerda de a ou para o próprio a, ao qual chamaremos de d;

então, F corresponde ao próximo "token" normal à direita de c;

se d for um "token" normal, então I corresponde ao próximo "token" normal à esquerda de d; senão, I corresponde ao próximo "token" normal à esquerda de e, sendo e o "token" normal obtido seguindo os ponteiros de reconstrução sintática a partir de d.

mente o trecho de programa compreendido entre I e F, não incluindo

estes, deverá ser reanalisado do ponto de vista sintático após as modificações introduzidas em v. A garantia de que I e F realmente delimitam o escopo das modificações em v decorre dos seguintes fatos:

1) A gramática base é LR(1), portanto toda redução efetuada à esquerda de I não depende de quaisquer símbolos à direita de I.

2) A gramática base é também RL(1), portanto toda redução à direita de F, segundo o reconhecedor RL(1), não depende de nenhum símbolo à esquerda de F.

3) I e F, por construção e pela forma como o apontador de reconstrução sintática foi definido, delimitam o menor fragmento de programa na forma intermediária que contém o trecho modificado.

Convém ressaltar que a e b acima citados são os delimitadores do escopo da modificação efetuada pelo usuário a nível de código fonte. I e F, no entanto, definem o escopo da modificação em termos sintáticos, ou seja, delimitam o trecho de código cuja sintaxe pode ser afetada pela alteração efetuada pelo usuário, e por este motivo deve ser reanalisado.

3.4. A REANÁLISE SINTÁTICA

O analisador incremental efetua a reanálise considerando como entrada os "tokens" normais da estrutura intermediária compreendidos entre os delimitadores I e F, os quais ele recebe como parâmetro. Para a primeira análise realizada em um programa fonte, I e F denotam o primeiro e o último "token" respectivamente. Para reanálises subsequentes, os valores de I e F são determinados conforme o algoritmo da Seção 3.3.

Durante uma reanálise, os "tokens" de redução existentes no trecho de lista delimitado por I e F são irrelevantes pois refletem a análise sintática efetuada antes da última modificação no fonte, podendo ser removidos antes ou durante a reanálise.

Toda modificação no programa fonte é assistida por um editor orientado por "token", cuja filosofia geral de funcionamento é descrita na seção 3.5. Do ponto de vista sintático, o editor é responsável por preparar a estrutura intermediária após cada modificação a fim de que o analisador possa efetuar a reanálise, ou seja, substituir o trecho da lista correspondente à porção modificada no fonte pelos novos "tokens" normais, lidos do teclado. Este novo fragmento de lista que é inserido na estrutura pelo editor corresponde ao v do algoritmo apresentado na Seção 3.3.

dos I e F, a análise sintática LR(1) incremental é feita da seguinte neira:

inicialmente, o estado referente a I é empilhado;

I é lido novamente;

o analisador prossegue de acordo com as açoẽs usuais de um reconhecedor LR(1) tomando como entrada os "tokens" normais da lista até a posição definida por F. Caso não tenham sido removidos, os "tokens" de redução existentes no trecho de lista delimitado por I e F são ignorados e eliminados pelo analisador;

para cada redução identificada pelo reanalisador sintático, insere-se um "token" de redução na posição corrente do fragmento de programa sendo reanalisado.

5. EDIÇÃO ORIENTADA POR "TOKEN"

ssa proposta é trabalhar com um editor capaz de tratar o programa como a sequência de "tokens". Tal escolha, além de beneficiar a análise sintica, apresenta uma série de vantagens.

visão de programas como sequências de "tokens" parece ser mais natural e a visão estrutural, visto que os programas não são exibidos em forma árvore. Além disso, a visão por "tokens" permite que os movimentos do rsor ocorram nas direções esperadas pelo usuário ao editar seu programa neste aspecto, é bem mais agradável ao usuário do que um editor rigido por sintaxe que dá ao usuário a sensação de estar editando os dos de uma árvore e não um texto.

unidades léxicas constituintes do programa seriam editadas na ordem rmal, e um mecanismo simples do tipo "cut-paste" seria suficiente para modificações. Neste aspecto, tal abordagem se aproxima bastante da são textual, oferecendo as mesmas vantagens e comodidades ao usuário.

visão por "tokens" é ainda mais natural do que a textual pois as idades léxicas constituintes de um programa são vistas pelo programador mo entidades indivisíveis, e não como sequências de caracteres.

6. A REANALISE SEMÂNTICA

mo a execução das rotinas semânticas é dirigida por sintaxe, o estado qual o analisador LR(1) será ativado é obtido do "token" de reinício do por I. A análise semântica é feita percorrendo a lista de "tokens"

da esquerda para a direita. Se um "token" normal é encontrado, são empilhados os atributos do "token"; se um "token" de redução é alcançado, são executadas as ações semânticas correspondentes, são desempilhados os atributos relativos a tantos "tokens" no topo da pilha quantos forem os símbolos do lado direito da respectiva produção e empilhados os valores relativos às ações semânticas.

Dada a complexidade da compilação parcial no aspecto de análise semântica e geração de código, o sucesso da reanálise semântica depende do ponto de reinício estar ou não correto.

Pode-se incorporar ao analisador incremental a execução de ações semânticas que são ativadas a cada redução. Neste caso, dever-se-á levar em conta os efeitos da modificação do fonte do ponto de vista semântico durante o cálculo de I e F, isto é, é possível que se tenha que considerar um novo par (I',F') que abranja uma porção de código mais extensa que a delimitada pelo método puramente sintático apresentado na Seção 3.3.

Neste trabalho, não será apresentado um critério para determinação dos delimitadores análogos a I e F para a reanálise semântica. Este assunto será objeto de estudos posteriores.

4. CONCLUSÕES

Nosso trabalho propõe um algoritmo de análise incremental associado a uma nova representação intermediária mais econômica e que proporciona maior eficiência de processamento que as apresentadas até o momento. Neste sentido, pode-se construir uma ferramenta de auxílio à implementação de compiladores incrementais para linguagens LR(1), em particular LALR(1), baseado em um editor dirigido por "token", um parser incremental LALR(1) e em técnicas de suporte à análise semântica e geração de código parciais.

A contribuição de nosso trabalho se concentra basicamente na forma de representação proposta para a estrutura intermediária, os algoritmos de determinação de I e F e o correspondente analisador sintático, o que conjuntamente possibilita a recompilação mínima de código após uma alteração no fonte.

Do ponto de vista sintático, o problema pode ser satisfatorimante solucionado. Contudo, dada a complexidade da compilação parcial no aspecto de análise semântica e geração de código, o sucesso da reanálise semântica depende do ponto de reinício estar ou não correto.

REFERÊNCIAS BIBLIOGRÁFICAS

C Ghezzi and D Mandrioli, Incremental Parsing, ACM TOPLAS, I(1), 58-70 (1979).

G Brun, A Businger and R Schoenberger, The Token-Oriented Approach to Program Editing, SIGPLAN Notices Vol. 20(2) 17-20 (February 1985).

M Crowe, C Nicol, M Hughes and D Mackay, On Converting a Compiler into an Incremental Compiler, SIGPLAN Notices Vol.19(7) 14-22 (July 1984).

F J Budinsky, R C Holt and S G Zaky, SRE - A Syntax Recognizing Editor, Software - Practice and Experience, Vol. 15(5), 489-497 (May 1985).

A V Aho and J D Ullman, Principles of Compiler Design, Addison-Wesley, 1977.

T Teitelbaum and T Reps, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, Communications of the ACM Vol. 24(9) 563-574 (September 1981).

M K Crowe, An Incremental Compiler, SIGPLAN Notices Vol. 17(10) 13 (1982).

R Medina-Mora and P H Feiler, An Incremental Programming Environment, IEEE Trans. Software Eng., Vol. SE-7, No.5 , 472-482, (1981).

L Allison, Syntax Directed Program Editing, Software - Practice and Experience, Vol. 13(5), 453-465 (1983).

T Teitelbaum, The why and wherefore of the Cornell Program Synthesizer, SIGPLAN Notices, Vol. 16(6), 8-16 (1981).

L V Atkinson, J J McGregor and S D North, Context sensitive editing as an approach to incremental programming, The Computer Journal, Vol. 24(3), 222-229, (1981).

J M Morris and M D Schwartz, The design of a language-directed editor for block-structured languages, SIGPLAN Notices Vol. 16(6), 28-33 (1981).

C R Jesshope, M J Crawley and G L Lovegrove, An Intelligent Pascal Editor for a Graphical Oriented Workstation, Software - Practice and Experience, Vol. 15(11), 1103-1119 (November 1985).

14. S R Wood, Z-The 95% program editor, SIGPLAN Notices, (June 1986 pp. 1-7.

15. O Stromfors and L Jones jo, The Implementation and Experience of Structure-Oriented Text Editor,SIGPLAN Notices, (June 1986), pp. 22 27.

16. C W Frases, Syntax-Directed Editing of General Data Structures SIGPLAN Notices, (June 1986), pp. 17-21.

17. M Kahrs, Implementation of an interactive programming system, SIGPLA Notices, (June 1986), pp. 76-82.