RECUPERAÇÃO DE ERROS EM ANALISADORES SINTÁTICOS DESCENDENTES

HELOISA VIEIRA DA ROCHA CORRÊA SILVA

Orientador: Prof. Dr. Tomasz Kowaltowski

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Dezembro - 1981

AGRADECIMENTOS

Ao prof. Tomasz Kowàltowski pelo seguro trabalho de orientação.

Aos colegas e amigos que direto ou indiretamente contribuiram para a realização deste trabalho.

A paciência e incentivo de meu marido, cuja contribuição, às vezes silenciosa, foi inestimável.

Ao Raul, pelo excelente trabalho de datilografia.

à meus pais

SUMMARY

We present in this thesis a survey of compiling error recovery methods applicable to top-down parsers for LL languages.

First we describe the compilation process and possible sources of errors. We also include a short description of recovery methods for lexical errors, and of methods applicable to bottom-up parsers.

Next we describe the implementation of top-down parsers, and of several recovery methods applicable to this kind of parsing.

Finally we show the experimental results of implementing some of these methods for the Pascal language. Our conclusion is that none of these methods is satisfactory in all circumstances, but some of them seem to perform in general better than others.

RESUMO

O trabalho aqui apresentado é um estudo de métodos de recuperação de erros no processo de compilação, aplicáveis à analisadores descendentes para linguagens do tipo LL.

Inicialmente faz-se uma exposição sobre o processo de compilação e possíveis fontes de erro. Está incluida também uma indicação sucinta dos métodos de recuperação para erros léxicos e dos métodos aplicáveis à analisadores ascendentes.

Segue um resumo sobre a implementação de análise descendente e a descrição de vários métodos de recuperação aplicáveis à este tipo de análise.

Finalmente, apresentam-se os resultados experimentais da implementação de alguns destes métodos para a linguagem Pascal. Chega-se à conclusão de que nenhum desses métodos apresenta comportamento satisfatório em todas as situações, mas alguns parecem ter desempenho geral melhor do que outros.

SUMÁRIO

1 - INTRODUÇÃO

1.1 - Compilação 1
1.2 - Erros e Recuperação 4
1.2.1 - Erros 4
1.2.2 - Fontes de Erro 5
1.2.3 - Erros Sintáticos 7
1.2.4 - Erros de Contexto 9
1.2.5 - Erros dinâmicos 9
1.2.6 - Recuperação em cada fase 10
1.2.6.1 - Fase léxica (erros de ortografia) 10
1.2.6.2 - Fase sintática 11
1.2.6.3 - Fase de contexto 12
1.3 - Objetivo e Roteiro do Trabalho 13
1.4 - Notação Utilizada 13
2 - ALGUNS MÉTODOS DE RECUPERAÇÃO 17
2.1 - Correção de erros de ortografia 17
2.2 - Recuperação de erros sintáticos para analisadores ascendentes 23
2.2.1 - Aspectos gerais da análise ascendente 23
2.2.2 - Métodos de recuperação de erros sintáticos 24
3 - ANÁLISE SINTÁTICA DESCENDENTE E MÉTODOS DE RECUPERAÇÃO DE ERROS 35
3.1 - Métodos de implementação de análise descendente 35
3.1.1 - Implementação com retrocesso 35
3.1.2 - Implementação com pilha explícita s/retrocesso 44
3.1.3 - Implementação recursiva 46
3.1.4 - Implementação com pilha explícita orientada por Tabela 49
3.1.4.1 - Funções PRI e SUC 53
3.1.4.2 - Construção da Tabela de análise 55
3.2 - Métodos de recuperação de erros 56
3.2.1 - Método de Turner 62

3.2.2 - Método de Wirth 70
3.2.3 - Método de Hartmann e Pemberton 79
3.2.4 - Método de Irons 90
3.2.5 - Método de Fisher, Milton e Quiring 94
3.2.6 - Método de Ghezzi 100
3.2.7 - Método de Setzer 108
3.2.8 - Método de Pai e Kieburtz 110
3.2.8.1 - Algumas definições e Teoremas 113
3.2.8.2 - Recuperação local 116
3.2.8.3 - Recuperação de contexto global 118
3.2.8.4 - Caracterização dos símbolos fiduciais 122
4 - CONCLUSÕES 130
BIBLIOGRAFIA 137

CAPÍTULO 1

INTRODUÇÃO

Neste capítulo, dá-se uma visão geral do processo de compilação e dos tipos de erro que ocorrem neste processo, procurando definir o que se entende por recuperação de erro. Depois disso, apresentam-se os objetivos e o roteiro do trabalho em questão, e a notação utilizada.

1.1 - Compilação

Um programa fonte em uma linguagem de programação nada mais é que uma cadeia de caracteres. Um compilador converte esta cadeia de caracteres em uma cadeia de bits denominada código objeto. Neste processo vários subprocessos podem ser identificados e pode-se definir cada um desses subprocessos como uma fase do compilador, sendo conveniente pensar-se em análise léxica, análise sintática, análise de contexto e geração do código objeto como as principais fases do compilador. Entretanto, em alguns compiladores mais sofisticados, estas fases são subdivididas em diversas subfases, e outras fases, como otimização do código objeto gerado, podem estar presentes.

A análise léxica é a primeira fase.

A entrada para o compilador, e portanto para o analisador léxico, é uma cadeia de caracteres de um determinado alfabeto.

Em um programa, certas combinações de caracteres geralmente são tratados como uma entidade simples. Alguns exemplos disto podem ser citados

- certas linguagens possuem palavras chaves tais como begin, end, goto, do, etc., as quais são tratadas como entidades simples.

- cadeias representando constantes numéricas.

- identificadores usados como nomes de variáveis, funções, procedimentos, rótulos, etc.

O analisador léxico agrupa caracteres em entidades sintáticas simples, chamadas átomos. O que vai ser considerado um átomo geralmente está explicitado na especificação da linguagem de programação, podendo esta decisão ficar a cargo do implementador do compilador.

Portanto, o analisador léxico é um tradutor que tem como entrada uma cadeia de caracteres representando o programa fonte, e como saída uma cadeia de átomos. Esta saída será a entrada para o analisador sintático.

À medida que os átomos são descobertos na análise léxica, informações sobre eles são coletadas e armazenadas em uma ou mais tabelas. Por exemplo, a seguinte declaração em Pascal:

var a: array [1..10, 1..20] of integer.,

Ao encontrar-se esta declaração é preciso armazenar a informação de que a é um identificador, o qual é o nome de uma matriz bidimensional de tamanho 10 por 20, e que seus elementos são números inteiros. Pode-se considerar, de uma maneira geral, que existe uma tabela na qual são armazenadas informações sobre identificadores. Esta tabela é normalmente denominada Tabela de símbolos e não é necessáriamente manipulada pelo analisador léxico, ficando esta decisão a cargo do implementador.

A análise sintática é o processo no qual a cadeia de átomos, gerada

na análise léxica, é examinada para determinar se ela obedece às convenções estruturais explicitadas na definição sintática da linguagem. É essencial no processo de geração de código que se conheça qual é a estrutura sintática de uma determinada cadeia. Por exemplo, a estrutura sintática da expressão a + b - c precisa refletir o fato de que b e c devem ser multiplicados primeiro e que o resultado deve ser somado a a. Nenhuma outra ordem de operação produz o cálculo desejado. A saída do analisador sintático, implícita ou explícita, é uma árvore a qual representa a estrutura sintática inerente ao programa fonte.

Existem dois tipos básicos de analisadores sintáticos: descendentes e ascendentes. Como indicam seus nomes, os analisadores ascendentes constroem as árvores sintáticas da base (folhas) para a raíz, enquanto os descendentes começam com a raíz e trabalham em direção às folhas. Em ambos os casos a entrada para o analisador é percorrida da esquerda para direita, um átomo por vez. O compilador precisa também verificar se certas convenções de contexto da linguagem fonte foram obedecidas.

Alguns exemplos destas convenções podem ser:

(1) cada rótulo referenciado precisa aparecer como rótulo de algum comando apropriado no programa fonte.

(2) Nenhum identificador pode ser declarado mais de uma vez, em um mesmo escopo.

(3) Todos os identificadores precisam ser definidos antes de serem utilizados.

(4) Os argumentos de chamada de um procedimento precisam

ser compatíveis em número e atributos com os argumentos de definição do procedimento.

A verificação destas condições pelo compilador é denominada análise de contexto.

A árvore construída pelo analisador sintático é usada para gerar a tradução final do programa fonte. Esta tradução na maioria das vezes é um programa em linguagem de máquina, mas pode ser também um programa em um linguagem intermediária que atenda às necessidades de fases subsequentes, como otimização, por exemplo.

1.2 - Erros e Recuperação

Programas submetidos ao compilador frequentemente têm várias espécies de erros. Um bom compilador deve dar atenção a tantos erros quanto possível. Ao examinar os erros, o compilador precisa ter um modo claro de comunicação com o usuário, fornecendo-lhe comentários em uma linguagem que ele possa entender.

Além de detetar os erros, o compilador precisa também ser capaz de se recuperar, isto é, mesmo na presença de erros, o compilador deve poder analisar todo o programa fonte.

As facilidades de uma boa deteção e diagnóstico requerem um considerável tempo e devem ser consideradas desde o início do projeto do compilador.

1.2.1 - Erros

Um compilador pode reagir de diversas maneiras ao encontrar erros no programa fonte. Óbviamente, existem modos inaceitáveis, como

produzir uma saída incorreta ou simplesmente parar no primeiro erro detetado.

Um compilador simples pode parar algumas de suas atividades após a deteção do primeiro erro, fazendo algum tipo de recuperação de maneira a poder continuar a análise léxica e sintática.

Já os compiladores mais completos e sofisticados podem tentar corrigir o programa errado fazendo alterações que correspondam da melhor maneira possível às intenções do programador, não interrompendo, portanto, nenhuma de suas atividades.

Deteção, recuperação e correção são termos para descrever as possíveis reações do compilador a erros.

A maioria dos compiladores recuperam os erros comuns e continuam a análise sintática e léxica. Alguns compiladores orientados para programas de estudantes, tentam corrigir, de maneira a continuar a compilação do resto do programa fonte e executar o resultante código objeto. O objetivo disso é detetar tantos erros quanto possível em um processamento, incluindo erros detetáveis em tempo de execução.

Nenhum compilador tenta fazer a correção perfeita e existem razões convincentes para isto.

Uma razão é que o compilador ao fazer a correção precisaria conhecer a intenção do programador e as intenções são muito difíceis ou impossíveis de se conhecer, mesmo em se tratando de programas sem erros.

Portanto a correção perfeita somente poderá ser feita pelo programador e isso é um bom motivo para que a maioria dos compiladores não gastem tempo nesta tarefa.

1.2.2 - Fontes de Erro

É difícil conseguir um esquema preciso de classificação de erros de programação. Um modo de classificar erros é de acordo com a maneira como são introduzidos.

Ao se olhar o processo todo de projetar e implementar um programa, pode-se ver que tais erros podem estar em todos os estágios do processo:

- o projeto das especificações para o programa pode ser inconsistente ou falho.

- o algoritmo usado para resolver o problema pode ser inadequado ou incorreto (erros de algoritmo).

- o programador pode introduzir erros na implementação do algoritmo, tanto por introdução de erros lógicos quanto pelo uso de construções impróprias da linguagem (erros de codificação).

- erros de transcrição na perfuração ou teclagem do programa.

- o programa pode exceder os limites da máquina ou do compilador.

A fonte do erro determina como o compilador poderá tratá-lo.

Em erros de algoritmo o compilador pode fazer muito pouco. Já em erros de transcrição existe muito mais redundância e uma maior oportunidade para o compilador fazer uma boa recuperação.

Erros típicos de transcrição são:

- inserção de um caractere ou átomo estranho à linguagem.

- falta de um átomo ou caractere requerido.

- substituição de um átomo ou caractere correto por outro incorreto.

- transposição de dois caracteres ou átomos adjacentes.

Desde que tratemos de linguagens livres de contexto, do ponto de vista do implementador do compilador é conveniente a classificação arbitrária dos erros em sintáticos e de contexto. Define-se erro sintático como sendo aquele que é detetado na fase léxica e/ou sintática do compilador. Outros erros detetáveis pelo compilador são denominados erros de contexto.

1.2.3 - Erros Sintáticos

Na fase léxica, geralmente são detetados erros do seguinte tipo:

- identificador com número de caracteres maior do que especifica à implementação da linguagem.
- presença da caractere não pertencente ao vocabulário.
- identificadores ou números mal escritos.

A deteção ou não de erros deste tipo na fase léxica depende, mais uma vez, de decisões do implementador do compilador sobre átomos, manuseio da tabela de símbolos, etc.

Vejamos agora, alguns exemplos de erros sintáticos, (identificáveis geralmente na fase sintática) ou seja erros que levam à construções imprôprias da linguagem:

a) falta do parêntese direito

$$a \times (a + (2-b)$$

b) vírgula estranha

```
for i := 1 to 10, do
```

c) dois pontos ao invés de ponto e vírgula

```
i: = 1 :
j: = 2 ;
```

d) palavra escrita encorretamente

```
porcedure   a ;
```

e) branco extra

```
w  hile  k > 10  do
```

Em cada um dos exemplos acima, é fácil para o programador determinar o tipo do erro e a posição na qual ele ocorre.

Quase sempre, entretanto, é difícil dizer exatamente onde o erro ocorreu, sem conhecer as intenções do programador. Por exemplo, considere o comando

```
a: = b-c * d + e)
```

Sabe-se que ocorreu um erro, mas não se pode determinar ao certo se é um parêntese direito extra ou se falta algum parêntese esquerdo em um ponto qualquer da expressão.

Muitas vezes, um erro não pode ser detetado assim que ele aparece. Por exemplo, se temos o seguinte comando Pascal

```
ifa = b then  soma:= soma + a ;
```

O erro óbvio é que falta um branco entre o if e a variável a que só será detetado ao se encontrar o "=" e o compilador tentará tratar ifa = b como um comando de atribuição ou chamada de procedimento, supondo que ao invés de '=' deveria aparecer ':=', ou '('.

Este exemplo mostra que a deteção de um erro pode ocorrer em qualquer

ponto depois do lugar onde o erro realmente ocorreu. Para contornar este problema supõe-se que o erro ocorre onde ele for detetado, isto é, a cadeia já processada é considerada correta.

1.2.4 - Erros de Contexto

Os erros de contexto mais comumente detetados em tempo de compilação são os erros de declaração e de escopo. Os exemplos mais típicos são identificadores não declarados ou multiplamente declarados. Retornando a um dos exemplos da seção anterior

```
ifa = b then soma;= soma + a ;
```

provavelmente ao ser encontrado o ifa, teríamos detetado um erro de contexto, ou seja, identificador não declarado. Incompatibilidade de tipo entre operandos e operadores, e entre parâmetros formais e de chamada também pode ser detetada em muitas linguagens, em tempo de compilação. Linguagens como o PL/I definem conversões de tipo que ocorrem automáticamente entre operadores e operandos de tipos diversos. Em tais linguagens fica eliminada a possibilidade da deteção de erros deste tipo.

1.2.5 - Erros Dinâmicos

Em algumas linguagens certas espécies de erros podem ser detetados somente em tempo de execução.

Um exemplo típico de erro detetável em tempo de execução é o campo de validade de certos valores, particularmente subscritos de matrizes

e seletores de comandos do tipo "case".

1.2.6 - Recuperação em cada fase

Como já se viu cada fase do compilador espera que sua entrada siga certas especificações. Quando isto não acontece, esta fase deteta uma inconsistência ou um erro, o qual deve ser relatado ao programador. Entretanto, de modo a continuar o processamento da entrada, a fase tem que recuperar-se a partir de cada erro que deteta.
Portanto pode-se discutir técnicas de recuperação que o analisador léxico, sintático e de contexto podem usar em resposta aos erros encontrados em suas respectivas entradas.
É bom mencionar que depois de detetar e relatar o erro, a fase pode tentar repará-lo ou simplesmente transpassá-lo às fases subsequentes. Se a fase tentar reparar, ela precisa tomar precauções para que não introduza novos erros e consequentemente uma avalanche de mensagens de erro espúrias.
Por outro lado, se a fase transmite erros sem reparo, então as fases subsequentes precisam estar preparados para lidar com a entrada errônea passada adiante.

1.2.6.1 - Fase Léxica (erros de ortografia).

Como já foi visto, a função do analisador léxico é transformar a sequência de caracteres que constitui o programa fonte em uma sequência de átomos. Cada classe de átomos tem uma especificação, a qual é tipicamente um conjunto regular. Se depois de algum proces

samento, o analisador léxico descobre que nenhum prefixo do resto da entrada preenche a especificação de alguma classe de átomos, ela pode chamar uma rotina de recuperação que tomará uma ação remediadora. Infortunadamente, não existirá apenas uma ação remediadora que é a ideal em uma dada situação.

O expediente mais simples é pular caracteres errados até encontrar um átomo. Esta ação pode levar o analisador sintático a detetar um erro de omissão. Pular caracteres indiscriminadamente no programa fonte, pode causar sérias dificuldades para o analisador sintático e demais fases do compilador.

Um outro modo do analisador léxico ter possibilidade de recuperar-se de erros, é fazer com que o analisador sintático forneça à rotina de recuperação de erros léxicos uma lista daqueles átomos que legitimamente podem aparecer no momento, no corrente contexto. A rotina de recuperação pode então decidir se um outro prefixo do resto de sua entrada combina com um desses átomos.

Outros métodos de correção de erros léxicos, ou erros de ortografia como são comumente chamados, encontrados na literatura, são discutidos no próximo capítulo.

1.2.6.2 - Fase sintática

Geralmente, a maior parte da deteção e recuperação de erro em um compilador é centralizada em torno da análise sintática.

Para recuperar-se de um erro, o analisador sintático deve idealmente localizar a posição do erro, corrigir o erro, revisar sua configuração e retornar à análise.

Existem métodos que se aproximam deste ideal e que possibilitam a continuação da análise, mas nunca se pode garantir que o erro foi corrigido com sucesso.

Os métodos de recuperação para erros sintáticos variam dependendo da técnica de análise usada e alguns deles serão vistos com bastante detalhe nos capítulos subsequentes.

1.2.6.3 - Fase de contexto

Já mencionou-se as fontes primárias de erros de contexto: nomes não declarados e incompatibilidade de tipos. O único ponto que vai-se considerar é a recuperação para tais erros e a supressão de mensagens de erros espúrias e/ou duplicadas.

Recuperação para um símbolo não declarado é muito simples. A primeira vez que se encontra um nome não declarado, cria-se uma entrada para este nome na tabela de símbolos, com atributos especiais. Os atributos podem ser determinados pelo contexto no qual o nome é usado. Uma marca nesta entrada indicará que foi criada em resposta a um erro de contexto e não a uma declaração. Um dos atributos de um nome não declarado ou multideclarado pode ser uma lista de maneiras como o nome é errôneamente utilizado. Em todo momento em que o nome é usado incorretamente esta lista é verificada, para se determinar se o mesmo uso já foi préviamente detetado.

Se sim, nenhuma mensagem de erro é impressa se não, uma mensagem de erro é impressa e o novo modo errôneo de uso é adicionado à lista.

No caso de incompatibilidade de tipos, que inclui os erros em chamadas de procedimento com parâmetros incompatíveis, geralmente

o compilador não pode fazer nada, pois neste caso mais uma vez teriam que ser levadas em conta as intenções do programador, o que não é viável.

Portanto neste caso, em geral, o compilador continua o processamento emitindo apenas a mensagem correspondente.

1.3 - Objetivo e Roteiro do Trabalho

O objetivo principal desta pesquisa foi um estudo generalizado e comparativo dos esquemas existentes de recuparação de erros sintáticos aplicáveis à analisadores descendentes.

Para tal, também foram estudados alguns métodos de recuperação de erros de ortografia e de erros em analisadores sintáticos ascendentes. Este estudo é apresentado no segundo capítulo.

O terceiro capítulo tratará especificamente da área de interesse, ou seja, analisadores descendentes. Apresentam-se detalhadamente métodos de se construir um analisador descendente e a descrição dos métodos de recuperação aplicáveis à estes analisadores.

No quarto capítulo apresentam-se os resultados da implementação de alguns dos métodos descritos no capítulo três, em um analisador sintático para a linguagem Pascal.

Os métodos implementados foram testados em um conjunto de programas Pascal, coletadas por Ripley e Druseiks, a partir dos quais eles efetuaram as estatísticas de [Ripley, Druseiks, 1978]

1.4 - Notação Utilizada

Foge do escopo deste trabalho uma apresentação sobre especificação de linguagens de programação, que pode ser vista, por exemplo, em [Kowaltowski, 1979] em que foi baseada a notação descrita a seguir:

Uma gramática livre de contexto G é uma quádrupla G = (T,N,P,S) onde:

1. T é o conjunto de símbolos terminais.
2. N é o conjunto de símbolos não terminais.
3. P é o conjunto de produções.
4. S é a raiz ou símbolo inicial.
6. T e N são disjuntos.

Os símbolos terminais serão denotados por letras minúsculas iniciais do alfabeto latino: a, b, c ... z.

Os símbolos não terminais por letras maiúsculas: A, B, C, D ... Z.

Cadeias de símbolos terminais e/ou não terminais por letras minúsculas gregas: α, β, γ ... . A cadeia nula será denotada por λ.

V denotará o vocabulário de G (V = TUN).

V* será o conjunto de todas as cadeias sobre V e T* o conjunto de todas as cadeias terminais sobre T.

A relação deriva diretamente (denotada por "$\Longrightarrow$") sobre V* é definida por

$$\text{se } \alpha A \beta \in V^* \text{ e } A ::= \gamma ,$$

$$\text{então } \alpha A \beta \Longrightarrow \alpha \gamma \beta$$

Para indicar derivações em n passos ($n \geq 0$) utiliza-se $\overset{*}{\Longrightarrow}$ (deriva).

Caso tenhamos $n > 0$ denotaremos $\overset{+}{\Longrightarrow}$ (deriva não trivialmente).

Se $S \overset{*}{\Longrightarrow} \alpha$, então α é chamada de forma sentencial de G e se $\alpha \in T^*$, então α é uma sentença.

Uma linguagem L(G) definida ou gerada por uma gramática G, é o conjunto de suas sentenças

$$L(G) = \{\alpha \mid S \overset{*}{\Longrightarrow} \alpha \quad e \quad \alpha \in T^*\}$$

A relação deriva diretamente à esquerda ($\underset{e}{\Longrightarrow}$) sobre V* fica definida por:

se $\alpha \in T^*$ e $\beta \in V^*$

e a produção $A ::= \gamma$ está em P,

então $\alpha A \beta \underset{e}{\Longrightarrow} \alpha \gamma \beta$

A relação deriva diretamente a direita ($\underset{d}{\Longrightarrow}$) fica definida de maneira análoga impondo-se $\alpha \in V^*$ e $\beta \in T^*$

Utiliza-se também '$\overset{*}{\underset{e}{\Longrightarrow}}$', '$\overset{*}{\underset{d}{\Longrightarrow}}$', '$\overset{+}{\underset{e}{\Longrightarrow}}$', '$\overset{+}{\underset{d}{\Longrightarrow}}$' para derivações e derivações não triviais, esquerdas e direitas respectivamente.

Se $\alpha = \gamma_0 \Longrightarrow \gamma_1 \Longrightarrow \gamma_2 \Longrightarrow \ldots \Longrightarrow \gamma_n = \beta$, então $\gamma_0\gamma_1\ldots\gamma_n$ é uma derivação de β a partir de α.

Se $\alpha = \gamma_0 \underset{e}{\Longrightarrow} \gamma_1 \underset{e}{\Longrightarrow} \gamma_2 \underset{e}{\Longrightarrow} \ldots \underset{e}{\Longrightarrow} \gamma_n = \beta$, então a sequência $\gamma_0\gamma_1\ldots\gamma_n$ é uma derivação esquerda de β a partir de α. Define-se de maneira análoga derivação direita.

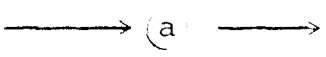
A "análise ascendente" produz a derivação direita, a partir de S, de uma cadeia de entrada β, e a análise descendente a derivação esquerda análoga.

Utilizaremos também grafos sintáticos para representar gramáticas. O uso desta notação está comumente ligada ao uso dos métodos descendentes de análise sintática.

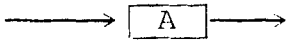
Nesta representação, a cada não terminal da gramática corresponde um grafo orientado cujos nós são rótulados com os símbolos terminais e

não terminais da gramática. Este grafo é obtido através de um conjunto de regras enumeradas a seguir:

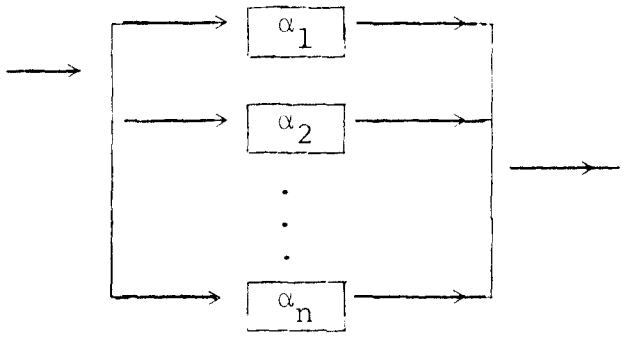
1. Um símbolo terminal a é representado por:



2. Um símbolo não terminal A é representado por:

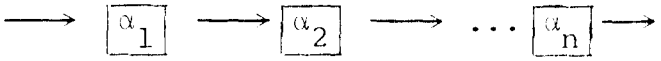


3. Uma construção $\alpha_1|\alpha_2|\ldots|\alpha_n$ é representada por:



onde todo $\boxed{\alpha_i}$ indica o grafo obtido para α_i.

4. Uma construção $\alpha_1\alpha_2\ldots\alpha_n$ é representada por:



onde $\boxed{\alpha_i}$ indica o grafo obtido para α_i.

A cadeia nula λ é indicada por uma simples aresta.

CAPÍTULO 2

2 - Alguns Métodos de Recuperação

Neste capítulo faremos uma referência sucinta à alguns métodos mais significativos de recuperação de erros encontradas na literatura e que não são estritamente aplicáveis à analisadores descendentes.

O objetivo é fazer com que o leitor se situe melhor dentro da área e obtenha referências caso deseje se aprofundar em algum dos métodos.

O capítulo foi subdividido em duas partes principais. A primeira tratará de uma área específica de recuperação de erros, que é a de recuperação de erros de ortografia ou transcrição, já mencionados no capítulo anterior. A outra parte tratará de métodos de recuperação de erros sintáticos aplicáveis à analisadores ascendentes.

2.1 - Correção de erros de ortografia

Existem diversas técnicas especializadas para incorporar eficientemente correção de erros de ortografia em compiladores. Elas geralmente incluem o uso de informações sintática e de contexto, uma organização especial da tabela de símbolos, e consideram uma classe limitada de erros de ortografia.

O uso de sistemas que executam a correção de ortografia implica numa razoável diminuição no número de processamentos para depuração, economizando portanto, tempo de máquina e do programador.

Um dos últimos artigos sobre correção de erros de ortografia em compiladores é [Morgan, 1970]. O leitor interessado deve usá-lo como referência para outros artigos na mesma área.

Existem várias situações onde o compilador pode suspeitar que um identificador ou palavra reservada está escrita de maneira incorreta, e essas situações geralmente estão associadas com as fases da compilação. Portanto, diferentes tipos de erros de ortografia podem ser corrigidos em cada uma dessas situações, e a informação, a qual é necessária para o processo de compilação, pode ser usada adicionalmente no controle do algoritmo de correção.

Vamos ver brevemente quais são algumas dessas situações:

1 - Durante a análise sintática sempre sabe-se, se o próximo símbolo precisa pertencer a um subconjunto de palavras reservadas. Caso isto aconteça, se aparecer um identificador então pode-se verificar se houve um erro de ortografia na escrita de uma das palavras reservadas esperada.

Essa situação ocorre em linguagens onde, por exemplo, todo comando começa com uma palavra chave. Um outro caso é quando da análise de uma expressão boleana sabe-se que um operador como o and ou or precisa aparecer.

Pode-se, também, tentar verificar um erro de concatenação quando uma palavra reservada é esperada. Por exemplo, se é esperada um begin e encontra-se o identificador begina, ele pode ser alterado para begin a.

2 - Durante a análise de contexto, suponha-se que um identificador declarado como rótulo é usado em um contexto onde apenas um

nome de matriz pode aparecer. Então o identificador pode estar mal escrito e deverá ser comparado com a lista de nomes de matrizes declaradas.

Situações similares, onde o contexto determina o tipo de identificador esperado ocorrem com frequência.

3 - Por causa de um erro de ortografia, um identificador pode aparecer uma ou duas vezes sem que lhe seja feita uma atribuição de valor ou uma referência ao seu valor. Isto pode ser verificado facilmente; coloca-se um contador de atribuições e um contador de referências como atributos em cada entrada da tabela de símbolos. Depois da análise sintática e de contexto, percorre-se as entradas da tabela de símbolos e qualquer entrada com um dos cortadores igual a zero é um candidato à uma correção de ortografia. Também, pode-se considerar os identificadores não declarados, em linguagens do tipo Pascal, como candidatos a este tipo de correção.

A próxima questão a se fazer é como determinar qual identificador foi mal escrito.

O primeiro trabalho nisso, no contexto de compiladores, aparece em [Freeman, 1963]. A técnica de Freeman estima a probabilidade de um identificador ser um outro mal escrito, baseada em uma complexa função de "pontos". Esta função usa informações tais como o número de letras que combinam, o número de letras que combinam depois de uma ou duas transposições de caracteres e o número de letras que combinam depois de se levar em conta erros de perfuração que frequentemente ocorrem, como por exemplo, Ø (zero) por O , 1 por I, etc. Esta técnica identifica corretamente muitos erros de ortografia, mas é muito ineficiente devido ao grande número de possibilidades testadas. Morgan [1970]

apresenta uma adaptação da técnica de Freeman, mais eficiente embora menos poderosa. Ele baseia-se na evidência que cerca de 80% dos erros de ortografia em programas se encaixam em uma das seguintes quatro classes:

1 - um caractere errado

2 - um caractere omitido

3 - um caractere extra inserido

4 - dois caracteres adjacentes transpostos.

Portanto a maioria dos erros de ortografia pode ser corrigida verificando-se estes tipos de erros. Em Morgan [1970] pode-se ver detalhadamente esta técnica, e vamos sòmente dar uma visão geral do procedimento de correção.

Existem dois estágios básicos:

1º estágio: A partir da tabela de símbolos (chamada dicionário) seleciona-se um subconjunto contendo todos os símbolos que poderiam ser o identificador mal escrito, que é denominado palavra de entrada.

2º estágio: Para cada símbolo do subconjunto gerado no estágio anterior é aplicado um algoritmo de comparação, baseado nas quatro classes de erro, o qual determina se a palavra de entrada é ou não um erro desse símbolo.

Supondo que o segundo estágio do procedimento tem um tempo aproximadamente constante t2 para cada símbolo do subconjunto, o tempo máximo do algoritmo é mt2 onde m é o número de símbolos do subconjunto. Portanto reduções significativas tanto em m como em t2 implicam em melhor eficiência do procedimento, levando-se em conta o fator tempo.

Limitando a classe de erros pode-se tanto diminuir m como t2. Desde que apenas erros de caracteres simples e de transposição são considerados, qualquer palavra do dicionário cujo tamanho difere de mais de um caractere do tamanho da palavra de entrada é rejeitada imediatamente, diminuindo m. Também, limitando o número de categorias de erro, simplifica-se o algoritmo de comparação, reduzindo t2.

O preço pago por esta simplificação no caso de Morgan, é o de que cerca de 20% dos símbolos errados não serão corrigidos e serão considerados erros.

Com o uso de informações sintáticas e de contexto consegue-se diminuir m, pois apenas os símbolos que podem fazer sentido no atual estado do compilador são comparados com a palavra de entrada.

Um outro modo de melhorar o desempenho do procedimento é utilizar-se métodos eficientes de acesso a tabela de símbolos. Geralmente usa-se alguma função de espalhamento (em inglês, "hashing") para pesquisar ou armazenar elementos na tabela de símbolos. Certas funções de espalhamento são mais úteis que outras quando a correção de ortografia é executada considerando-se apenas a limitada classe de erros descrita anteriormente, pode-se ver que apenas as palavras do dicionário que possuem o primeiro ou segundo caractere iguais ao primeiro ou segundo caractere da palavra de entrada necessitam ser incluídas no subconjunto gerado no primeiro estágio do procedimento.

Por exemplo, se a palavra de entrada é stpo as palavras do dicionário que devem ser considerados são:

a) aquelas que se iniciam com
   s ou t, e

b) aquelas cujo segundo caractere é s ou t.

Para localizar fácilmente tais palavras no dicionário poderia-se usar uma função de espalhamento da forma

$$f_1(a_1) + f_2(a_2) + f_3(a_3, a_4 \ldots a_n) \text{ onde}$$

$a_1, a_2 \ldots a_n$ são os caracteres que compõem a palavra de entrada e f_1, f_2, f_3 possuem um intervalo de variação de valores apropriado para o que se deseja.

Utilizando-se dessas melhorias adicionais, o método de Morgan tem-se mostrado eficiente nas implementações efetuadas.

Em [Wagner, 1974] descreve-se um método de recuperação para linguagens regulares, ou seja, aquelas que podem ser reconhecidas por autômatos finitos. O critério de correção que ele investiga é o da "distância mínima de edição". Ele define um conjunto de operações de edição, as quais podem ser aplicadas à um símbolo fonte para modificá-lo, que são:

- substituir um caractere por outro
- inserir um caractere
- apagar um caractere

O critério da mínima distância se preocupa em transformar o símbolo de entrada em um sintáticamente correto, que possa ser gerado com o mínimo possível de operações de edição.

Seu método corrige erros de ortografia em palavras reservadas e em nomes introduzidos pelo programador. Ele constrói um autômato finito o qual pode aceitar, por exemplo, uma palavra reservada válida e então aplica o algoritmo de correção, guiado por este autômato, para um

dado símbolo de entrada.

2.2 - Recuperação de erros sintáticos para analisadores ascendentes

2.2.1 - Aspectos gerais da análise ascendente

Como já foi dito, nos algoritmos de análise sintática ascendente, a construção da árvore de derivação para uma dada cadeia , começa pelas folhas da árvore e procede na direção de sua raíz. Caso seja obtida uma árvore cuja raíz tenha por rótulo o símbolo inicial da gramática, e na qual a sequência dos rótulos das folhas forma a cadeia dada, então a cadeia é uma sentença da linguagem, e a árvore obtida é a sua árvore de derivação.

O funcionamento básico de um algoritmo ascendente pode ser descrito da seguinte maneira:

1 - Adota-se como o valor inicial de α a cadeia dada.

2 - Procura-se decompor α de tal maneira que $\alpha = \beta x_1 x_2 \ldots x_n \gamma$, e existe uma produção da forma $X: = x_1 x_2 \ldots x_n$. Caso isto seja possível adota-se a nova cadeia $\alpha = \beta\ X\ \gamma$, associando-se com esta ocorrência do não terminal X, uma árvore cuja raíz tem rótulo X, e cujos subárvores diretas são as árvores que estavam associadas com as ocorrências de x_1, x_2 ... x_n que foram substituídas. Se x_i é um terminal, então a árvore associada será uma folha de rótulo x_i. Nota-se que este processo é o inverso de uma derivação, e é chamado de redução.

3 - O passo 2 é repetido até que o valor de α seja o símbolo inicial da gramática.

É bom observar que o analisador sintático não precisa, em geral conhe

cer a cadeia α para decidir qual é à próxima redução; é suficiente conhecer uma parte inicial α' de α que é decomposta em $\alpha'=\beta x_1 x_2 \ldots x_n \gamma'$. À medida que for necessário, novos símbolos serão buscados pelo analisador, por exemplo lidos, estendendo a cadeia α'.

Observando-se o algoritmo anterior, pode-se concluir que os problemas básicos da análise ascendente são:

(1) identificação da parte da cadeia que deve ser reduzida, denominada redutendo (em inglês, "handle").

(2) identificação da produção que deve ser usada na redução

A maneira de encontrar o redutendo pode variar para classes de gramáticas diferentes. As classes cujos métodos são mais conhecidos são as de precedência simples, precedência de operadores e LR.

Foge do escopo deste trabalho uma visão detalhada dos algoritmos de análise para cada classe de gramáticas, que pode ser encontrada em [Kowaltowski , 1979], [Gries, 1971] e [Aho e Ullman, 1977].

2.2.2 - Métodos de Recuperação de erros sintáticos

Em analisadores ascendentes, geralmente um erro é detetado de dois modos:

- quando não se consegue determinar um redutendo, e
- quando não se tem uma produção para se efetuar a redução de um dado redutendo.

Suponha-se que em algum ponto da análise o programa tem a forma

$$\alpha \; t \; \beta$$

onde α representa a parte já processada. Em uma pilha, denominada pi

lha de análise, geralmente tem-se armazenada uma cadeia α' de terminais e não terminais que representa α. $\underline{t}$ é o próximo simbolo de entrada, e $\underline{\beta}$ é o resto da cadeia de entrada.

Suponha-se que um erro ocorra. Então dependendo da classe de gramáticas analisada pode ser que t não possa aparecer depois de α, segundo a definição da gramática, ou que não se tenha um redutendo no final de α', ou alguma situação similar.

Neste ponto é preciso determinar-se como mudar a cadeia de entrada , de maneira a reparar o erro. Ela pode ser mudada de uma das maneiras seguintes, ou uma combinação delas:

1 - Apagar t e continuar a análise

2 - Inserir uma cadeia de terminais
γ entre α e t, obtendo-se $\alpha\gamma t\beta$
e prosseguir a análise. Esta inserção deve permitir que se processe γ t por completo, antes de ocorrer um outro erro.

3 - Inserir uma cadeia γ na pilha de análise e reiniciar a análise a partir de t.

4 - Apagar alguns símbolos do topo da pilha de análise.

Os métodos 3 e 4 são muito indesejáveis e devem ser evitados pelas seguintes razões. Desde que α foi processada, existe uma informação de contexto associada a ela, que está contida em α'. Adicionar γ à α', ou apagar parte de α', significa que se precisa alterar de maneira conveniente a informação de contexto, e isto não é fácil de ser feito.

Não alterando-se a pilha de análise, não é preciso alterar qualquer

contexto, dai os métodos 1 e 2 serem preferidos.

Note-se que pode-se fazer o equivalente a 3 ou 4 por inserção de uma cadeia de terminais (método 2). Por exemplo, suponha que tem-se:

$$\alpha' = \ldots \underline{\text{then}}\ \underline{\text{begin}}\ \langle\text{comando}\rangle$$

e t = else

Uma possível correção seria substituir-se begin <comando> por <comando> (utilizando-se uma combinação de 3 e 4).

O equivalente a isto pode ser conseguido inserindo-se end entre α e t e teríamos $t\beta$ = end else ... e a análise prosseguiria normalmente.

Também pode ser viável adicionar regras à gramática que tratam os erros. Por exemplo, poderia-se adicionar uma regra do tipo

<comando de atribuição> :: = : = E

para tratar o caso de se esquecer da variável à esquerda do ': =' .

Entretanto a gramática tende a, rápidamente, tornar-se muito grande, além de ser muito trabalhoso alterar de forma conveniente, para tal, o algoritmo de análise.

Em analisadores ascendentes de uma maneira geral, é difícil obter alguma informação útil sobre o que pode aparecer na cadeia de entrada, e portanto, não se pode usar fácilmente o contexto global, e tem que se contar com o contexto local, imediatamente ao redor do ponto do erro.

Em [Gries, 1971] é descrito um método de recuperação, similar à técnicas usadas em vários analisadores, consideravelmente primitivo.

O programador do compilador inicia um conjunto de símbolos C, com símbolos importantes tais como o ';' e o end. Quando um erro é detetado o seguinte acontece:

1 - Os símbolos em tβ são examinados e descartados até que seja encontrado um que pertença ao conjunto C.

2 - Tem-se então um novo símbolo t.

Os símbolos do topo da pilha de análise são examinados e descartados até que o símbolo t possa seguir legalmente o que restar em α'.

Um segundo método, também descrito em [Gries, 1971] trabalha bem com a técnica de matriz de transição descrita em [Gries, 1968]. Esta técnica usa alguns símbolos α'_1 do topo da pilha de análise, considerando-se $\alpha' = \alpha'_2\ \alpha'_1$, os quais formam a parte inicial do lado direito de uma produção e servem para selecionar a linha i da matriz de transição M; o símbolo de entrada t determina a coluna j. O elemento $M_{i,j}$ é o endereço do procedimento o qual ou empilha t e pega o próximo símbolo da entrada, ou faz uma redução.

O interessante desta técnica, é que usualmente cerca da metade dos elementos da matriz representam pares ilegais (α'_1,t) e então pode-se detetar muitas condições de erro diferentes.

O esquema de recuperação que Gries apresenta pode ser feito através de um "construtor", o qual, dada uma determinada gramática, constrói os procedimentos de recuperação de erro e preenche as entradas da matriz com seus nomes.

Portanto, as rotinas de recuperação de erro não serão baseadas no contexto; elas recuperam sómente baseadas em α'_1 e t, de uma maneira predeterminada.

Relembrando, que é mais conveniente recuperar tão somente descartando t ou inserindo-se uma cadeia terminal γ entre α e t, o construtor

deve usar os seguintes critérios para determinar como recuperar para um dado par (α_1', t).

1 - Se existe uma produção $U::= \alpha_1'$ At.... na gramática, uma cadeia terminal γ pode ser inserida tal que $A \overset{*}{\Rightarrow} \gamma$

2 - Se existe uma regra $U::= \alpha_1'$ V ....
Tal que $V \overset{*}{\Rightarrow} Zt$, uma cadeia terminal γ pode ser inserida tal que $Z \overset{*}{\Rightarrow} \gamma$

3 - Se existe uma regra U:: = ... Vt....
Tal que $V \overset{*}{\Rightarrow} .... W\delta_2$ e $W::= \alpha_1' \delta_1$ é uma regra, uma cadeia terminal γ pode ser inserida tal que $\delta_1\delta_2 \overset{*}{\Rightarrow} \gamma$

4 - Se nenhuma das regras anteriores se aplica, t pode ser descartado.

Essas regras como poderemos verificar no próximo capítulo, são muito similares àquelas usadas em analisadores descendentes, mas menos contexto é utilizado para tomar a decisão e isto implica na recuperação não ser tão eficiente.

Wirth [1968] descreve um esquema para recuperação em analisadores de precedência simples, implementado no compilador PL360.

Um ponto importante a ser observado é que uma construção incorreta deve ser detetada tão logo ela ocorra, isto é, antes que futuros passos sejam dados com base no texto.

A técnica para gramáticas de precedência simples é um excelente esquema neste aspecto, pois é baseada nas relações existentes entre pares de símbolos da linguagem. Se nenhuma das relações, denotados por $\lessdot$, $\doteq$ e $\gtrdot$, existirem entre dois símbolos isto implicará na impossibilidade destes dois símbolos serem adjacentes em qualquer forma sentencial da linguagem. A relação vazia, denotada por (·) pode ser

definida como válida quando nenhuma das outras o for.

Ao percorrer-se a cadeia de entrada da esquerda para direita, o encontro de uma relação vazia é o ponto exato de deteção de uma construção errônea. O algoritmo para diagnóstico e recuperação apresentado por Wirth [1968] não é baseado em princípios teóricos, sendo uma solução heurística.

Existem dois pontos no processo de análise, onde construções ilegais podem ser detetadas. Considerando-se o conteúdo α' da pilha de análise:

1 - a relação vazia existe entre o símbolo q do topo da pilha de análise e o símbolo de entrada t, então $q \odot t$

Neste caso uma lista I de símbolos de inserção é percorrida.

Se para algum $m \in I$, $q \not\odot m$ e $m \not\odot t$, então m é inserido antes de t e a análise prossegue. Se nenhum m é encontrado que satisfaça estas condições, então t é empilhado.

Não é dito o que é feito a seguir neste caso, mas provavelmente a análise continua, sendo que será detetado um erro do tipo 2 em passos subsequentes.

2 - Não existe nenhuma regra para a qual a cadeia α'_1 correspondente aos símbolos no topo da pilha de análise é parte direita.

Neste caso uma tabela de produções de erro é percorrida até encontrar uma com lado direito igual a α'_1. Se é encontrada, a mensagem de erro correspondente à regra é impressa e a análise continua, desempilhando α'_1. Caso contrário, símbolos da entrada são descartadas, até poder

ser eliminada a condição de erro.

Não existe uma definição formal de como se escolhe os símbolos de inserção e as produções de erro adequadas, sendo a escolha baseada num profundo conhecimento do algoritmo de análise por parte do projetista do compilador e também num conhecimento antecipado de erros frequentes na sintaxe. É claro que futuros símbolos de inserção e produções de erro podem ser fácilmente adicionadas de maneira a aumentar a capacidade de diagnóstico do compilador, e se este é capaz de obter informações estatísticas sobre situações errôneas encontradas, estas informações podem ser avaliadas de tempos em tempos para expandir o conjunto de símbolos de inserção e produções de erro.

James e Partridge [1973] aplicam métodos comumente utilizados na área de inteligência artificial ao problema de recuperação de erro. A cadeia de entrada é comparada com a descrição da linguagem, usando uma estratégia descrita como uma árvore binária. Cada nó da árvore representa um estágio do processo de análise e contém os seguintes campos:

1 - um código de estratégia que irá especificar a ação de recuperação a ser iniciada caso o processo de comparação falhe.

2 - os enderêços dos nós associados, são usualmente dois : o primeiro indicando o próximo nó a ser usado caso a comparação tenha sucesso, e o outro o nó alternativo, caso contrário.

3 - o caractere a ser testado no processo de comparação ou o nome de um procedimento, que descreve uma subárvore, a ser chamado.

As estratégias no caso de erro, são procedimentos, que de uma maneira geral, fazem coisas do tipo : "suponha que um caractere foi esque

cido na cadeia de entrada", "descarte um caractere da cadeia de entrada e tente comparar o novo caractere com o corrente caractere na árvore sintática", etc.

Este esquema funciona a nível microscópico, ou seja, os nós da árvore binária contém caracteres individuais e não átomos sintáticos, o que adiciona grande poder ao método, mas também acrescenta um tempo extra na compilação de programas corretos, o que não é desejável e que normalmente não ocorre com os outros métodos.

Graham e Rhodes [1975] apresentam um método de recuperação testado para linguagens do tipo LR(K).

Analisadores para linguagens do tipo LR(K) e seus variantes (SLR(K), LALR(K), etc.) possuem a propriedade (já mencionada no caso dos analisadores de precedência simples) de prefixo correto, ou seja, tem-se sempre assegurado que nenhuma porção do texto de entrada é desnecessáriamente percorrida a partir do ponto em que um símbolo incorreto é lido, ou seja, a porção do texto de entrada já processada constitui um prefixo válido para algum programa.

No método de Graham e Rhodes, primeiro é analisado o contexto no qual o erro ocorre, ou seja, a fase de correção é precedida por uma fase de condensação que resume o contexto circunvizinho ao ponto de erro.

Na fase de condensação primeiro tenta-se fazer reduções dentro da pilha de análise, precedendo o ponto de deteção do erro, que é marcado por um símbolo especial guardado na pilha.

Depois disso, tenta-se analisar a entrada imediatamente após o ponto de deteção do erro. Esta análise termina quando ocorre um erro em um ponto futuro da entrada, ou porque a única ação possível de análise é uma redução envolvendo parte da pilha que contém o erro detetado.

O propósito da fase de correção é alterar a pilha de análise condensada, de maneira que a situação de erro seja corrigida e consequentemente a pilha de análise contenha uma sequência de símbolos que pode ocorrer na análise de uma sentença da linguagem. Geralmente, existirá mais de uma possível mudança que irá corrigir o erro.

Considere-se o exemplo:

```
.
.
.
m: = q - 3 ;
i = 2* (m-p) then K: = 1 else m: = 1 ;
.
.
.
```

O erro mais provável é que foi esquecido um if antes do identificador i, e o erro é detetado quando o '=' é visto pelo analisador.

Pelo método de Graham e Rhodes, teríamos a fase de condensação que analisaria até o símbolos then, tendo-se como próximo símbolo de entrada o identificador K. Duas mudanças são cabíveis no caso:

(1) considerar-se i = 2 * (m-p) then como "<comando>;" (descartando-se então a expressão i = 2 * (m-p) e o then) que pode ser seguida por um "<identificador>" ; ou

(2) inserir-se um if , obtendo-se if i = 2 * (m-p) then , que pode seguir o comando m: = q-3 ;

O critério usado para decidir qual correção deve ser aplicada é o de utilizar a alteração que requera a mínima modificação símbolo a símbolo na pilha de análise, portanto é utilizado o critério de "distância mínima". Para tal são utilizados dois conjuntos I e D.

Para cada símbolo na gramática, o conjunto I contém o custo de inserir e o conjunto D o custo de eliminar este símbolo.

No caso da modificação (1) do exemplo anterior teríamos que o custo seria dado por:

D(<expressão>) + D(then) + I(<comando>) + I(;)

e no caso 2 por:

I(if)

sendo que seria escolhida a alteração de menor custo. Caso o menor custo seja maior que um custo limite determinado "à priori", símbolos da entrada são descartados até que se altere tal situação.
São definidas várias heurísticas que permitem os conjuntos I e D serem determinados mecânicamente.

Musinski [1977] apresenta uma melhora nos esquemas de recuperação geralmente existentes para analisadores de linguagens do tipo LALR que descartam símbolos da pilha de análise de maneira a eliminar a situação de erro, e quando isto não ocorre a pilha é restaurada e símbolos da entrada são descartados. Isto, inevitavelmente gera uma torrente de erros espúrios.

O que Musinski [1977] propõe é que símbolos da pilha de análise sejam descartados até que se encontre um estado que exija que se percorra a entrada, e a partir daí são descartadas símbolos da entrada. O percurso da entrada, quase sempre irá parar quando um símbolo demarcador ('end', ';', etc.) da linguagem for encontrado, reiniciando-se então o processo normal de análise. Em implementações realizadas notou-se que foram eliminadas todas as mensagens de erro espúrias, sendo obtida uma grande melhoria em relação aos métodos existentes.
La France [1970] descreve um método de recuperação para linguagens de

produções de Floyd, procurando mostrar que a sintaxe, a qual é usada para dirigir a análise também pode ser utilizada para dirigir a análise de erros, e que de fato os resultados obtidos desta maneira são superiores àqueles obtidos por rotinas de tratamento de erro construídas "ad hoc".

Conway e Wilcox [1973] descrevem a implementação de um compilador PL/C que é um dialeto do PL/I. O compilador utiliza a técnica de matriz de transição e como método de recuperação o descrito por Gries [1971].

CAPÍTULO 3

3 - Análise sintática descendente e métodos de recuperação de erros

Neste capítulo são apresentados, de maneira detalhada, os diversos tipos de implementação de análise descendente e os métodos existentes de recuperação de erros, específicos para este tipo de análise.

Procurou-se apresentar em sequência os métodos de recuperação que apresentam similaridades, isto para facilitar um estudo comparativo.

3.1 - Métodos de implementação de análise descendente

Nesta primeira seção descreveremos diversos métodos de fazer análise sintática descendente. O primeiro é o mais genérico, mas muito ineficiente. Os outros são específicos para linguagens do tipo LL(1), sendo que o segundo método foi o utilizado em nossa implementação.

As descrições dos métodos apresentados a seguir foram extraídas de [Kowaltowski, 1979] e [Aho e Ullman, 1977].

3.1.1 - Implementação com Retrocesso

Esta é a forma geral de um analisador descendente, que apesar de muito ineficiente ainda é usada em algumas aplicações especiais.

Por retrocesso entende-se repetidas análises de uma mesma porção da entrada.

Análise descendente é uma tentativa de encontrar a derivação esquerda para uma dada cadeia de entrada. Isto pode ser visto como uma tentativa de construir a árvore de derivação para uma cadeia de entrada, começando pela raíz e criando os demais nós em preordem.

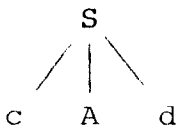
Por exemplo, considere-se a gramática:

$$S :: = cAd$$

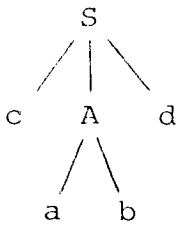
$$A :: = ab \mid a$$

e a entrada α = cad

Para construir a árvore de derivação, pelo método descendente, inicialmente cria-se um único nó, rotulado S. Usa-se agora a primeira produção de S para expandir a árvore e obtém-se



Como a folha mais à esquerda, rotulada c, casa com o primeiro símbolo de α, avança-se então para o segundo símbolo de α, e considera-se a próxima folha da árvore, rotulada A. Tenta-se então expandir A usando a primeira alternativa para A e obtém-se:

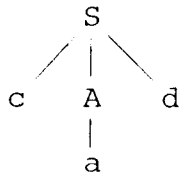


Agora casa-se o segundo símbolo de α e avança-se para o símbolo d, e

considera-se a próxima folha rotulada b.

Verifica-se então que b não casa com d, tem-se então uma falha e retrocede-se para A, a fim de verificar se existe uma outra alternativa para A, que ainda não foi tentada e que possa dar sucesso.

Ao se retornar para A, também é preciso retroceder-se na cadeia de entrada, para o segundo símbolo a . Tenta-se então a segunda alternativa para A e obtém-se a árvore:



A folha a casa com o segundo símbolo de α e a folha d casa com o terceiro símbolo. Desde que conseguiu-se construir a árvore de derivação para α, tem-se o término da análise com sucesso. Se em alguma etapa da análise, retrocede-se à configuração inicial, depois de esgotar-se todas as alternativas para o símbolo inicial da gramática, então a análise termina com insucesso e α não pertence à linguagem.

Pode-se ver algumas dificuldades associadas com o método exposto.

1 - Sua implementação pode tornar-se bastante complicada, pois em cada passo será necessário conhecer a cadeia de entrada e a árvore (ou pelo menos sua fronteira) correntes. Além disso, tem que se guardar todas as configurações anteriores em que foram tomadas decisões sobre a escolha de alternativas, para poder executar o retrocesso. Isto requer uma estrutura de dados bastante elaborada.

2 - ele pode ser extremamente ineficiente devido à possibilidade de haver retrocessos.

O número de operações para analisar uma cadeia pode chegar a ser uma

função exponencial de seu comprimento.

3 - geralmente as decisões sobre as alternativas entre as produções acarretam certas ações por outras partes do compilador, como por exemplo, geração de instruções de código objeto, ou modificação das tabelas internas. O efeito dessas ações terá que ser anulado caso haja retrocesso, e isto, em geral, é bastante complicado.

Os problemas que acabamos de citar, fazem com que análise descendente seja usada quase que exclusivamente quando se pode eliminar retrocessos.

Existem algumas maneiras práticas de resolver os problemas da análise descendente, tornando o método eficiente, e aplicável à muitas linguagens de programação.

A maneira mais simples de evitar retrocessos é fazer com que o algoritmo sempre tome a decisão correta quanto à produção a ser aplicada. Uma classe muito simples de gramáticas para as quais isto pode ser feito é obtida impondo-se as restrições:

1 - Toda produção é da forma $A :: = a\alpha$,

onde

2 - Se $A :: = a_1\alpha_1 | a_2\alpha_2 | \ldots | a_n\alpha_n$ são todas as alternativas para o não terminal A, então os terminais a_i são todos distintos entre si.

É suficiente, então, modificar ligeiramente o processo de análise descrito, fazendo com que a escolha da produção seja baseada no primeiro símbolo da cadeia α. É óbvio que poderá haver no máximo uma alternativa que deverá ser escolhida. Caso não haja nenhuma escolha, isto é, α não começa com nenhum dos símbolos a_i que correspon-

dem a um dado não terminal A da folha corrente, então a cadeia não é uma sentença da linguagem.

Estas restrições são muito severas para serem viáveis na prática, mas podemos generalizar a idéia obtendo uma classe mais ampla de gramáticas.

Vamos, para isso utilizar uma definição auxiliar. Seja X um símbolo terminal ou não terminal. Então $\psi(x)$ contém todos os símbolos terminais y tais que $x \stackrel{*}{\Longrightarrow} y\alpha$ para algum α.

Em particular, se X é um terminal, então $\psi(x) = \{X\}$.

A restrição a ser imposta agora é:

Se $A::= x_1\alpha_1 | x_2\alpha_2 | \ldots | x_n\alpha_n | \lambda$ são todas as alternativas para o não terminal A, então os conjuntos $\psi(x_i)$ são disjuntos dois a dois. Note-se que os X_i podem ser tanto terminais como não terminais e que λ pode ou não ser uma das alternativas.

O algoritmo de análise continua muito simples só que agora ele deve verificar a qual dos conjuntos $\psi(X_i)$ pertence o primeiro símbolo da cadeia α. Como os conjuntos devem ser disjuntos haverá no máximo uma alternativa possível.

Caso, se tenha a alternativa $A::= \lambda$, deve-se tomar o cuidado para que ela não seja verificada antes de que todas o tenham sido, e então caso todas as outras falhem, ela é adotada.

A classe de gramáticas que satisfaz esta restrição é denominada LL(1) (em inglês, Left - to-right parsing producing Leftmost derivation).

O nome vem do fato de se poder analisar uma cadeia da esquerda para direita, produzindo uma derivação esquerda, verificando apenas um símbolo da cadeia de entrada para decidir qual é a produção a ser apli

cada. A definição pode ser generalizada para LL(k), $k \geq 1$. Para tais gramáticas podem-se obter analisadores descendentes sem retrocesso, se a escolha da produção a ser aplicada for baseada nos k primeiros símbolos da cadeia de entrada corrente.

Toda gramática do tipo LL não possui recursão à esquerda, sendo esta uma característica importante na construção do analisador descendente.

Uma gramática G tem recursão esquerda se para um não terminal A existe uma derivação $A \Longrightarrow^{*} A\alpha$ para algum α Uma gramática com recursão esquerda leva o analisador descendente à uma repetição infinita, isto é, pode-se expandir A e eventualmente expandi-lo novamente, sem que se tenha percorrido a entrada.

Vamos ver rápidamente como eliminar a recursão esquerda.

Se temos um par de produções com recursão esquerda da forma:

$$A :: = A\alpha \mid \beta$$

onde β não se inicia com A, então podemos eliminar a recursão esquerda substituindo-se este par de produções por:

$$A :: = \beta A'$$

$$A' :: = \alpha A' \mid \lambda$$

De uma maneira geral então, se tivermos

$$A :: = A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_n$$

onde nenhum β_i começa com A, elimina-se a recursão esquerda substituindo-se as produções por:

$$A :: = \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$$
$$A' :: = \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \lambda$$

Este processo pode eliminar todas as recursões esquerdas diretas (supondo-se que $\alpha_i \neq \lambda$), mas não elimina recursão esquerda envolvendo derivações de um ou mais passos.

Por exemplo, considere

$$S :: = Aa | b$$
$$A :: = Ac | Sd | e$$

O não terminal S tem recursão esquerda pois $S \Rightarrow Aa \Rightarrow Sda$.

Uma maneira de resolver este problema seria: primeiro eliminar-se todas as produções de A na forma $A :: = S\alpha$, obtendo-se

$$A :: = Ac | Aad | bd | e$$

e agora elimina-se a recursão direta, resultando a seguinte gramática:

$$S :: = Aa | b$$
$$A :: = bdA' | e\ A'$$
$$A' :: = cA' | adA' | \lambda$$

Um algorítmo, que resolva o problema desta forma pode ser construído,

e funcionará se a gramática não tiver ciclos (derivações da forma $A \Longrightarrow^{+} A$) ou produções vazias ($A :: = \lambda$). O algoritmo pode ser visto em [Aho e Ullman, 1977].

Pode-se notar que um problema que surge com este tipo de modificação, é o aumento do comprimento das derivações o que será refletido num maior número de operações para realizar a análise.

Este problema pode ser minorado, adotando-se uma notação estendida para gramáticas, e modificando convenientemente o algoritmo de análise.

Suponha-se que algumas alternativas para o não terminal A tem a forma:

$$A :: = \beta\gamma_1 | \beta\gamma_2 | \ldots | \beta\gamma_n$$

$\beta \neq \lambda$. Pode-se fatorar estas produções escrevendo:

$$A :: = \beta(\gamma_1 | \gamma_2 | \ldots | \gamma_n)$$

Caso $\gamma_i = \lambda$ para algum i, coloca-se esta alternativa em último lugar, isto é, $\gamma_n = \lambda$.

Usando esta notação, o algoritmo poderá adiar a decisão sobre a escolha de alternativa até que seja reconhecida a parte comum derivável de β.

O problema da recursão também pode ser contornado adotando-se a substituição da notação recursiva por uma iterativa. Assim, a ocorrência da construção $\{\alpha\}$ numa produção denotará a repetição da cadeia α zero ou mais vezes, ou seja, um elemento de α^*.

Por exemplo, a seguinte produção recursiva:

$$E :: = E + T | T$$

será escrita como

$$E :: = T\{+T\}$$

Em alguns casos mais complicados, esta notação poderá ser combinada com a fatoração.

Assim

$$A::= Ac|Aad|bd|e$$

pode ser reescrita como:

$$A::= (bd|e)\{(c|ad)\}$$

De uma maneira geral se existirem produções da forma

$$A::= \beta\gamma_1|\beta\gamma_2|\ldots|\beta\gamma_n \ , \ \beta \neq \lambda$$

deve-se substituí-las por

$$A::= \beta(\gamma_1|\gamma_2|\ldots|\gamma_n)$$

Desta maneira existirá uma única alternativa com recursão esquerda.

Se todas as alternativas para a não terminal A são dados por:

$$A::= \gamma_1|\gamma_2 \ \ldots| \ \gamma_n|A\beta$$

então reescreve-se

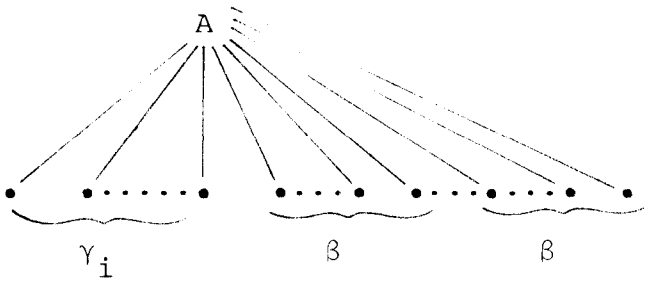
$$A::= (\gamma_1|\gamma_2|\ldots|\gamma_n)\{\beta\}$$

Esta notação indica que se o rótulo da folha corrente é A, então o algoritmo deve reconhecer uma parte inicial da cadeia de entrada α derivável de algum dos γ_i, e depois deve procurar zero ou mais ocorrências de cadeias deriváveis de β.

Para que não haja retrocessos, adota-se a convenção de procurar o número máximo possível de ocorrências de cadeias deriváveis de β.

Para que esta convenção produza resultados corretos, deve-se ter a certeza de que numa forma sentencial, o não terminal A nunca pode ser seguido de uma cadeia derivável de β.

É bom observar a forma que terá a árvore de derivação com esta nova notação:



Esta árvore não indica mais a associação à esquerda original, nem as derivações diretas $A \Longrightarrow A\beta$ que estariam explícitas na árvore de derivação, e o compilador deverá dar uma interpretação conveniente a este tipo de árvore.

Por exemplo, considere a gramática de expressões:

E:: = E+T|E-T|+T|-T|T

T:: = T×F|T÷F|F

F:: = a|b|(E)

Reescrevendo, de acordo com a nova notação introduzida, tem-se

E:: = (+T|-T|T){(+T|-T)}

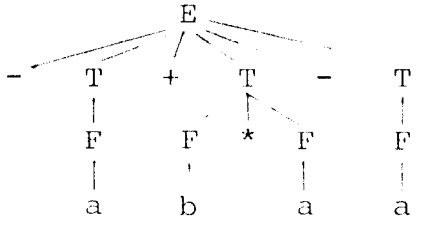
T:: = F{(×F|÷F)}

F:: = a|b|(E)

Dada a cadeia de entrada

α = -a+b*a-a

ao final da análise tem-se a árvore:



o que não corresponde exatamente à uma árvore de derivação e o compilador é que deverá encarregar-se de associar corretamente os operadores +, -, * e ÷ .

3.1.2 - Implementação sem retrocesso, com pilha explícita

Implementar análise descendente sem retrocesso é bastante simples. Para facilitar a exposição vai-se considerar uma gramática do tipo LL(1), em que não foi necessário o uso de notação estendida. Estudando-se o método geral descrito na seção anterior, vê-se que em cada passo é suficiente conhecer a cadeia dos rótulos das folhas da árvore corrente D, começando pela folha corrente.

Esta cadeia pode então ser mantida em uma pilha, onde o elemento do topo será o rótulo da folha corrente. Quando o topo da pilha contiver um terminal, ele será comparado com o símbolo corrente da cadeia de entrada, e se estes símbolos forem iguais, serão ambos removidos. Caso o símbolo do topo da pilha seja um não terminal, ele será substituído na pilha pela cadeia de símbolos correspondente à alternativa escolhida. O primeiro símbolo desta cadeia deverá ficar no topo da pilha.

A escolha da alternativa será baseada nos conjuntos ψ definidos anteriormente, como também pode ser guiada (no caso de uma gramática LL(1)) por uma tabela de análise, levando-nos ao método, descrito na seção 3.1.4.

A seguir apresenta-se a forma geral do procedimento desse tipo de implementação.

```
procedimento ANALISE ;
    inicio
        pilha [1]: = 'símbolo principal da gramática' ;
        Topo: = 1 ; fim: = falso ;
        PROXIMO (símbolo);
        repita
            X: = pilha [topo];
            se "X é terminal"
            então se X = símbolo
                        então { PROXIMO (símbolo);
                                Topo: = topo - 1;
                                se topo = 0
                                então fim: = verdadeiro
                  senão ERRO
            senão se "existe uma produção da forma
                        X:: = X1X2...Xn com símbolo ∈ ψ(x1)"
                        então { Pilha [topo]: = xn ;
                                Pilha [topo+1]: = xn-1 ;
                                  :
                                Pilha [topo+n-1] = x1
                                topo: = topo+n-1
                  senão se "existe uma produção da forma X:: = λ"
                        então topo: = topo-1
                        senão ERRO
```

```
        até que fim = verdadeiro;
        se topo = 0 e simbolo ≠ 'delimitador de final da
                              cadeia de entrada'
        então ERRO
fim ; (*ANALISE*)
```

onde

PROXIMO = procedimento que pega o próximo símbolo da cadeia de entrada

ERRO = procedimento que efetua o tratamento do erro

Observação: estes dois procedimentos serão usados em todas as seções subsequentes.

3.1.3 - Implementação Recursiva

Uma outra maneira simples e comumente usada de se implementar a análise descendente é através de um conjunto de procedimentos mutuamente recursivos. Cada procedimento corresponde a um não terminal da gramática, e a sua função é analisar a parte inicial da cadeia de entrada corrente α que pode ser derivada a partir desse não terminal.

Tem-se então para cada ocorrência de um não terminal do lado direito de uma produção a chamada do procedimento correspondente. Para cada terminal, verifica-se se o mesmo casa com o primeiro símbolo da cadeia de entrada corrente. Em caso afirmativo, avança-se na cadeia de entrada, caso contrário a cadeia dada não é uma sentença.

Para exemplificar este tipo de implementação considera-se a gramática:

S:: = Aa|b

A:: = bdB|e B

B:: = c|ad

A seguir tem-se o analisador descendente recursivo para esta gramática.

```
procedimento RECURSIVO ;
        procedimento S ;
            inicio
            se símbolo ≠ 'b'
                então { A;
                        se símbolo = 'a'
                        então PROXIMO (símbolo)
                        senão ERRO
                senão PROXIMO (símbolo)
                fim ; (*S*)
procedimento A ;
        inicio
            se símbolo = 'b'
                então { PROXIMO (símbolo) ;
                        se símbolo = 'd'
                        então { PRÓXIMO (símbolo) ;
                                B
                        senão ERRO
                senão se símbolo = 'e'
                        então { PRÓXIMO (símbolo)
                                B
                        senão ERRO
```

```
                    fim ; (*A*)
procedimento B ;
    inicio
        se símbolo = 'a'
                ⎧PRÓXIMO (símbolo) ;
        então   ⎨se símbolo = 'd'
                ⎪então PRÓXIMO (símbolo)
                ⎩senão ERRO
    senão se símbolo = 'c'
              então PROXIMO (símbolo)
              senão ERRO
    fim ; (*B*)
        (*programa principal*)
    inicio
        PROXIMO (símbolo) ; S ;
        se símbolo ≠ 'delimitador de final da cadeia de entrada'
        então ERRO
    fim ; (*recursivo*)
```

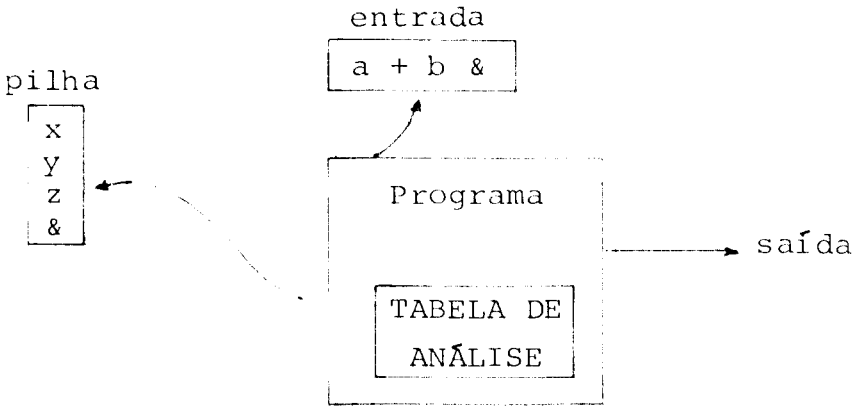
Uma grande vantagem desse tipo de analisador sintático é a sua flexibilidade. Muitas vezes é conveniente que o compilador execute certas ações em pontos intermediários que não correspondem à aplicação de produções. Isto pode ser conseguido fácilmente inserindo-se comandos apropriados em pontos correspondentes dos procedimentos recursivos. A implementação recursiva pode ser bastante eficiente se usada num sistema onde as chamadas de procedimentos causam pouca sobrecarga. Na prática os analisadores descendentes recursivos são bastante uti

lizados, conjugados algumas vezes com outros métodos.

3.1.4 - Implementação orientada por tabela, com pilha explícita

Para evitar a necessidade de uma linguagem recursiva, considera-se também os métodos com pilha explícita (vide 3.1.2), onde a pilha é montada pelo analisador, ao invés de o ser pelo sistema de execução da linguagem na qual o analisador é implementado.

O analisador orientado por tabela é um modo eficiente de implementar análise descendente. O esquema de um analisador tabular pode ser dado pela figura a seguir.



O analisador orientado por tabela tem uma entrada, uma tabela de análise e uma saída. A entrada contém a cadeia a ser analisada seguida por um demarcador de final (&). A pilha contém uma sequência de símbolos da gramática precedida por '&', marca de base da pilha. Inicialmente a pilha contém o símbolo inicial da gramática precedido por '&' . A tabela de análise é uma matriz bidimensional com entradas de forma M[A,a], onde A é um não terminal, e a é um símbolo terminal ou '&'.

O analisador é controlado por um programa que funciona da seguinte maneira. O programa determina X, o símbolo do topo da pilha, e a o símbolo corrente de entrada. Estes dois símbolos determinam a ação do analisador.

Existem três possibilidades:

1 - Se X = a = '&', então termina a análise com sucesso.

2 - Se X = a # '&' o analisador retira X da pilha e avança na entrada para o próximo símbolo.

3 - Se X é um não terminal, o programa consulta a entrada M[X,a] da tabela de análise M. Esta entrada pode ser tanto uma produção de X como uma entrada de erro. Se, por exemplo, $M[X,a] = [X: = x_1x_2...x_n]$, o analisador substitui X no topo da pilha por $x_1x_2...x_n$ (com x_1 no topo). Se M[X,a] = erro, o analisador transfere contrôle para uma rotina de recuperação de erro.

O esquema do procedimento que faz a análise orientada por Tabela pode ser visto a seguir:

```
procedimento TABULAR ;
    inicio
        pilha[1]: = '&' ; pilha[2]: = 'S' ;
        Topo: = 2 ; PROXIMO (símbolo) ;
        repita
           x: = pilha [topo] ;
           se (x é terminal)
           então se x = '&'
                 então se x = símbolo
```

```
                então topo: = topo-1
                senão ERRO
          senão se x = símbolo
                então { topo: = topo-1 ;
                        PROXIMO (símbolo)
                senão ERRO
    senão se M[x, símbolo] = (x:: = x1 x2 ... xk)
                então { pilha [topo]: = Xk ;
                        pilha [topo-1]: = Xk-1 ;
                          :
                        pilha [topo+K-1]: = X1 ;
                        topo: = topo+K-1
                senão ERRO
    até que topo = 0
fim ; (*TABULAR*).
```

Em todos os exemplos a seguir vai-se condiderar a gramática:

$E :: = TE'$

$E' :: = +\ TE' \mid \lambda$

$T :: = FT'$

$T' :: = *\ FT' \mid \lambda$

$F :: = (E) \mid a$

A tabela de análise desta gramática pode ser vista na figura a seguir.

	a	+	*	(	)	&
E	E:: = TE'			E:: = TE'		
E'		E':: = + TE'			E':: = λ	E':: = λ
T	T:: = FT'			T:: = FT'		
T'		T':: = λ	T':: = * FT'		T':: = λ	T':: = λ
F	F:: = a			F:: = (E)		

As entradas em branco são as entradas de erro. Mesmo ainda, sem ter sido dito como construir esta tabela pode-se ver que com a entrada a + a * a o analisador orientado por tabela fará os movimentos descritos a seguir:

pilha	entrada
& E	a + a * a &
& E' T	a + a * a &
& E' T' F	a + a * a &
& E' T' a	a + a * a &
& E' T'	+ a * a &
& E'	+ a * a &
& E' T +	+ a * a &
& E' T	a * a &
& E' T' F	a * a &
& E' T' a	a * a &
& E' T'	* a &
& E' T' F *	* a &
& E' T' F	a &
& E' T' a	a &

& E' T'	&
& E'	&
&	&

3.1.4.1 - Funções PRI e SUC

Define-se agora como preencher as entradas da tabela de análise.

Necessitam-se duas funções associadas com a gramática G. Estas funções irão determinar como preencher as entradas de uma tabela para G, se tal tabela existir.

A primeira delas denominada Primeiro, que denota-se por PRI já foi parcialmente definida na seção 3.1.1 através da função ψ.

Se α é uma cadeia de símbolos da gramática PRI(α) é o conjunto de terminais que iniciam cadeias deriváveis a partir de α.

Se $\alpha \Longrightarrow^{*} \lambda$ então λ também está em PRI(α)

Denota-se a função Sucessor por SUC. Define-se SUC(A) para um não terminal A, como sendo o conjunto de terminais a que podem aparecer imediatamente à direita de A em alguma forma sentencial, ou seja,

$S \Longrightarrow^{*} \alpha A a \beta$ para algum α e β. Se A é o símbolo mais à direita de alguma forma sentencial, então λ está em SUC(A).

Antes de se ver como calcular PRI(α) para qualquer cadeia α, vai-se definir como calcular o conjunto PRI(X) para todos os símbolos da gramática. Para tal, aplica-se as seguintes regras até que nenhum terminal ou λ possa ser adicionado a qualquer conjunto PRI.

1 - Se X é um terminal, então PRI(x) é {x}

2 - Se X é um não terminal e X::= aα é uma produção, então adicione a a PRI(X). Se X::= λ é uma produção então adicione λ a PRI(X).

3 - Se $X::= y_1y_2\ldots y_k$ é uma produção, então para todo i tal que todos $y_1\ldots y_{i-1}$ são não terminais e PRI(y_j) contém λ para j = 1,2,...i-1 (isto é, $y_1y_2\ldots y_{i-1} \Longrightarrow^* \lambda$), adicione todos os símbolos diferentes de λ em PRI(y_i) a PRI(X). Se λ está em PRI(y_j) para todo j = 1,2,...k, então adicione λ a PRI(X). É bom observar que, em particular pode-se ter i=1.

Agora pode-se calcular PRI para qualquer cadeia $x_1x_2\ldots x_n$ como segue:

Adicione a PRI($x_1x_2\ldots x_n$) todos os símbolos não nulos de PRI(x_1). Também adicione todos os símbolos não nulos de PRI(x_2) caso λ esteja em PRI(x_1), e todos os símbolos não nulos de PRI(x_3) caso λ esteja em PRI(x_1) e em PRI(x_2), e assim sucessivamente.

Finalmente adicione λ a PRI($x_1x_2\ldots x_n$), se para todo i, i=1...n, PRI(x_i) contém λ.

Para calcular SUC(A) para todos os não terminais A, aplica-se as regras seguintes até que nada mais possa ser adicionado ao conjunto SUC.

1 - λ está em SUC(S) se S é o símbolo principal de G

2 - Se existe a produção A::= α B β, então tudo que está em PRI(β), menos λ, está em SUC(B)

3 - Se existe a produção A::= αB, ou a produção A::= α B β onde PRI(β) contém λ, então tudo que está em SUC(A) está em SUC(B).

Para exemplificar, vamos considerar a gramática exemplo, e teremos:

PRI(E) = PRI(T) = PRI(F) = {(, a}

PRI(E') = {+ , λ}

PRI(T') = {* , λ}

SUC(E) = SUC(E') = {) , λ}

SUC(T) = SUC(T') = {+ ,) , λ}

SUC(F) = {+ , * ,) , λ}

3.1.4.2 - Construção da tabela de análise

A idéia do algoritmo de construção é simples. Suponha que A::= α é uma produção com a em PRI(α). Então quando o analisador tiver A no topo da pilha e a como corrente símbolo de entrada, pode-se expandir A para α. A única complicação ocorre quando $\alpha = \lambda$ ou $\alpha \stackrel{*}{\Rightarrow} \lambda$ Neste caso pode-se também expandir A para λ, se o corrente símbolo de entrada está em SUC(A), ou se o demarcador '&' for encontrado e λ está em SUC(A).

O algoritmo de construção pode ser descrito por:

1 - Para cada produção A::= α da gramática faça os passos 2 e 3.

2 - Para cada terminal a em PRI(α), acrescente A::= α a M[A,a].

3 - Se λ está em PRI(α), acrescente A::= α a M[A,b] para cada terminal b em SUC(A). Se λ está em PRI(α) e em SUC(α) adicione A::= α a M[A,&]

4 - Faça cada entrada indefinida ser uma entrada de erro.

É bom observar que sómente para gramáticas do tipo LL(1), tem-se a tabela corretamente montada, sem entradas múltiplas.

3.2 - Métodos de recuperação de erros

Nesta seção são apresentados mais detalhadamente os métodos existentes para recuperação de erro, específicos para analisadores descendentes.

Alguns dos métodos foram por nós implementados em um analisador sintático, com pilha explícita sem retrocesso, para a linguagem Pascal. Os resultados dessas implementações são discutidos no capítulo quatro.

De uma maneira geral os métodos de recuperação giram em torno de um método bastante simples e grosseiro denominado método de pânico (em inglês, "panic-mode") que foi usado independentemente por várias pessoas. Neste esquema quando um erro é detetado, avança-se na entrada até que um de uma classe especial de símbolos (geralmente os símbolos considerados delimitadores de comando), tais como o ';' ou o 'end' seja encontrado.

Então, se estiver sendo utilizada uma implementação com pilha de análise explícita retira-se elementos da pilha até um símbolo que torne o atual símbolo de entrada válido. Para isso poderia ser utilizado um procedimento da seguinte forma:

```
procedimento  PÂNICO
        inicio
          S: = "conjunto de delimitadores de comando" ;
          (* símbolo = corrente símbolo de entrada *)
```

```
repita
    enquanto símbolo ∉ S
        faça PROXIMO (símbolo);
    recupera: = falso ; topo1: = topo ;
    repita
        se pilha [topo1] é terminal
        então se pilha [topo1] = símbolo
                então recupera: = verdadeiro
                senão topo1: = topo1 - 1
        senão se símbolo ∈ PRI(pilha [topo1])
                então recupera: = verdadeiro
                senão topo1: = topo1 - 1
    até que (recupera) ou (topo1 = 0)
    se não recupera
    então PROXIMO (símbolo)
    senão topo: = topo1
  até que (simbolo = delimitador de cadeia) ou (recupera)
fim ; (*PANICO*)
```

Vejamos alguns resultados, da implementação, feita por nós, deste método.

Como linguagem analisada foi o Pascal, o conjunto S é composto por: end, until , ';' , else e o símbolo delimitador de cadeia.

Um dos maiores problemas deste método, é que na busca de um delimitador muitos símbolos de cadeia de entrada são descartados e consequentemente muitos erros deixam de ser detetados.

Vê-se isso no exemplo a seguir:

```
program P ;
var ℓ, a ail ,top, ac, 1: array [1..k) of integer ;
                ↑ esperado ';'*              *
   begin
      x: = 1
   end.
```

Ao ser detetado o primeiro erro (indicado por '↑') a cadeia de entrada é descartada até o ';' deixando de ser detetados os outros dois erros existentes (indicados por '*').

Um outro problema verificado, é que devido a função dupla do ';' no Pascal (delimitador de comando e de declaração), tem-se as vezes uma recuperação prematura, ou melhor, no procedimento de recuperação a pilha é acertada em um ponto anterior ao ponto real de recuperação. Vejamos um exemplo disto:

```
program P;
    function F(var x: array [1.. max] of integer): integer ;
                              ↑esperado identificador
        Var q: integer ;
          begin
                ↑esperado var ou identificador
          x: = 1
        end.
```

Devido ao erro ocorrido na declaração dos parametros, a entrada foi descartada até o próximo ';' e a pilha é acertada de maneira a continuar a compilar a lista de parâmetros. É encontrada a declaração da variável q e o compilador continua esperando declarações de parâme

tros e daí a deteção de um erro ao encontrar o begin.

Tem-se a seguir mais alguns, dos exemplos processados:

```
program P ;
    var nodat, norels, noreturned: integer
    function Topsort (C: order ; var sort: sorted ; integer);
            ↑esperado ;                ↑esperado begin
        begin
            x: = 1
        end;
              ↑esperado .
    begin
        x: = 1
    end.
```

Neste caso ao não ser encontrado um ';' esperado o texto é descartado até encontrar o primeiro ';' , sendo portanto descartada a declaração de função e então ao ser encontrado o ';' da declaração de parâmetros ele é encarado como o delimitador da parte de declarações do programa sendo então detetado um erro espúrio, pois o próximo símbolo esperado é o begin que inicia o corpo do programa principal. Daí o texto vai ser descartado até o end sendo este entendido como fim do corpo do programa e em consequência disto é detetado o outro erro espúrio, pois após o end é então esperado o '.' .

```
program P;
   var I: real ;
   const A[1] 10 ; A[2] 15 ; A[3] 25 ; A[4] 3 ; ? A[5] 50 ;
        ↑esperado begin
```

```
begin
   x: = 1
end.
```

Note que no exemplo acima a recuperação somente é feita no end deixando de ser detetados diversos erros.

```
program P ;
     const A[1] = 10 ; A[2] = 15 ; A[3] = 25 ; A[4] = = 35 ;
           ↑esperado = ↑esperado = ↑esperado = ↑esperado =
       var q: integer ;
         begin
            x: = 1
         end.
```

Analisando nossos resultados concluí-se que o grande problema do método de pânico é o descarte de grandes partes do texto de entrada, deixando de detetar vários erros e introduzindo outros tantos. Será fácil perceber que em quase todos os métodos descritos nas próximas seções, a preocupação básica é tentar minimizar este problema.

Um exemplo real de implementação deste método é o utilizado no compilador Algol B6700, onde toda vez que é encontrado um erro, um procedimento da seguinte forma é chamado:

```
procedimento ERRO(n) ;
   início
   (* escreve mensagem de erro n *)
   repita
       se símbolo ='begin'
       então COMANDO
```

```
                senão PROXIMO (símbolo)
            até que (símbolo = 'end') ou
                    (símbolo = ';')
        fim ; (*ERRO*)
```

Nota-se que para adicionar uma melhoria ao método e evitar que partes significativas do texto de entrada sejam descartadas toda vez que é encontrado um 'begin' o contrôle é transferido para o procedimento que compila <comando> (dado que a implementação é recursiva) independente de se estar em condição de erro. Após a compilação de <comando>, retorna-se ao procedimento de recuperação, o qual sòmente terminará ao encontrar um delimitador ('end', ';'). Como a implementação é recursiva não é feito o acerto na pilha de análise, e isto está implícito na transferência de controle entre os procedimento.

Se estivéssemos utilizando uma implementação com pilha explícita, bastaria ao invés de chamar um procedimento COMANDO, colocar na pilha o lado direito da produção que descreve <comando>, juntamente com alguma indicação de quese está em recuperação, e na saída do procedimento de recuperação fazer o acerto na pilha.

Foram processados alguns exemplos utilizando este compilador e os resultados foram satisfatórios, mas deve ser levado em conta a estrutura de blocos da linguagem Algol. Vejamos um dos exemplos processados:

```
integer procedure T ;
   begin
   integer x , t , k ; boolean b ;
     begin
```

```
        b: = true ;
        if b begin
                    ↑Then expected
             x: = t + k
             t: = t + 2 ;
              ↑semicolon expected
             t: = t - -
                        ↑arithmetic primary cannot start
                         with This
             end ;
        Then k: = x + t ;
            ↑no statement con start with This
        k: = k + begin
                        ↑arithmetic primary cannot start
                         with This
                  n: x + t ;
                  y: = y - S
                  end ;
        k: = k + 1
    end
end ;
```

3.2.1 - Método de Turner

Em [Turner, 1977] está descrito um método de recuperação onde supõe-se que o erro encontrado é de primeira ordem, ou seja, in-

serção errônea, omissão ou erro de transcrição de um símbolo simples e que erros diferentes destes ocorrem muito raramente.

Segundo o autor, a vantagem do método é que ele requer muito pouco código extra (cerca de doze linhas) e é muito geral, não envolvendo nenhum conhecimento especial (na forma de tabelas, por exemplo) da particular gramática analisada.

Para maior clareza do método, vamos exemplificá-lo supondo a implementação recursiva do analisador.

Será considerado o seguinte trecho de gramática como exemplo:

```
<comando repetitivo> ::= while <expressão> do
                         <comando> |
                         repeat <comando> { ; <comando>}
                         until <expressão>
```

O método de análise que estamos supondo prevê um procedimento para cada não terminal da gramática.

Incluindo diagnóstico de erro, segundo Turner, teríamos para <comando repetitivo> o seguinte procedimento:

```
procedimento COMREP ;
    inicio
      se TEM ('while")
                 EXPRESSÃO ;
      então      VERIFICA ('do') ;
                 COMANDO
      senão se TEM ('repeat')
                 COMANDO
            então enquanto TEM (';') faça COMANDO ;
                  VERIFICA ('until')
                  EXPRESSÃO
```

```
            senão MENSAG ('comrep')
fim ; (*COMREP*)
```

Os procedimentos VERIFICA, MENSAG e TEM são definidos da seguinte maneira (observe que por enquanto, sòmente está se fazendo o diagnóstico do erro):

```
procedimento TEM (s: cadeia) : booleano ;
   inicio
       se símbolo = s
       então { PROXIMO (símbolo) ;
             { TEM: = verdadeiro
       senão TEM: = falso.
fim ;

procedimento VERIFICA (s: cadeia) ;
   inicio
       se símbolo = s
       então PROXIMO (símbolo)
       senão MENSAG (s)
fim ;

procedimento MENSAG (s: cadeia) ;
   inicio
        erro: = verdadeiro ;
        'escreve mensagem de erro' ;
fim ;
```

O procedimento PRÓXIMO (símbolo) pega o próximo símbolo da cadeia de entrada colocando-o na variável símbolo ; a variável erro inibe a exe

cução do programa compilado.

Note-se que ao ser detetado um erro, provavelmente haverá uma sequência de mensagens de erro espúrias, devido ao fato do compilador poder ter tomado um caminho errado em seu progresso através do texto de entrada.

Isto é altamente indesejável, pois irá interferir na deteção de erros reais, além de confundir o usuário.

Para resolver este problema, Turner introduz uma nova variável denominada recupera que funciona como uma chave, sendo ligada toda vez que o procedimento MENSAG é chamado. A recuperação do erro é efetuada transferindo-se o controle para o próximo ponto no compilador onde seja requerido um determinado símbolo, e então o texto de entrada é descartado até encontrar este símbolo. Para fazer isto, quando ocorre um erro retorna-se o controle ao compilador como se tudo estivesse bem, e espera-se pela próxima chamada do procedimento VERIFICA.

As seguintes modificações são necessárias para que este esquema de recuperação funcione:

```
procedimento VERIFICA (s: cadeia) ;
   inicio
      se recupera
            ⎧enquanto símbolo < > s faça PROXIMO (símbolo) ;
      então ⎨PROXIMO (símbolo) ;
            ⎩recupera: = falso
      senão se símbolo = s
            então PROXIMO (símbolo)
            senão MENSAG (s)
fim ; (*VERIFICA*)
```

```
procedimento MENSAG (S: cadeia) ;
    inicio
        se não recupera
                 ⎧erro: = verdadeiro ;
        então    ⎨recupera: = verdadeiro
                 ⎩escreve mensagem de erro
    fim ; (*MENSAG*).
```

Podemos ver que a recuperação não será eficiente se dois símbolos vizinhos forem incorretos e neste caso pode-se ter uma larga porção do texto de entrada descartada e provavelmente a recuperação será feita em um ponto errado podendo introduzir erros. No pior dos casos quando um símbolo esperado não se encontrar na entrada, todo texto restante será pulado, mas isto não deve acontecer com frequência visto que o autor considera a maioria dos erros como sendo de primeira ordem e portanto sempre se encontrará um determinado símbolo procurado. Pode-se ver um dos exemplos de nossos testes onde tem-se grande parte do texto descartado.

```
program P ;
     var i: real ;
     const  A[1]  10 ; A[2]  15 ; A[3]  25 ; A[4]  3 ;
                 †esperado begin
                  ? A[5]  50 ; A[6] 75 ;
          begin
              x: = 1
          end.
```

Note que neste caso, não há melhora nenhuma em relação ao método de

pânico, muito embora deva ser considerado que o erro não é de primeira ordem.

Observe o exemplo a seguir.

```
program P ;
    const A[1] = 10 ; A[2] = 15 ; A[3] = 25 ; A[4] = = 35 ;
            esperado =
                esperado ;
          A[5] = 50 ; A[6] = 75 ;
    var q : integer ;
        begin
           x: = 1
        end.
```

Neste caso a recuperação é bem pior que a do método de pânico.

Ao ser detetado o primeiro erro é descartado o texto até o próximo símbolo requerido que no caso é um número. Então a recuperação é feita prematuramente no 1 e então é detetado o segundo erro. Daí o texto é descartado até o próximo símbolo requerido que no caso é o begin.

Observando-se exemplos desse tipo, conclui-se que o método poderia ser melhorado se também em cada busca a um símbolo fossem considerados os símbolos delimitadores. Se isto fosse feito, teríamos , no exemplo anterior todas as declarações de constantes compiladas e muitos outros erros detetados.

Este tipo de alteração, de acôrdo com as indiossincrasias da linguagem tornariam o método mais poderoso o que de certa maneira é feito no método de WIRTH descrito na próxima seção.

Em nossos testes foi verificado com frequência a inserção de erros, do tipo do exemplo a seguir:

```
program P ;
    begin
      if x: = 0 then  X: = 1
                 ↑          ↑esperado :
                 |esperado then
    end.
```

Depois de reconhecido o if é aceita a variável x como expressão e a seguir é esperado o then. Como é encontrado o ':=' é detetado um erro, sendo agora procurado um comando.

Como um número pode iniciar um comando com rótulo, a recuperação é feita no 0, sendo então detetado um erro no then pois o próximo símbolo esperado é o ':' .

Portanto, mesmo em um erro de primeira ordem tem-se a inserção de erros espúrios.

Vejamos mais alguns exemplos de nossos testes:

```
program P ;
      funcition topsort (var x: order , var y: sorted ,
                                        ↑esperado)        ↑
                         x: integer) ;                    |
          begin                                  esperado ;
             x: = 1
          end ;
      begin
         x: = 1
      end.
```

Observe que neste exemplo, embora não seja perceptível à primeira vis

ta, ocorre um outro problema do método. Suponha-se que ao procurar um símbolo, seja encontrada uma outra subestrutura que também contém este símbolo, neste caso teremos a recuperação feita prematuramente, levando a se ter texto de entrada descartado ou introdução de erros. No exemplo anterior depois de detetado o primeiro erro o símbolo procurado foi o ':' e portanto sorted é considerado como tipo da função e daí a deteção do erro seguinte, pois então era esperado o ';' .

O mesmo problema é verificado no exemplo a seguir:

```
program FACTORIAL ;
   var ffact, fst ; x, m: integer ;
                    ↑esperado:
                     |esperado ;
     begin
        x: = 1
     end.
```

Neste caso ao ser detetado o erro o próximo símbolo esperado é um identificador que daria o tipo das variáveis ffact e fst. Então a recupeção é feita no x e daí a introdução do erro seguinte. Observe que este também é o caso de um erro de primeira ordem.

Vejamos mais dois exemplos:

```
program P ;
    var ℓ, a ail , top, ac, l : array [l...k) of integer ;
               ↑↑              ↑esperado identificador
               esperado :       |esperado]
                |
                |esperado ;
      begin
        x: = 1
      end.
```

Aqui tem-se exatamente o mesmo problema citado no exemplo anterior.

```
program P ;
   function F(var x: array [1..max] of integer): integer ;
                           ↑esperado identificador
      var Q: integer ;
      begin
         x: = 1
      end.
          ↑esperado ';'
```

Neste caso o método funciona satistatóriamente. Ao ser detetado o primeiro erro o texto é descartado até o próximo símbolo requerido que no caso é o ')' e a recuperação é feita corretamente. Pode-se verificar que mesmo com erros de primeira ordem o método nem sempre efetua a recuperação satisfatóriamente. Se o erro não é de primeira ordem, o que ocorre com frequência, o método torna-se bem ineficiente.

3.2.2 - Método de Wirth

Wirth [1976] sugere um método bastante simples para tratar de erros sintáticos quando utilizam-se analisadores descendentes recursivos. Wirth postula duas regras, as quais ele acha que devem ser levadas en conta ao se planejar o método de recuperação de erro:

1ª se após o diagnóstico de um erro alguma parte do texto de entrada tenha que ser descartada, deve ser levado em conta que quase todas as linguagens de programação possuem algumas palavras chave sem uso dúbio e que portanto não devem ser descartadas, pois servem

para continuar a análise em um passo determinado. No caso do Pascal, seriam palavras tais como o begin, if, while, procedure, etc..

2ª deve-se, quando necessário, descartar o texto de entrada depois da deteção de um erro até um próximo símbolo que possa seguir corretamente a construção sentencial analisada. Isto implica, no caso da implementação recursiva, que todo procedimento do analisador precisa conhecer no momento de sua ativação, o conjunto de símbolos que o podem seguir.

A partir daí, em um primeiro passo, Wirth provê todos os procedimentos de análise de um parâmetro explícito que especificará os possíveis símbolcs que podem seguir a construção sintática analisada por este procedimento. E na entrada de cada procedimento um teste explícito é incluído que verificará se o próximo símbolo da cadeia de entrada pertence ao conjunto de símbolos que iniciam a construção analisada e se não pertencer símbolos da entrada são descartados até encontrar um que pertença. Mas isto, as vezes, pode ser muito desastroso, ou seja, o conjunto de símbolos pode ser muito restrito o que implicaria em grande perda do texto de entrada (o que acontece com o método de Turner da seção anterior).

Wirth sugere, então, adicionar a este conjunto palavras chave que especificam o início de uma construção sintática, e que portanto não deve deixar de ser identificada e analisada. Portanto, os símbolos passados como parâmetro aos procedimentos são agora símbolos de parada e não somente símbolos sucessores de uma determinada construção sentencial. Isto se assemelha à idéia do método de recuperação utilizado no Algol B6700 onde durante a recuperação de erro é iden

tificada e analisada a estrutura básica do Algol que é o bloco.
O conjunto de símbolos de parada, no caso do Pascal, seria iniciado por identificadores de declaração (var, const, type, procedure, function, label) e por begin, e seria gradualmente aumentado por símbolos que seguem construções, à medida que se vai penetrando na hierarquia da definição da linguagem.
Vamos então ver, de uma maneira geral, o funcionamento do método, dada em [Kowoltowski, 1979].

Suponhamos que a rotina P do compilador (correspondente ao não terminal <p> da gramática) chama a rotina Q (correspondente ao não terminal <q>). Seja s1 o conjunto de símbolos terminais que podem aparecer no início de uma cadeia derivado de <q> e seja s2 o conjunto de símbolos terminais que podem seguir uma cadeia derivada de <q>, numa cadeia derivada de <p> .
Para maior flexibilidade do esquema é utilizado um procedimento denominado TESTE. Este procedimento terá três parâmetros s1, s2 e n, onde n identifica a mensagem de erro a ser emitida. Tem a seguinte forma:

```
procedimento TESTE (s1, s2 : símbolos ; n: mensagem) ;
   início
     se símbolo ∉ s1
                 ⎧escreve mensagem de erro n ;
     então       ⎨s1: = s1 U s2.
                 ⎩enquanto símbolo ∉ s1 faça PROXIMO (símbolo)
fim ; (*TESTE*)
```

A rotina P ao chamar a rotina Q, deve passar-lhe como parâmetro, o

conjunto s2 aumentado com os símbolos que foram passados à própria P (conjunto de símbolos de parada). A rotina Q deve começar pela chamada de TESTE (s1, s2, n). Se após o retorno de TESTE o próximo símbolo pertence a s1, então Q prossegue a compilação. Caso contrário, o controle volta a rotina P, que deve prosseguir se o próximo símbolo pertencer a s2, ou então pode voltar por sua vez à rotina que a chamou.

Pode-se ver então, como seria o procedimento que analisa a seguinte construção: <expressão> :: = <expr. simples>{ <relop><expr. simples>}

<relop> :: = = |<|>|<>|<=|>=|in

```
procedimento EXPRESSÃO (s: símbolos) ;
   início
      s1: = 'símbolos que iniciam <expr. simples>' ;
      TESTE (s , s1, k) ;
      se símbolo ∈ s1
                 { s2: = símbolos de <relop>
                 { EXPRSIMPLES (s U s2) ;
      então      { enquanto símbolo ∈ s2
                 {    faça { PROXIMO (símbolo)
                 {         { EXPRSIMPLES (s U s2)
   fim ; (*EXPRESSÃO*)
```

O esquema apresentado tem a propriedade de tentar recuperar, retrocedendo dentro do passo da análise, descartando um ou mais símbolos do texto de entrada. Esta estratégia não é muito feliz, quando o erro é a omissão de um símbolo. Mas estes tipos de erro, como já mencionamos em seções anteriores, estão praticamente restritos à símbolos

que possuem função puramente sintática (como o ';') e que portanto não representam ações.

Vejamos alguns exemplos de nossos testes onde temos omissões de símbolos.

```
program P ;
    var ℓ , a ail , top , ac , 1 : array [1...k) of integer ;
                  ↑ ↑             ↑                 ↑
                  esperado :      esperado identificador
                    |                               |
                  esperado ;                        esperado]
        begin x: = 1 end.

program P ;
    begin
        for i: = 1 to max  y|i| : = 0 ;
                           ↑
                           esperado do
        x: = 1
    end.

program P ;
    var n, m, i integer ; fx, sx, ℓx  real ;
                  ↑                   ↑
                  esperado :          esperado :
        begin
           x: = 1
        end.
```

Pode-se ver que em se tratando de omissão de símbolo com função somente sintática o método funciona satisfatóriamente. Vejamos outro exemplo de omissão:

```
program P ;
   Matrix (name: char , L : boolean , P : integer) ; boolean ;
         ↑esperado begin
     begin
        x: = 1
     end.
```

Neste exemplo, vê-se o quanto é desastroso o esquecimento de um símbolo, que implicaria em uma ação (no caso o begin ou procedure). Tendo-se um grande descarte do texto de entrada. No caso, ao ser detetado o erro o texto é descartado até o begin.

Um outro ponto a considerar-se é que devido ao fato do conjunto de símbolos sucessores ir sendo aumentado por certas palavras chave, pode levar a ter-se uma recuperação prematura, e o analisador pode se comportar como se um símbolo esquecido (ou esperado) tivesse sido encontrado. Isso as vezes é bom que seja feito (como pode-se ver em exemplos anteriores), mas às vezes é muito desastroso. Vejamos um exemplo:

```
program P ;
    function F(var x : array [1..max] of integer) : integer ;
                              ↑esperado ↑identificador ↑esperado ;
                                        ↑esperado )
        var q: integer ;
          begin x: = 1 end.
                           ↑esperado ;
```

Observa-se que depois da deteção do primeiro erro, a recuperação é feita ao se encontrar a variável max, supondo então que este era o

identificador esperado e daí a inserção dos dois erros subsequentes. É a isto que denominamos recuperação prematura.

Tem-se a seguir mais alguns exemplos tirados dos testes feitos em nossa implementação:

```
program P ;
    begin
       if x: = 0 then x: = 1
              ↑esperado Then
                    ↑esperado :
    end.
```

Observa-se que este mesmo problema ocorre com o método de Turner (vide seção anterior)

```
program FACTORIAL ;
    var ffact , fst ; x, m : integer ;
                          ↑esperado :
       begin x: = 1 end.
```

Neste caso a recuperação é feita no próprio ';' .

```
program P ;
     const n = 10 ;
     const mx = 20 *
          ↑esperado ;
     var q: integer ;
     begin x: = 1 end.
```

Neste caso a recuperação é feita no var deixando de serem detetados dois erros: o segundo const e a falta de ';'(indicada por *)

```
program P ;
    begin
       for k1: = to  noelems do x: = 1
                   ↑esperado expressão
    end.
```

Neste caso a recuperação é feita no ponto correto ou seja no próprio to.

```
program P ;
    begin
        Label 1 ; read (x) ; x: = 1
              ↑  ↑
              ↑comando esperado
                 ↑: esperado
    end.
```

Mais um exemplo de recuperação prematura que no caso é feita no 1 considerando-o como rótulo de comando.

```
program P ;
   begin ;
     procedure FACTR(n: integer ; var factor ; integer) ;
               ↑esperado end
        begin x: = 1 end ;
                       ↑esperado .
   x: = 1
   end.
```

Aqui depois da deteção do primeiro erro é descartado todo o texto até o end e daí a deteção do segundo erro, pois este end é considerado como o final do programa.

Amman [1977] utiliza-se deste método de recuperação de erro. Nessa implementação o conjunto de símbolos de parada não é determinado da maneira sistemática que Wirth propõe. A escolha de alguns símbolos provávelmente é baseada também na experiência do uso da linguagem e das características da mesma, visando uma melhor recuperação. A menos dessas pequenas alterações, o funcionamento básico da implementação de Amman é o proposto por Wirth.

Vanini e Burnier [1978] descrevem a implementação de um gerador de compiladores com tratamento automático de erros. O compilador gerado é descendente com pilha explícita sem retrocesso, e o mecanismo de tratamento e recuperação de erro é essencialmente o mesmo descrito por Wirth, só que adaptado a este tipo de implementação do analisador. Quando ocorre algum erro sintático continua-se a análise ignorando-se símbolos na cadeia de entrada até encontrar algum símbolo que possa iniciar uma das derivações "esperando na pilha". O procedimento básico para recuperação de erros é o seguinte:

```
procedimento ERRO ;
        início
          s: = ∅ ;
          para todo símbolo t na pilha
            faça s: = s U PRI(t) ;
          enquanto símbolo ∉ s
             faça PROXIMO (símbolo).
          t: = 'conteúdo do topo da pilha' ;
          enquanto símbolo ∉ PRI (t)
                  faça  {topo: = topo - 1
                         t: = conteúdo do topo da pilha
        fim ; (*ERRO*)
```

O gerador apresentado foi usado na construção de um compilador para um subconjunto da linguagem Pascal, orientado para o ensino. O desempenho no tratamento de erros foi considerado satisfatório, comparado com [Amman, 1977].

Este tipo de implementação foi a utilizada por nós, e é, portanto, de onde foram extraídos os exemplos citados no decorrer dessa seção.

3.2.3 - Método de Hartmann e Pemberton

Hartmann [1977] apresenta um método de recuperação de erro, para analisadores descendentes recursivos que é ditado pelos grafos sintáticos.

Pemberton [1980] mostra algumas falhas e omissões no trabalho de Hartmann, dando a versão corrigida.

As estruturas sintáticas básicas de uma linguagem geralmente se iniciam por um particular conjunto de símbolos denominados, por Hartmann, alavancas (em inglês, 'handles').

Por exemplo, o conjunto de alavancas de um comando em Pascal é:

{identificador, begin, if, case, while, repeat, for , with}

Quando um erro é detetado, zero ou mais símbolos são descartados até se obter um símbolo para o qual a compilação possa continuar.

Este símbolo geralmente não é único, pertencendo a um conjunto denominado por Hartmann de conjunto de símbolos chave. O conjunto de símbolos chave é fornecido para uma rotina de erro, juntamente com um número que identifica a mensagem de erro:

procedimento ERRO (n: inteiro ; chaves símbolos) ;

```
inicio
    escreve mensagem de erro n ;
    enquanto símbolo ∉ chaves
        faça PRÓXIMO (símbolo)
fim ; (*ERRO*)
```

Note que esta é a idéia do método de Wirth apresentado na seção anterior, e a diferança básica entre os dois métodos está no maneira de determinar o conjunto de símbolos chave.

Para Hartmann, o conjunto chaves irá conter qualquer operador que possa ocorrer no grafo correntemente analisado e poderá também conter alavancas de qualquer outro grafo, do qual o grafo corrente seja um subgrafo. Como na método de Wirth, este esquema implica em que todo procedimento de análise aceita como entrada as chaves de seu "Chamante". Chaves locais são adicionadas às iniciais quando um dado procedimento chama outro. Mas Hartmann não faz menção à um conjunto de símbolos iniciais de parada como Wirth o faz. De uma maneira geral as chaves aumentam a medida em que procedimentos são chamados e diminuem quando as chamados são completadas. O conjunto de chaves conterá símbolos de cada nível ativo na hierarquia sintática.

Hartmann deriva o conjunto de símbolos chave diretamente dos grafos síntáticos. Existe um procedimento que é chamado em toda bifurcação do grafo sintático:

```
procedimento VERIFICA (n: inteiro ; chaves: símbolos) ;
    inicio
        se símbolo ∉ chaves
        então ERRO (n, chaves)
    fim ; (*VERIFICA*)
```

Hartmann apresenta seu método baseado na idéia de que grafos sintáticos podem ser decompostos em três elementos básicos:

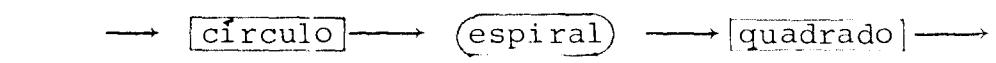
1 - sequência

2 - seleção

3 - laço

Para exemplificar, vamos considerar duas construções sintáticas abstratas círculo e quadrado e um terminal abstrato espiral vê-se então a seguir como deve ser feito a recuperação em cada um dos três, elementos do grafo sintático:

① sequência




```
procedimento SEQUENCIA (chaves: símbolos) ;
    início
        CÍRCULO (chaves U {espiral}U{alavancas de quadrado }) ;
        se símbolo ∈ {espiral}
        então PROXIMO (símbolo)
        senão ERRO (erro de sequência, chaves U {alavancas de quadrado})
        QUADRADO (chaves)
fim ; (*SEQUENCIA*)
```

Note que ao ser chamado este procedimento, chaves irá conter todos os símbolos que podem seguir esta construção no grafo no qual ela é um subgrafo.

Para poder-se comparar, vejamos como seria escrito este procedimento segundo o método de Wirth.

```
procedimento SEQUENCIA (s: símbolos) ;
    inicio
        s1: = 'símbolos que iniciam círculo' ;
        TESTE (s1, s,k) ;
        se símbolo ∈ s1
        então { CIRCULO (s ∪ {espiral})
                se símbolo ∈ {espiral}
                então { PROXIMO (símbolo)
                        s1: = 'símbolos que iniciam quadrado' ;
                        TESTE (s1, s,L) ;
                        se símbolo ∈ s1
                        então QUADRADO (S)
    fim ; (*SEQUENCIA*)
```

Pode-se notar uma diferença básica entre os dois métodos: a análise da sequência no caso de Wirth seria imediatamente interrompida caso não fosse identificada alguma das construções e no método de Hartmann a análise prossegue como se todas as construções estivessem presentes. Portanto, supondo-se que se tem a seguinte construção para ser analisada:

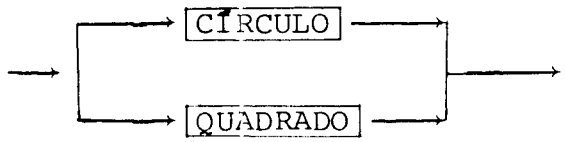
S:: = Ab

A:: = Ca

Pelo método de Wirth ao ser chamado o procedimento A, a partir de S, o conjunto de símbolos de parada é acrescido de b e caso não seja encontrado algum símbolo que inicie C é feito o retorno ao procedimento S sem tentar encontrar um a como é feito no método de Hartmann. Isto faz com que tenhamos menos descarte do texto de entrada na ocorrência de um erro, mas por outro lado pode levar mais fácilmente à

se ter uma recuperação prematura do tipo da mencionada na seção anterior.

② seleção




```
procedimento SELEÇÃO (chaves: símbolos) ;
   início
     VERIFICA (erro de seleção , chaves U {alavancas de círculo}
                                        U{alavancas de quadrado}) ;
     se símbolo ∈ {alavancas de círculo}
     então CÍRCULO (chaves)
     senão  se símbolo ∈ {alavancas de quadrado}
            então QUADRADO (chaves)
            senão ERRO (erro de seleção, chaves)
   fim ; (*SELEÇÃO*)
```

Pemberton aponta os seguintes problemas neste procedimento.
Pode-se ter, nesta análise, duas mensagens de erro (uma no procedimento VERIFICA e outra no próprio SELEÇÃO) motivadas pelo mesmo erro, o que não é desejável. Quando se identifica um erro sintático em VERIFICA é dado uma mensagem de erro, e então pula-se o texto de entrada até um símbolo, o qual talvez não inicie nenhum dos ramos da seleção, e então quando se retorna do procedimento VERIFICA tem-se imediatamente outra mensagem de erro, essencialmente causada pelo mesmo erro sintático.

Isto pode ser evitado redefinindo-se o procedimento VERIFICA, tendo-se os símbolos procurados nos ramos da seleção e os símbolos de recuperação como conjuntos diferentes tem-se então:

```
procedimento VERIFICA (esperados: símbolos ; n: inteiro ;
                       chaves: símbolos) ;
    inicio
       se símbolo ∉ esperados
       então ERRO (n, esperados U chaves)
    fim ; (*VERIFICA*)
```

e o procedimento SELEÇÃO será:

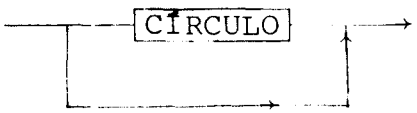
```
procedimento SELEÇÃO (chaves: símbolos) ;
    inicio
       VERIFICA ({alavancas de círculo} U {alavancas de quadrado},
                 erro-seleção, chaves) ;
       se símbolo ∈ {alavancas de círculo}
       então CIRCULO (chaves)
       senão { se símbolo ∈ {alavancas de quadrado}
               então QUADRADO (chaves)
    fim ; (*SELEÇÃO*)
```

Segundo o método de WIRTH este procedimento seria escrito da seguinte forma:

```
procedimento SELEÇÃO (s: símbolos) ;
    inicio
       s1: = 'conjunto de símbolos que iniciam quadrado' ;
       s2: = 'conjunto de símbolos que iniciam círculo' ;
       TESTE (s1 U s2, s, k) ;
```

```
    se símbolo ∈ s1
    então QUADRADO (s)
    senão { se símbolo ∈ s2
            então CIRCULO (s)
fim ; (*SELEÇÃO*)
```

Não existe portanto neste caso diferença alguma entre os dois métodos. Existe um caso que Hartmann não cobre quanho estuda seleção que é o de uma das alternativas ser vazia:



Segundo Hartmann o procedimento para este grafo seria:

```
procedimento OPÇÃO (chaves: símbolos) ;
    inicio
       VERIFICA (erro-opção, {alavancas de círculo} U chaves) ;
       se símbolo ∈ {alavancas de círculo}
       então CIRCULO (chaves)
    fim ; (*OPÇÃO*)
```

A versão revisada segundo Pemberton será:

```
procedimento OPÇÃO (chaves: símbolos) ;
    inicio
       VERIFICA ({alavancas de círculo} U chaves, erro-opção, [ ]) ;
       se símbolo ∈ {alavancas de círculo}
       então CIRCULO (chaves)
    fim ; (*OPÇÃO*)
```

O terceiro parâmetro de VERIFICA é vazio, pois os símbolos do contex

to são incluídos nos símbolos esperados devido à opção vazia

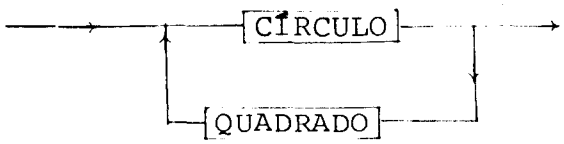
(3) laço


```
procedimento LAÇO (chaves: símbolos) ;
    variável
        chaves-laço, chaves-tudo: símbolos ;
        acabou: booleana ;
    início
        chaves-laço: = {alavancas de círculo} U {espiral} ;
        chaves-tudo: = chaves-laço U chaves ;
        acabou: = falso ;
        repita
          CÍRCULO (chaves-tudo) ;
          VERIFICA (erro-laço, chaves-tudo) ;
          se símbolo ∈ chaves-laço
          então {se símbolo ∈ {espiral}
                 então PRÓXIMO (símbolo)
                 senão ERRO (erro-laço, chaves-tudo)
          senão acabou: = verdadeiro
    até que acabou
fim ; (*LAÇO*)
```

Vejamos agora as observações de Pemberton sobre este procedimento. Nota-se que se o símbolo espiral estiver ausente no programa analisado, uma mensagem de erro é dada e a análise continua como se ele es

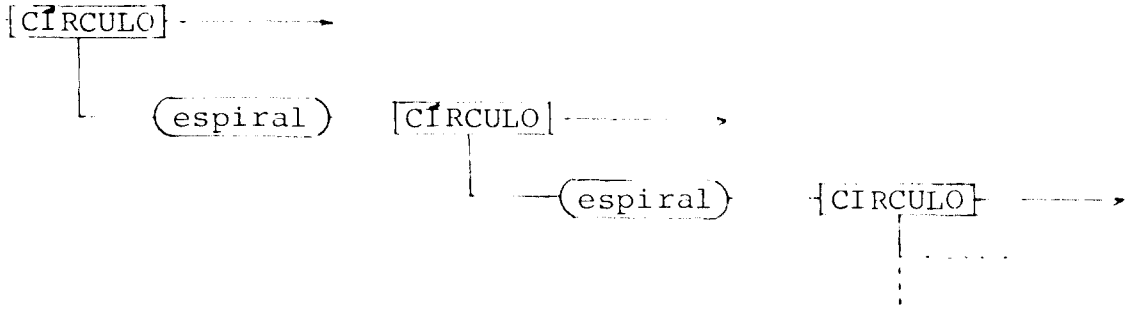
tivesse presente (o mesmo que acontece com a sequência).

Além disso parece que o procedimento é específico para um determinado exemplo e não se aplica para o caso geral, como por exemplo:



Mais importante que isso, o procedimento dado faz a suposição, não necessáriamente correta de que o conjunto de símbolos que podem seguir a construção é disjunto das alavancas de círculo. Em particular se um círculo pode seguir o laço, o procedimento dado falha em programas corretos. Isto porque o conjunto de chaves-laço foi errônea - mente escolhido e isto se torna claro quando verifica-se que:

é uma contração de:



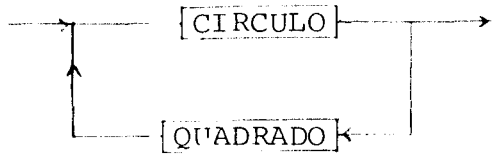
e que em uma determinada bifurcação o conjunto de símbolos a ser verificado é:

{espiral} U chaves

Daí a versão revisada do procedimento para laço:

```
procedimento LAÇO (chaves: símbolos) ;
    variável
        acabou: booleana ;
    início
        acabou: = falso ;
        repita
            CIRCULO ({alavancas de círculo} U {espiral} U chaves) ;
            VERIFICA (chaves U {espiral}, erro-laço, {alavancas de
                                                círculo}) ;
            se símbolo ∉ chaves - {espiral}
            então ⎧se símbolo ∉ {espiral}
                  ⎪então ERRO (erro-laço, chaves U {alavancas de
                  ⎪                                      círculo})
                  ⎪senão PRÓXIMO (símbolo)
                  ⎩
            senão acabou: = verdadeiro
    até que acabou
fim; (*LAÇO*)
```

e para o caso:



Teríamos:

```
procedimen                ves: símbolos) ;
    v
    ini
```

```
        acabou: = falso ;
        repita
           CIRCULO ({alavancas de círculo} U {alavancas de
                        quadrado} U chaves) ;
           VERIFICA ({alavancas de quadrado} U chaves ,
                        erro-laço1 , {alavancas de círculo}) ;
           se símbolo ∉ chaves - {alavancas de quadrado}
           então {se símbolo ∈ {alavancas de quadrado}
                  então QUADRADO (chaves U {alavancas de
                                círculo} U {alavancas de quadrado})
                  senão ERRO (erro-laço1, {alavancas de
                                círculo} U chaves)
           senão acabou: = verdadeiro
    até que acabou
fim ; (*LAÇO1*)
```

Pelo método de Wirth teriamos:

```
procedimento LAÇO1 (S: símbolos) ;
    inicio
      s1: = 'símbolos que iniciam círculo' ;
      TESTE (s1, s, K) ;
      se símbolo ∈ s1
      então {S2 = 'símbolos que iniciam quadrado'
             CÍRCULO (s U s2) ;
             enquanto símbolo ∈ s2
             faça {QUADRADO (s U s1) ;
                   CÍRCULO (s U s2)
fim ; (*LAÇO1*)
```

As diferenças são as mesmas observadas para o caso da sequência. Como a maioria dos métodos, este é bastante símples, podendo ser gerado automáticamente a partir da gramática da linguagem e com a preocupação em fazer a recuperação baseada no contexto global da análise (não se observa isso no método de Turner e no método geral de pânico).

3.2.4 - Método de Irons

Um dos primeiros métodos publicados em recuperação automática de erros, foi o método de Irons descrito em [Irons, 1963]. Vamos ilustrar a técnica de Irons, utilizando a seguinte gramática exemplo:

P:: = A ;

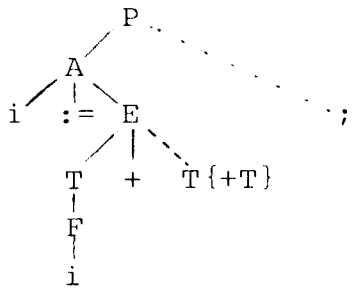
A:: = i: = E

E:: = T{+T}

T:: = F{*F}

F:: = i | (E)

Vamos supor a implementação de um analisador descendente, com pilha explícita e sem retrocesso. Em qualquer ponto da análise, podemos ter a árvore sintática construída com alguns ramos incompletos.
Por exemplo:



onde as linhas pontilhadas mostram como os ramos P e E precisam ser completados.

Um ramo incompleto chamado U corresponde a uma aplicação de regra:

$$U:: = X_1 X_2 \ldots X_{i-1} X_i \ldots X_n$$

onde $X_1 \ldots X_{i-1}$ é a parte completa do ramo e $X_i \ldots X_n$ é a parte incompleta.

Na figura anterior um ramo incompleto de P corresponde à aplicação da regra P:: = A ; , e ';' é a parte incompleta do ramo. A parte incompleta do ramo chamado E corresponde a uma aplicação da regra E:: = T{+T}. Para completar o ramo necessita-se de um único T seguido de qualquer número de '+T' . A parte incompleta á portanto T{+T}.

Um outro exemplo, considere a regra:

$$U:: = (A|BC|DE)(F|G)$$

onde os parênteses são metasímbolos.

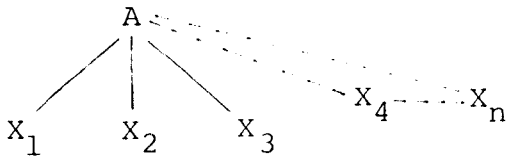
Suponha-se que B é a parte completa do ramo. Então a parte incompleta é C(F|G), desde que tanto CF como CG podem completar a parte direita.

Estas partes incompletas dos ramos exercem um papel fundamental na recuperação de erro ; elas nos dizem, em última instância, o que deve ou pode aparecer no programa fonte.

Agora, suponha-se que um erro ocorre durante a análise sintática e a árvore sintática não pode ser construída. Então a seguinte ação de recuperação é executada.

1 - Uma lista L das partes incompletas dos ramos é construída.

Observa-se aqui que se tivermos



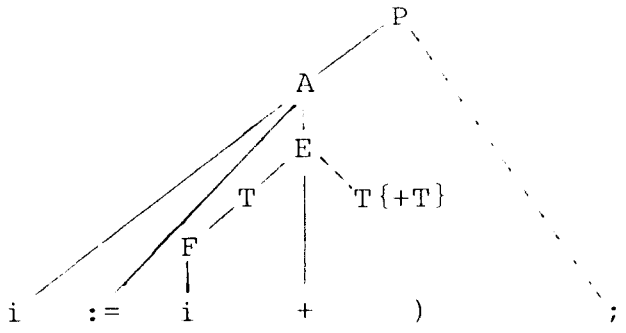
$X_4 \ldots X_n$ entram nesta lista, não sendo importante a ordem em que aparecem.

2 - Os símbolos da cadeia α de entrada são repetidamente examinados e descartados (gerando uma nova cadeia α) até que seja encontrado um símbolo t, tal que $U \Longrightarrow^{*} t \ldots$ para algum $U \in L$ (então ou U=t ou $U \Longrightarrow^{+} t$).

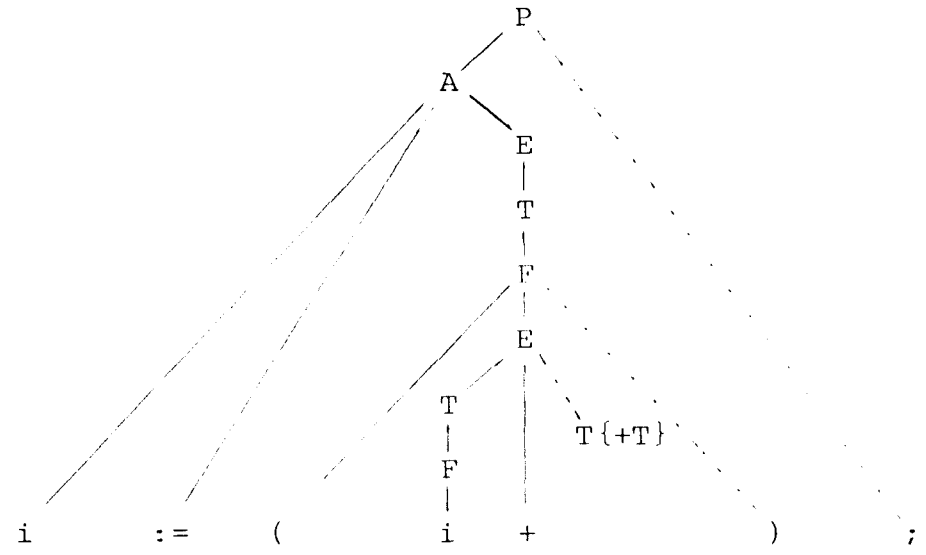
3 - O ramo incompleto o qual causou o símbolo U do passo 2 ser posto em L é determinado.

4 - Uma cadeia terminal β é determinada tal que, se inserida exatamente antes de t, a continuação da análise cause t ser corretamente ligado ao ramo incompleto do passo 3, e a todos os ramos incompletos da subárvore determinada por ele. Portanto para cada ramo incompleto, uma cadeia de terminais é gerada a qual o completa, e estas cadeias são concatenadas de maneira a formar β.

5 - A cadeia β é inserida imediatamente antes de t e a análise continua com o primeiro símbolo de β como próximo símbolo de entrada. Considera-se, por exemplo, a análise indicada pela figura seguinte:



Um erro ocorre quando ')' é o símbolo de entrada. Tem-se então L = {T, + , ;}. O passo 2 da recuperação consiste em descartar o ')' . O ramo incompleto o qual ocasionou o ';' ser posto em L é P:: = A ; Precisa-se então inserir uma cadeia ß para completar o ramo E:: = T{+T}. A cadeia a ser inserida é o identificador i. Para outro exemplo, considere a figura a seguir:



Através deste exemplo pode-se ver como o esquema se utiliza do contexto global ao determinar a recuperação.

O erro nesta figura parece ser o mesmo da figura anterior, ou seja, o '+' seguido de ')' .

Agora entretando L = {; ,) , + , T} e neste caso o ')' não é descartado pelo passo 2. O ramo incompleto o qual causou ')' ser posto em L é F:: = (E). Para que ')' seja associado com este ramo, precisa-se inserir uma cadeia para completar o ramo E:: = T{+T} e isto novamente é o identificador i. Portanto, sempre, todos os ramos incompletos exercem uma função na recuperação de erro e não sòmente os do ponto do erro.

Note-se que a cadeia β a ser construida poderá ser nula, caso o símbolo de entrada no ponto do erro, seja um símbolo espúrio que possa ser apagado. Nenhum símbolo de entrada é descartado caso o erro seja uma omissão, e por consequência o símbolo de entrada original irá seguir um ponto futuro na árvore.

A aplicação deste método geralmente não é inteiramente possível quando se usa a implementação recursiva, pois a árvore sintática nem sempre está disponível.

3.2.5 - Método de Fisher, Milton e Quiring

A estratégia do método descrito em [Fisher, 1977] é a de na ocorrência de um erro, colocar símbolos adicionais na cadeia de entrada de maneira a torná-la correta.

O método é limitado às gramáticas do tipo LL(1) denominados corrigíveis por inserção (em inglês, "insert-correctable"), ou seja gramáticos do tipo LL(1) para os quais sempre é possível efetuar a correção de um erro pela inserção de uma cadeia terminal.

O corretor de erros apresentado tem algumas propriedades desejáveis. Ele requer no máximo tempo e espaço linear, sempre produz um programa sintáticamente correto para qualquer cadeia de entrada e pode ser automáticamente gerado para uma gramática do tipo LL(1) corrigível por inserção, apesar de ser necessário fornecer, "à priori", um custo de inserção para cada símbolo terminal.

Devido ao fato de ser necessário considerar inserções no final da cadeia de entrada, este método se utiliza de gramáticas aumentadas,

definidos da seguintes forma:

Dada G = (T, N, P, S)

a gramática aumentada G' é definida por:

$G' = (T \cup (\&), N \cup \{S'\}, P \cup \{(S' ::= S\&\}, S')$

onde $\& \notin T$ e $S' \notin N$.

$\hat{T}$ irá denotar $T \cup \{\&\}$ e $\hat{N}$, $N \cup \{S'\}$

Para que esse algoritmo tenha sucesso são necessárias duas restrições: uma sobre a classe de gramáticas utilizada e outro sobre o algoritmo de análise.

É claro que nem todos as linguagens do tipo LL(1) são suscetíveis à um corretor somente de inserção. Porexemplo, considere a gramática:

G1 : S' ::= S&

S ::= a | (a)

para a entrada a) ... não existe nenhuma inserção $\beta \in T^*$ tal que a β) ... $\in L(G1)$. Deve-se então definir a classe de gramáticas para a qual se pode usar este tipo de corretor.

"Uma gramática livre de contexto é dita corrigível por inserção, se e sómente se, para todo $\alpha \in T^*$ e $a \in \hat{T}$ tal que:

$S' \Rightarrow^* \alpha \ldots$ e $S' \not\Rightarrow^* \alpha a \ldots$

existe $\beta \in T^*$, tal que:

$S' \Rightarrow^* \alpha \beta a \ldots$. "

Exemplo de gramática corrigível por inserção:

G2 : S' ::= E&

E ::= TE'

E' ::= + TE' | λ

T ::= a | (E)

Também requer-se um algoritmo de análise que nunca faça uma transição sobre um símbolo errado, ou seja, que detete o erro tão logo este símbolo seja encontrado. Esta restrição não é muito forte, pois a maioria dos analisadores descendentes para linguagens do tipo LL possuem esta característica.

A restrição de que a linguagem tenha que ser corrigível por inserção é mais forte, pois na maioria dos casos onde a transformação é possível (o que acontece, por exemplo com o Algol) ela acarreta um grande número de alterações o que não é desejável.

O algoritmo de correção de erros utiliza-se de duas tabelas auxiliares denominados S e E. A construção destas tabelas é baseada em uma função de custo C definida da seguinte maneira:

$C(a)$, $a \in T$ é fornecido "à priori".

$C(\lambda) = 0$

$C(\beta) = C(a_1) + \ldots + C(a_n)$ para $\beta = a_1 a_2 \ldots a_n$ e $\beta \in T^*$

A tabela S irá identificar a cadeia terminal de menor custo derivável a partir de X, e é definida por:

"Para $x \in \hat{N} \cup \hat{T}$, $S(x) = \beta$ onde

$\beta \in \hat{T}^*$, $x \Longrightarrow^* \beta$ e para todo $\gamma \in \hat{T}^*$ tal que

$x \Longrightarrow^* \gamma$, $C(\beta) \leq C(\gamma)$. Se $x \in \hat{T}$, $S(x) = x$" .

Esta definição não implica em que S(x) seja único. A partir da definição anterior, pode-se estender a função de custo C para os símbolos não terminais, e tem-se:

$$C(A) = C(S(A))$$

O algoritmo a seguir, calcula a tabela S, para todo $A \in \hat{N}$. Simultaneamente também calcula C(A), iterando sobre a gramática, até que o cus

to convirja para o mínimo

```
procedimento CONSTRÓI-S ;
    início
        para todo A ∈ N̂ faça { C(A) := ∞ ;
                               S(A) := λ ;
        repita
            mudou := falso ;
            para todo p = (A::= x1 ... xn)
            faça se C(x1 ... xn) < C(A)
                 então { C(A) := C(x1 ... xn) ;
                         S(A) := S(x1). S(x2)...S(xn) ;
                         mudou := verdadeiro
    até que não mudou
fim ; (*CONSTRÓI-S*)
```

Veremos agora como definir a tabela E.

Se $A \in \hat{N}$ estiver no topo da pilha de análise e a for o símbolo de entrada que ocasiona um erro, então E(A,a) irá identificar uma expansão de A, a qual irá levar a inserção de menor custo de uma cadeia terminal. Caso no topo da pilha se tenha um terminal b diferente de a, então insere-se S(b).

Portanto a tabela E é definida da seguinte forma:

"Para $A \in \hat{N}$ e $a \in \hat{T}$, E(A,a) = (0,0) se $A \not\Rightarrow^{*} \ldots a \ldots$, caso contrário E(A,a) = (p,i) onde $p = (A::= x_1 \ldots x_i \ldots x_n)$ e $x_i \Rightarrow^{*} \ldots a \ldots$"

Adicionalmente (p,i) tem a propriedade do menor custo:

"Seja β a cadeia terminal de menor custo, tal que $x_i \Rightarrow^{*} \beta\ a \ldots$, en

tão para todo (q,j) onde q = $(A::= y_1 \ldots y_j \ldots y_n)$ e $y_j \overset{*}{\Longrightarrow} \gamma$ a ...,tem-se $C(x_1 \ldots x_{i-1}\beta) \leq C(y_1 \ldots y_{j-1}\gamma)$"

Define-se também k(A,a) = C, onde $C = C(x_1 \ldots x_{i-1}\beta)$ dado E(A,a) = (p,i). Para todo a,b $\in \hat{T}$, se a = b então K(a,b) = 0, caso contrário k(a,b) = ∞

Tem-se então o algoritmo que calcula E:

```
procedimento CONSTRÓI-E ;
    início
        para todo A ∈ N faça
            para todo a ∈ T̂ faça { E(A,a) = (0,0)
                                 { K(A,a) = ∞
        repita
            mudou: = falso ;
            para todo a ∈ T̂ faça
                para todo p = (A::= x1...xn)
                    faça { L: = MIN (C(x1...x(i-1)) + K(xi,a))
                               1≤i≤n
                         (*seja um j tal que L = C(x1...x(j-1)) +
                                                 + K(xj,a)*)
                         se L < K(A,a)
                         então { E(A,a) = (p,j)
                               { k(A,a): = L ;
                               { mudou: = verdadeiro
        até que não mudou
fim ; (*CONSTRÓI-E*)
```

Estas tabelas, dependendo do tamanho da gramática ($|\hat{N}| \times |\hat{T}|$) podem ser muito grandes, devendo ser armazenadas em um dispositivo de memória auxi

liar, desde que, como vê-se a seguir, para cada correção sòmente é necessário pesquisar uma coluna das tabelas.

O funcionamento do corretor de erros é dado pelo procedimento a seguir.

Seja a pilha de análise $x_n \ldots x_1$ & e o símbolo errado a o próximo símbolo da cadeia de entrada.

```
procedimento CORRETOR ;
    inicio
        para j: = n até 1 faça
          se x_j ≠ a então
             se (x_j ∈ T̂ ou (E(x_j,a) = (0,0) e K(x_j,a) = ∞)
             então INSERE (S(x_j))
             senão {PREFIXO (x_j,a) ;
                    J: = 0
    fim ; (*CORRETOR*)
```

O procedimento PREFIXO (A,a) calcula a cadeia β de menor custo ($\beta \in T^*$) tal que dado

$A \in \hat{N} \cup \hat{T}$, $a \in \hat{T}$ e $A \Longrightarrow^* \ldots a \ldots$, então $A \Longrightarrow^* \beta\ a \ldots$

```
procedimento PREFIXO (A,a) ;
    inicio
       se A ≠ a
       então {(*seja E(A,a) = (p,i) e K(A,a) = C onde p=(A::= x_1...x_i
                                                          ...x_n)*)
              INSERE (S(x_1) ... S(x_i-1)) ;
              PREFIXO (x_i,a)
    fim ; (*PREFIXO*)
```

O procedimento INSERE (w) insere a cadeia terminal w imediatamente à esquerda do símbolo a que ocasionou o erro.

Portanto, o procedimento de correção percorre a pilha de análise até encontrar um símbolo, o qual, possa derivar o símbolo errado, e insere $S(x_i)$ para todos os símbolos x_i verificados. Quando é encontrado um símbolo x_j tal que $x_j \stackrel{*}{\Longrightarrow} \ldots a \ldots$, se $x_j \neq a$, então é chamado o procedimento PREFIXO que calcula o restante dos inserções requeridas. O controle é então devolvido ao analisador, tendo como próximo símbolo de entrada o símbolo inserido mais à esquerda.

Note-se que a eficiência do corretor descrito, está presa à informação do custo de inserção, que não é formalmente definida.

Supõe-se que o ideal é fornecer um vetor de custo com valôres iguais para todos os símbolos terminais, e depois, de acôrdo com o desempenho do algoritmo ir alterando-o de maneira conveniente.

Também é bom ressaltar que nada do texto de entrada é descartado. Observa-se também que a cadeia β inserida antes de um símbolo incorreto a é a de menor custo, mas não é única e portanto pode não ser a melhor opção no contexto corrente, podendo ocasionar futuros erros espúrios.

Provávelmente, este fato pode ser contornado com uma boa definição do vetor de custos, mas que dificilmente poderá ser formalizada pois varia de acôrdo com a linguagem compilada e seu uso.

3.2.6 - Método de Ghezzi

Ghezzi [1976] propõe um algoritmo de recuperação de erros

que pode ser gerado automáticamente à partir de descrição formal da linguagem. A filosofia geral do método é a de que quando um erro é detetado pelo analisador sintático, é tentada uma ação de reparo local e, se falhar, uma ação de recuperação é executado.

Para facilitar a explanação do método, vamos dar algumas definições usadas pelo autor:

Ele define a configuração de um analisador para uma liguagem do tipo LL(1) como sendo $(\alpha, k, x_j x_{j-1} \ldots x_1)$

1 - $\alpha \in T^*$, é a cadeia de entrada

2 - k é um inteiro entre 1 e n, onde n é o tamanho de α e $\alpha(k)$ é o símbolo corrente de entrada.

3 - $x_j x_{j-1} \ldots x_1$ é o conteúdo da pilha de análise (x_j é o símbolo do topo).

A configuração inicial é $(\alpha, 1, S)$ e a final é (α, n, λ) (S é o símbolo principal da gramática).

Além das funções PRI e SUC definidos em (3.1.4.1), o algoritmo usa as seguintes funções:

a) conjunto diretor para uma produção

p: A::= $\alpha \in P$ é:

$DS(p: A::= \alpha) = \{a \mid \alpha \Longrightarrow^* a\beta$ ou $(\alpha \Longrightarrow^* \lambda$ e $a \in SUC(A)\}$

$DS^2(p: A::= \alpha) = \{ab \mid \alpha \Longrightarrow^* ab\beta$ ou

$(\alpha \Longrightarrow^* a$ e $S \Longrightarrow^* \gamma_1 Ab\gamma_2)$ ou

$(\alpha \Longrightarrow^* \lambda$ é $S \Longrightarrow^* \gamma_1 Aab\gamma_2)\}$

b) O conjunto de entrada compatível com um não terminal $A \in N$ é:

$CS(A) = \{a \mid a \in PRI(A)$ ou $(A \Longrightarrow^* \lambda$ e $a \in SUC(A))\}$

O algoritmo pressupõe que nenhuma modificação vai ser necessária na pilha de análise e que nada do texto de entrada terá que ser reexaminado.

Seja $\alpha = \beta a \lambda$ a cadeia de entrada a ser analisada e a o símbolo de entrada o qual causa a deteção de um erro. A primeira etapa do método tenta efetuar um reparo local, considerando as seguintes três condições de erro:

e1: $\alpha' = \beta b a \gamma$ é uma cadeia correta

(isto é, um símbolo foi omitido em α)

e2: $\alpha' = \beta a' \gamma$ é uma cadeia correta

(ou seja, um símbolo mal escrito)

e3: $\alpha' = \beta \gamma$ é uma cadeia correta

(ou seja, um símbolo extra foi inserido em α)

Para poder-se identificar as condições de erro acima, considera-se que quando um erro é detetado, então um não terminal ou um terminal está no topo da pilha e pode-se ter uma das seguintes configurações:

$$C_1 = (y_1 y_2 \cdots y_{i-1} \; a \; y_{i-1} \cdots y_n, \; i \; , \; b x_{j-1} \cdots x_1)$$

ou

$$C_2 = (y_1 y_2 \cdots y_{i-1} \; a \; y_{i-1} \cdots y_n, \; i \; , \; B x_{j-1} \cdots x_1)$$

Seja $B:: = \beta_1 | \beta_2 | \ldots | \beta_k$ o conjunto de produções que reescrevem B de acordo com G ;

define-se então

$$SS(p_i = B:: = \beta i) = \{c | ac \in DS^2(p_i : B:: = \beta i)\}$$

Define-se também o conjunto corretor de uma produção $p_i = (B:: = \beta_i)$ como:

$$CRS(p_i: B::=\beta_i) = \{c \mid c \in SS(p_i: B::=\beta_i) \ \underline{e}$$
$$c \notin SS(p_j: B::=\beta_j \text{ , para todo } j \neq i\}$$

Voltando-se ao método de recuperação, teremos então que se C_1 é a configuração corrente os erros e_1, e_2 e e_3 podem ser respectivamente detetados se as seguintes condições permanecem:

C_1e_1: $X_{j-1}\cdots X_1 \stackrel{*}{\Longrightarrow}$ a $y_{i+1}\cdots y_n$

C_1e_2: $X_{j-1}\cdots X_1 \stackrel{*}{\Longrightarrow} y_{i+1}\cdots y_n$

C_1e_3: $y_{i+1} = b$

No caso da configuração corrente ser C_2 temos:

C_2e_1: existe a produção p: B::= β

Tal que a $\in$ CRS(p: B:: = β)

C_2e_2: existe a produção p: B::= β tal que

$y_{i+1} \in$ CRS(p: B:: = β)

C_2e_3: $y_{i+1} \in$ CS(B)

como pode-se observar as condições de erro são identificados diretamente da inspeção das regras da gramática que descreve a linguagem compilada.

Nota-se também que as condições de erro não são mutuamente exclusivas e portanto, precisam ser verificados em uma determinada ordem, que pode ser alterada de acordo com a frequência em que ocorrem.

A ordem escolhida pelo autor e utilizada em nossa implementação foi e_1, e_2 e e_3. Quando nenhuma ação de correção é identificada, um procedimento de recuperação é chamado. O objetivo deste procedimento é reiniciar a análise, descartando a menor porção possível do texto de entrada, e seu funcionamento pode ser descrito por: considerando o

configuração do analisador como sendo $(\alpha, i, x_j x_{j-1} \cdots x_1)$

$\alpha = y_1 y_2 \cdots y_{i-1}\ a\ y_{i+1} \cdots y_n$

a é o corrente símbolo de entrada

e

$x_j x_{j-1} \cdots x_1$ é o conteúdo da pilha de análise

Temos:

```
procedimento RECUPERA ;
    inicio
        para q: = i até n faça
         para h: = j até 1 faça
          se (Xh é um terminal e Xh = yq) ou
             (Xh é um não terminal e yq ∈ CS(Xn))
          então {(y1y2...yq...yn, q, xh...x1) é a corrente configura
                                                              ção ;
                 finalize o procedimento de recuperação
    fim ; (*RECUPERA*)
```

Observa-se que este procedimento não difere do apresentado por Wirth, se olharmos a implementação feita por Vanini e Burnier [1978] pois o que é feito no procedimento acima é ir descartando símbolos da entrada até encontrar um que seja compatível com alguma das configurações possíveis da pilha de análise, ou seja, com uma das derivações "esperando" na pilha (isto de acordo com a terminologia de Vanini e Burnier).

A utilização deste método poderá muito facilmente ocasionar a inserção de erros espúrios decorrentes dos reparos locais que são efetuados na primeira etapa da recuperação

Isto porque tais reparos não são únicos para um dado erro, podendo portanto ser feito o menos "correto" desde que não existe uma formalização criteriosa de qual reparo deva ser feito. Este tipo de problema geralmente é um constante quando tenta-se qualquer tipo de recuperação que não envolva o contexto global da análise.

Vejamos alguns exemplos coletados dos testes de nossa implementação.

```
program P ;
    procedure PR ;
          begin
             s: = 1
          end
    function getelemente (var X: integer) ; boolean ;
             ↑esperado ;                    ↑esperadc :
          var q: integer ;
           begin
               x: = 1
           end ;
begin
end.
```

Observa-se que neste caso tem-se uma recuperação mais eficiente que no método de Wirth, pois neste, teríamos a deteção de um erro espúrio depois de boolean, pois a recuperação seria feita no ; que ocasionou o erro, presumindo que este fosse o ';' final da declaração da função. Já neste método, boolean encaixa-se na situação $e_1 c_1$, e a recuperação é então efetuada.

```
program FACTORIAL ;
    var ffact , fst ; X, m: integer ;
                      ↑    ↑
                      esperado :
                           |
    begin                  esperado ;
       x: = 1
    end.
```

Já neste exemplo a correção local não foi a mais indicada e ocasionou a inserção de um erro espúrio.

```
program P ;
    var prt : array [1..2*MAX] of inyeger ;
                    ↑     ↑              ↑
                    |     |              |
                    esperado]          esperado :
                          |
                          esperado ;
       begin
          x: = 1
       end.
```

Aqui tem-se uma recuperação prematura. Após o primeiro erro a recuperação é feita no identificador max e então este é considerado como o tipo do "array" declarado e daí a inserção dos outros dois erros.

```
program P ;
    begin
      int[list] ; X: = 1
                   ↑    ↑
                   esperado :=
                        |
                        esperado end
end.
```

Neste exemplo novamente tem-se uma correção local ocasionando um erro espúrio.

```
program P ;
        begin
           for K1: = to noelems do x: = 1
                     ↑expressão esperada
        end.
```

geralmente quando o erro é de omissão como o acima a recuperação é bastante boa sendo efetuada no ponto exato onde deveria ocorrer

```
program P ;
   begin ;
   procedure F (n: integer ; var factor: integer) ;
            ↑esperado end
      begin
         x: = 1
      end ;
           ↑esperado .
   begin
   end.
```

Neste caso após a ocorrência do primeiro erro a recuperação somente é feita no end que finaliza a declaração do procedimento sendo este considerado o end final do programa e daí a inserção do segundo erro.

```
program NUNTER [x,y] ;
               ↑esperado ( ou ;
                    ↑esperado )
    var Q: integer ;
       begin
          x: = 1
       end.
```

3.2.7 - Método de Setzer

Setzer [1981] descreve a implementação de um analizador descendente para gramáticas do tipo ESLL(1), que é uma subclasse das gramáticas LL(1).

Não daremos a definição destas gramáticas, pois o algoritmo de uma maneira geral, não é restrito a este tipo específico de gramáticas.

Considere-se a cadeia de entrada

$$\beta = e_1 e_2 \cdots e_j e_{j+1} \cdots e_n$$

e suponha-se que houve uma deteção de erro no símbolo e_j.

A correçãc sintática deste erro será feita através de uma de quatro estratégias descritas a seguir.

Como veremos, este método é análogo ao de Ghezzi, diferindo sómente no critério de aplicação das estratégias de correção e na maneira de verificá-las visto que Setzer representa a gramática sob a forma de listas e Ghezzi sob a forma de tabelas.

Vejamos então a proposta de Setzer:

Considera-se

$$\alpha = \alpha_1 \alpha_2 \cdots \alpha_i$$

como conteúdo da pilha de análise, sendo α_i o símbolo do topo.

a) Estratégia E: eliminação de um símbolo

Nesta estratégia é suposto que o usuário cometeu o engano de escrever a cadeia de entrada com um símbolo a mais. Isto é, e_j é supérfulo e deve ser eliminado. Para testar-se se esta hipótese pode ser adotada, basta verificar se $e_{j+1} \in PRI(\alpha_i)$

b) Estratégia I: inserção de um símbolo

Nesta estratégia supõe-se que o usuário cometeu o engano de omitir um símbolo que deve ser inserido. Para testar se esta hipótese pode ser adotada basta supor-se que entre e_{j-1} e e_j deveria haver um símbolo que é idêntico a algum símbolo de $PRI(\alpha_i)$ e para tal verifica-se se e_j pertence ao conjunto de símbolos sucessores (SUC) de algum dos símbolos de $PRI(\alpha_i)$

c) Estratégia T: um símbolo trocado

Nesta estratégia supõe-se que o usuário escreveu um símbolo errada-mente. Isto é, um símbolo correto e'_j foi trocado pelo símbolo incorreto e_j. Para testar se esta hipótese pode ser adotada basta aplicarmos a estratégia I, considerando e_{j+1} ao invés de e_j .

d) Estratégia D: busca do delimitador

Para todo não terminal contido na pilha de análise, verifica-se se e_j pertence ao conjunto de símbolos sucessores, ou seja, para todo não terminal α_k em α verifica-se se $e_j \in SUC(\alpha_k)$. Caso e_j pertença a $SUC(\alpha_k)$ então $\alpha = \alpha_1 \cdots \alpha_{k-1}$.

Essas estratégias são testados sequencialmente; se nenhuma conseguir corrigir o erro, ignora-se o símbolo e_j e passa-se ao próximo símbolo e_{j+1} , voltando a testar as quatro estratégias novamente.

Se mais de uma tiver sucesso, usa-se a primeira estratégia da sequência que ignorar o menor número de símbolos de entrada.

Como pode-se ver existe uma preocupação clara em descartar o menor número possível de símbolos de entrada.

Este método de recuperação foi na prática implementado em um analisador sintático para a linguagem Pascal e, comparado os resultados com

compiladores Pascal padrão, pode-se constatar que as mensagens de erro são mais claras (isto em virtude da maneira como foi implementado analisador) e tem-se um maior número de erros detetado. Nada foi observado em relação aos erros espúrios, que geralmente são mais frequentes devido aos acertos locais.

Durante o desenvolvimento do trabalho apresentado em [Setzer, 1981] os autores chegaram à conclusão que a maior parte das vezes um só símbolo está errado em cada trecho do programa fonte e que por meio das estratégia de eliminação, inserção e troca deve-se conseguir tratar a maior parte desses erros.

Por outro lado, é a estratégia do delimitador que garante o fato de não se eliminar um número exagerado de símbolos de entrada e que se for necessário escolher uma das quatro estratégias, recomendam a última.

3.2.8 - Método de Pai

Pai [1978] e, Pai e Kieburtz [1980] descrevem um método de recuperação baseado na suposição de que a maioria dos erros encontrados pelo compilador podem ser colocados em poucas classes bem definidas.

Para Pai, a razão de qualquer esquema de recuperação de erro é que um programa errado sempre está muito próximo de um programa correto. De acordo com isso, quando um erro é detetado, Pai tenta fazer uma alteração local no texto do programa, ao redor do ponto de erro, de maneira a tornar esta porção de texto sintáticamente válida. As alte

rações permissíveis neste caso são usualmente restritas à eliminação, inserção ou troca de um certo número de símbolos. Quando não há sucesso neste tipo de alteração, uma estratégia de recuperação mais drástica é tentada. Descartando-se uma porção de texto de entrada é possível que se possa continuar a análise; isto é chamado por Pai de recuperação de contexto global.

Essa filosofia de recuperação é a mesma dos métodos de Ghezzi e Setzer vistos em seções anteriores.

A ênfase principal de Pai é na recuperação de contexto global, mas também é apresentado uma estratégia de correção local que segundo Pai vem funcionando satisfatóriamente. A recuperação de contexto global é efetuada em dois estágios:

1) a cadeia de entrada é percorrida e são descartados átomos até que um de um conjunto específico de átomos (chamados de símbolos fiduciais) seja encontrado.

2) o conteúdo da pilha de análise é substituído por uma cadeia a partir do qual o analisador possa aceitar o restante do texto de entrada que agora se inicia com o símbolo fiducial encontrado no primeiro estágio.

Torna-se necessário, na etapa de reconfiguração, ter-se a capacidade de inferir a estrutura do texto de entrada que segue o símbolo fiducial; supõe-se que a porção do texto de entrada começando com o símbolo fiducial é um sufixo sentencial válido e com base no conhecimento do símbolo fiducial deduz-se as propriedades adicionais deste sufixo sentencial.

Então a escolha de um símbolo como fiducial depende do quanto ele res

tringe a estrutura do sufixo sentencial iniciado por êle.

Pode existir um número infinito de sufixos sentenciais terminais começando com um dado símbolo, mas se todos eles podem ser derivados a partir de um único sufixo sentencial então diz-se que os sufixos sentenciais começando com este símbolo são completamente restringidos; tais símbolos são chamados símbolos fiduciais fortes.

Existem poucos símbolos fiduciais fortes em qualquer gramática de linguagem de programação comumente usada. Mas podem existir muitos símbolos em uma gramática para uma linguagem de programação os quais restringem completamente os sufixos sentenciais iniciados por eles, mas pertencentes à uma subclasse especial. Pai considera a subclasse de sufixos que derivam formas sentenciais não recursivamente; um símbolo é fiducial fraco se ele inicia um sufixo sentencial nesta subclasse, a partir do qual podem-se derivar todos os outros sufixos sentenciais desta subclasse, que começam com este símbolo.

Pai baseia a recuperação de contexto global nos símbolos fiduciais fracos.

Seja a cadeia $\alpha = \beta xy\gamma$ onde x é um símbolo que ocasiona um erro.

Por correção simples de átomo entende-se substituir x por outro símbolo, eliminar x, ou inserir um símbolo antes de x. Diz-se que um erro é normal se existe uma correção simples de átomo e anormal caso contrário.

Muitas vezes não existe um única correção simples de átomo possivel para um dado erro normal. Esta ambiguidade entre alternativas de correções pode algumas vezes ser resolvida verificando-se um número fixo de símbolos à frente, no texto de entrada.

Existem dois problemas envolvendo este "olhar para frente"

1) A cadeia verificada pode conter outro erro.

2) Não existe um número fixo de símbolos que possa resolver todas as ambiguidades.

Por exemplo, no seguinte segmento de programa Pascal:

.... ; x: = y a [<expressão inteira>] : = ...

é encontrado um erro em 'a' que pode ser corrigido inserindo-se, por exemplo, um '+' ou um ';' antes de 'a'. Mas a [<expressão inteira>] pode ser arbitrariamente longa e esta ambiguidade não pode ser resolvida olhando-se um número limitado de símbolos a frente.

3.2.8.1 - Algumas definições e teoremas

Em [Pai, 1978] tem-se muitas definições e teoremas que auxiliam a compreensão do método.

Definição 1: Pai define o que ele denomina de conjunto futuro (em inglês, 'lookahead set') do par (A,i), LAS(A,i) que é igual ao conjunto diretor de uma produção DS(p:A $\Longrightarrow \alpha$) considerando-se p a i-ésima produção (ver seção 3.2.6).

Em recuperação de erro é útil conhecer se dois dados símbolos terminais podem ocorrer contíguos em uma forma sentencial e também, se um símbolo terminal pode aparecer em uma cadeia derivada a partir de um não terminal.

Estas informações são contidas em duas funções booleanas, CONTIG e DERIV, respectivamente.

Definição 2:

CONTIG: T x T → {V,F} é definida como segue:

CONTIG (r,t) = V, se existem α e β tal que

$$S \overset{*}{\Longrightarrow} \alpha\ r\ t\ \beta$$

= F, caso contrário

Note que a condição necessária para que uma cadeia α r t seja um prefixo valido é que CONTIG (r,t) = V.

Definição 3:

DERIV : N x T → {V,F} é definida como segue:

DERIV (A,t) = V, se existem α e β tal que

$$A \overset{*}{\Longrightarrow} \alpha\ t\ \beta$$

= F, caso contrário.

A função DERIV é usada na recuperação de erro para reajustar a pilha de análise quando as correções locais falham.

Em qualquer estágio da análise certos símbolos são permitidos como símbolos de continuação. Este conjunto de símbolos aceitáveis, (em inglês, 'acceptable symbols') AS, é definido como segue:

Definição 4:

Supõe-se que a pilha de análise contém a cadeia xα, onde x é um símbolo terminal ou não terminal, sendo o conteúdo do topo da pilha.

Então,

$$AS(x) = \{t \mid x \overset{*}{\Longrightarrow} t\ \alpha\ \}$$

$$AS(x\alpha) = AS(x) \text{ se } x \overset{*}{\not\Longrightarrow} \lambda$$

$$= AS(x) \cup AS(\alpha), \text{ caso contrário}$$

$$AS(\alpha) = \emptyset, \text{ se } \alpha = \lambda$$

Na recuperação de erro o átomo produzido pela correção local precisa pertencer ao conjunto de símbolos aceitáveis para o estado (conteúdo da pilha de análise) do analisador no ponto de erro.
Como já foi mencionado, quando as correções locais falham, o analisador léxico avança até um símbolo fiducial e a pilha é ajustada para aceitar o sufixo começando com este símbolo. Se houver mais de uma maneira de fazer este ajuste, então se escolherá a que mais provavelmente conduza ao sucesso. Estas observações podem ser formalizadas, como vê-se a seguir.

Definição 5:

Para quaisquer símbolos A e z em uma gramática G, uma derivação z-imersível (em inglês, 'z-embedding') a partir de A, é uma derivação $A \overset{*}{\underset{e}{\Rightarrow}} \alpha$ tal que z ocorre em α.
Note que α pode conter várias ocorrências de z e para evitar ambiguidade sempre se fará referência à ocorrência mais à esquerda.

Definição 6:

A derivação z-imersível $A \overset{*}{\underset{e}{\Rightarrow}} \alpha$ é dita mínima se nenhuma cadeia intermediária na derivação contém uma ocorrência de z.

Definição 7:

Seja $D = (A \overset{*}{\underset{e}{\Rightarrow}} \alpha)$ uma derivação z-imersível e seja TR a árvore de derivação de D. Em TR, seja PT o caminho a partir da raiz à folha mais à esquerda rotulada z. Se não existirem dois nós em PT com o mesmo rótulo então D é chamada de derivação esquerda z-imersível não recursiva, LRNR, (em inglês, 'z-embedding left-to-right non-recursive derivation').

Intuitivamente uma derivação z-imersível é LRNR se z é derivado não recursivamente.

Teorema 1: Se DERIV (A,t) então existe uma derivação t-imersível LRNR mínima a partir de A.

Definição 8:

Seja $A \overset{*}{\Longrightarrow} \alpha\ z\ \beta$ uma derivação LRNR z-imersivel mínima a partir de A. Então z β é chamado de z-sufixo mínimo de A.

Usam-se derivações LRNR mínimas na reconfiguração da pilha de análise, na recuperação de contexto global.

3.2.8.2 - Recuperação local

Considera-se um analisador sintático sem retrocesso, com pilha explícita, sendo 'a' o símbolo que causou a deteção de um erro.

Passo 0: Pegue o próximo símbolo t da cadeia de entrada.

Passo 1: Calcule os conjuntos S1, S2, e S3 como segue. Seja γ o conteúdo da pilha de análise e seja R = AS(γ)

$S1 = \{x \mid x \in R \text{ e } CONTIG(x,b)\}$

$S2 = \{x \mid x \in R \text{ e } CONTIG(x,a)\}$ se

$CONTIG(a,t)$

$= \emptyset$, caso contrário

$S3 = \{t\} \cap R$

Passo 2a: Se todos os três conjuntos forem vazios, então tem-se um erro anormal; chame a recuperação global e retorne ao analisador ; senão

Passo 2b: Um ou mais dos conjuntos é não vazio. Este erro normal pode ser corrigido de um dos seguintes modos:

1) a pode ser substituído por um x ∈ S1.

2) qualquer x ∈ S2 pode ser inserido antes de a.

3) Se S3 é não vazio então pode-se descartar a.

Se algum destes conjuntos tem um único elemento e os outros são vazios, então a correção é única.

Aplica-se a correção apropriada e retorna-se ao analisador.

Passo 3: O número de correções locais possíveis é igual ao número total de símbolos nos três conjuntos. Salve o conteúdo da pilha de análise e aplique a primeira correção. Retorne o controle ao analisador. Se um erro ocorrer no próximo símbolo da cadeia de entrada, então restaure a pilha e a entrada, e tente a próxima correção aplicável.

Repita este procedimento até que uma correção dê sucesso ou não existam mais alternativas para serem testadas e neste caso chame a recuperação global.

Notou-se na prática que o uso da relação CONTIG corta drásticamente o número de alternativas e o passo3 é então repetido poucas vêzes.

Tem-se mais uma vez uma variação sobre o método de Ghezzi, no que diz respeito à recuperação local. As diferenças observadas são na verificação de situações de erro e no critério de aplicação das correções. Vê-se, assim como no método de Setzer, uma preocupação clara em não se fazer alterações locais que por serem precipitadas venham introduzir erros no texto de entrada.

3.2.8.3 - Recuperação de contexto global

Percorre-se a entrada até encontrar um símbolo fiducial p; depois retiram-se símbolos da pilha de análise até que DERIV (topo da pilha, p), e então substitui-se o elemento do topo por um dos p-sufixos mínimos (se os símbolos fiduciais são corretamente encontrados, o topo da pilha terá um único p-sufixo). Apresentam-se a seguir duas versões, uma não-determinística e a outra determinística, do algorítmo para reconfifurar a pilha.
É bom observar que nos algoritmos seguintes empilha-se uma 'marca-de-produção' antes de expandir um não terminal dentro da pilha, e desempilhar esta 'marca-de-produção' implica que a última produção aplicada falhou como tendo um p-sufixo mínimo dentro da pilha de análise.

Algoritmo 1 - Versão não determinística

```
procedimento RECTOTAL ;
   inicio
     sucesso: = verdadeiro ;
     enquanto (pilha [topo] ≠ p) e sucesso
           se pilha [topo] = 'marca-de-produção'
           então sucesso: = falso
           senão se pilha [topo] é terminal
                 então topo: = topo - 1
                 senão
                  se (não DERIV (pilha [topo], p))
                     ou (pilha [topo] foi préviamente expandida)
```

então topo: = topo - 1

senão { expanda pilha [topo]
depois de colocar
a marca-de-produção
na pilha

fim ; (*RECTOTAL*)

Onde expandir significa substituir o topo da pilha pelo lado direito de uma produção para o não terminal dado.

Se sucesso=verdadeiro na conclusão do algoritmo então substitui-se o elemento do topo da pilha por um p-sufixo mínimo.

No caso extremo a pilha de análise pode estar vazia em qualquer parte do algoritmo. Se isto acontecer empilha-se o símbolo principal e chama-se o procedimento novamente. Desde que o símbolo principal satisfaz a relação DERIV para todo terminal, uma derivação LRNR imersível a partir do símbolo principal existe para qualquer símbolo fiducial. E então, na segunda vez, o procedimento de recuperação deve terminar com sucesso, dando o desejado sufixo na pilha.

Algoritmo 2 - Versão determinística

(* variáveis globais e funções que são usadas *)

Topo

Tiposímbolos = (lista de átomos).

produções = 1 .. (nº de produções da gramática) ;

constantes

as: vetor [tiposímbolos] de um conjunto de tiposímbolos ;

```
    (* conjunto de símbolos terminais aceitáveis para cada não
       terminal *)
variáveis
        t: tiposímbolos ; (*símbolo terminal devolvido pelo analisa-
                           dor léxico*)
        pilha-análise: pilha de tiposímbolos ;
função SP: ↑pilha ; (*apontador do topo da pilha de análise*)
função TOPOPILHA: tiposímbolos ; (*símbolo do topo da pilha de
                                  análise*)
função PROD-IND (nt, t: tiposímbolos): produções ;
       (*determina se alguma sentença pode ser gerada a partir do pri
         meiro argumento, a qual contenha o símbolo terminal dado co-
         mo segundo argumento*)
função DERIV (NT,T: tiposímbolos): booleana ;
procedimento RECUPERAÇÃO-DE-CONTEXTO ;
    constante
         prisímbolo = primeiro símbolo de tiposímbolos ;
    variáveis
         expsímbolos: conjunto de tiposímbolos ;
         sucesso: booleana ;
procedimento TENTA-RECUPERAR ;
    variáveis
           nt,s:tiposímbolos ;
           lp: ↑pilha ;
           Tentasimb: conjunto de tiposímbolos ;
           Tentaprod: produções ;
```

```
inicio
    se 'pilha vazia' então coloque símbolo principal da gramáti-
        ca na pilha ;
    nt: = TOPOPILHA ;
    retire um elemento da pilha ;
    lp: = SP ;
    expsímbolos: = expsímbolos U {nt};
    s: = prisímbolo ;
    tentasimb: = as [nt] ;
    repita
       (* localizar o símbolo em as [nt] *)
       enquanto (s ∉ Tentasimb)
            faça  s: = succ(s) ;
       Tentaprod: = PROD-IND (nt,s) ;
       se 'o lado direito de tentaprod é não vazio'
          então {aplica Tentaprod ; (*coloque seu lado direito
                 na pilha de análise*)
                 repita
                     se t ∈ as [TOPOPILHA]
                     então SUCESSO: = verdadeiro
                     senão se TOPOPILHA é não Terminal e
                           TOPOPILHA ∉ expsímbolos e DERIV
                           (TOPOPILHA,t)
                           então TENTA-RECUPERAR
                           senão retire elemento da pilha de análise
                 até que (sucesso) ou (SP=lp)
```

```
            Tentasimb: = Tentasimb - {s}
        até que (sucesso) ou (Tentasimb = ∅)
fim ; (*TENTA-RECUPERAR*)
inicio (*RECUPERAÇÃO-DE-CONTEXTO*)
   avance o analisador léxico até um símbolo fiducial e atribua-o a
   variavel t ;
   expsímbolos: = ∅ ;
   sucesso: = falso ;
   repita
       TENTA-RECUPERAR
    até que SUCESSO
fim ; (*RECUPERAÇÃO-DE-CONTEXTO*)
```

3.2.8.4 - Caracterização dos símbolos fiduciais

Nessa parte vai-se estudar as propriedades que qualificam um símbolo como sendo fiducial.

Na primeira parte dessa seção mencionou-se duas suposições básicas sobre a recuperação de contexto global. Para recapitular:

1) A porção do texto de entrada seguinte e inclusive o símbolo fiducial (encontrado quando o analisador léxico avança) é um sufixo válido da linguagem.
2) Depois da reconfiguração da pilha de análise, o analisador pode analisar qualquer sufixo válido que venha a ser encontrado.

A suposição 2 dá uma ideal, mas restrita, definição de um símbolo fiducial.

Vejamos agora as propriedades que qualificam um símbolo como fiducial.

Definição 1: A cadeia α é chamada de t-sufixo terminal se:

1) α é uma cadeia de símbolos terminais e,

2) α é um t-sufixo de uma forma sentencial.

Note-se que um t-sufixo terminal tem t como primeiro símbolo.

Definição 2: Um símbolo terminal t é um símbolo fiducial forte (SFS) se existe uma cadeia β tal que β é o sufixo de uma forma sentencial e para qualquer t-sufixo terminal α existe uma derivação $\beta \stackrel{*}{\Longrightarrow} \alpha$.

Definição 3: Um símbolo terminal a em uma gramática G tem um único-nível forte de frase, SPLU, (em inglês, 'strong phrase level uniqueness')

se ou G é vazia, ou

1) existe no máximo uma produção $A \Longrightarrow \alpha$, a qual contém a no seu lado direito.

2) a ocorre apenas uma vez em α.

3) A não tem recursão esquerda ou embutida em G, isto é, não existe derivação em G da forma $A \stackrel{*}{\Longrightarrow} \alpha A \beta$ com $\beta \neq \lambda$ ou $\alpha = \lambda$.

4) A tem SPLU em G(A).

(G(A) é a gramática obtida a partir de G, apagando-se todas as produções para A e tratando A como um símbolo terminal).

Pode-se mostrar que SPLU é uma condição suficiente para um símbolo ser um SFS. Isto é feito através das definições e teoremas a seguir.

Definição 4: Seja a um símbolo terminal com SPLU em G. Então existe um único símbolo A_1 para qual a ocorre no lado direito de uma produção sua. Seja A_2 o único símbolo em $G(A_1)$ tal que A_1 ocorre no lado direito de A_2, e assim por diante. Portanto, tem-se uma única cadeia $\alpha = A_n,\ldots A_1, A_o = a$ com a propriedade que A_i tem SPLU em $G(A_1)(A_2)\ldots(A_i)$. Chama-se esta cadeia de cadeia de dominância (em inglês, 'dominance chain') para a.

Lema 1: Seja A_o, $A_1\ldots A_n$ a cadeia de dominancia para z em G. Então A_o, $A_1\ldots A_i$ são acessíveis a partir do símbolo principal apenas através de A_{i+1} .

(X é acessível a partir de A se $A \stackrel{*}{\Longrightarrow} \alpha\, x\, \beta$)

Lema 2: $G(A_i) = G(A_o)\ldots(A_i)$.

Lema 3: Nenhum dos $A_o, A_1\ldots A_n$ tem recursão esquerda ou embutida em G.

Definição 4: Seja $A_o, A_1, \ldots A_n$ a cadeia de dominancia para o símbolo z. Então tem-se as produções

$$A_{i+1} \Longrightarrow \alpha_i A_i \beta_i \ , \ i = 0,1,\ldots n-1$$

Tal que A_i ocorre apenas uma vez em $\alpha_i A_i \beta_i$. Chama-se $A_i\beta_i$ de A_i-sufixo mínimo de A_{i+1}

Teorema: O A_i-sufixo mínimo de S, isto é, $A_{n-1}\,\beta_{n-1}$, pode derivar todos os A_i-sufixos de S em $G(A_i)$.

Corolário: Se z tem SPLU em G, então z é um SFS.

Portanto, através das definições e teoremas vistos até agora mostra-se que SPLU é uma condição suficiente mas não necessária para um símbolo ser um SFS. Entretanto, parece ao autor que gramáticas nas quais todo símbolo é SFS devem gerar uma classe restrita de linguagens.

Em qualquer gramática para linguagens de programação em uso, existem poucos símbolos qualificados a serem SFS e poucos destes tem SPLU (em Pascal, por exemplo, apenas program tem SPLU).

Entretanto, a escolha de palavras reservadas como símbolos fiduciais conduz à uma recuperação de erro com sucesso, na maioria dos casos práticos. Este fato é atribuído à redundancia das palavras reservadas. Por exemplo, considere-se o seguinte comando Pascal:

if a > b then a: = c else a: = b ,

Suponha-se que se substituem todas as ocorrências das palavras reservadas if, then e else por uma nova palavra gen .

O novo comando seria escrito:

gen a > b gen a: = c gen a: = b

Nos testes feitos por Pai, verificou-se que o sucesso da recuperação de contexto global para uma linguagem é uma proporção direta da redundancia da linguagem. A razão dessa correspondência pode ser vista no exemplo acima.

No comando original apenas uma expressão booleana poderia seguir uma ocorrência do if, e no novo comando tanto uma expressão booleana como um comando podem seguir uma ocorrência de gen.

Desde que na recuperação de contexto global, se está interessado em conhecer, sem ambiguidade, o que pode seguir um símbolo fiducial, if é um bom candidato para símbolo fiducial.

Dá-se a seguir algumas conceituações que caracterizam o que Pai denomina de símbolo fiducial fraco, que são os símbolos utilizados em sua implementação do método.

Definição 5: Seja G = (T,N,P,S) uma gramática livre de contexto. O grafo de dominância (DG) de G é um grafo dirigido rotu

lado definido como segue:

1) DG(G) = (V,E), onde V é o conjunto de nós, e E o conjunto de arcos.

2) V = T U N

3) Arco (x,y) com rótulo i pertence a E se x é o lado esquerdo da i-ésima produção, e y ocorre no lado direito desta produção.

   Se y ocorre mais de uma vez no lado direito, então existem múltiplos arcos (x,y) com rótulo i.

exemplo: Considere uma gramática G com as seguintes produções:

(1) S:: = ABC

(2) A:: = aBC

(3) A:: = a

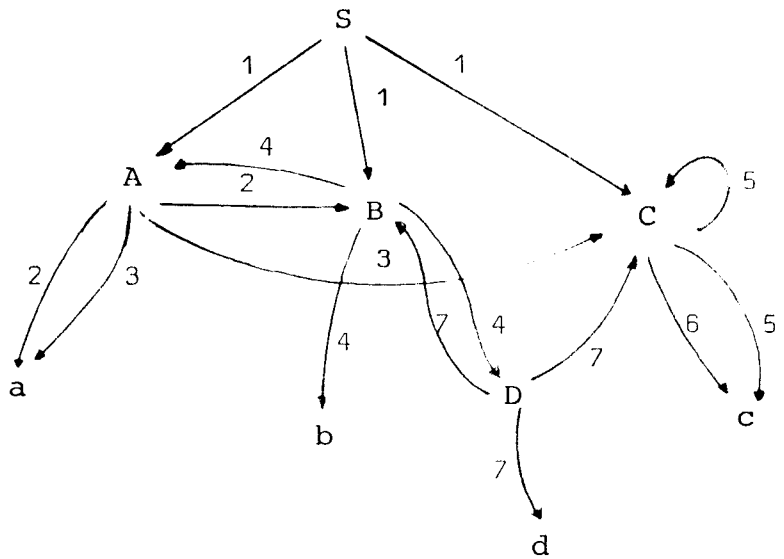
(4) B:: = bAD

(5) C:: = cC

(6) C:: = c

(7) D:: = dCB

seu grafo de dominancia é:



No grafo de dominancia DG(G) para uma gramática G, cada caminho simples (sem ciclos) a partir da raiz até o nó X descreve uma maneira de obter X a partir de S, em uma forma sentencial, sem recursão. Portanto, cada caminho simples da raiz ao nó X se refere a uma ocorrência de X em uma forma sentencial.

Definição 6: Um símbolo terminal z tem um único nível de frase fraco, WPLU, (em inglês, 'weak phrase level uniqueness') dentro de uma gramática G se, ou G é vazia ou

1) existe no máximo uma produção:

$A \Longrightarrow \alpha\ z\ \beta$ que contém z no seu lado direito, e

2) z ocorre apenas uma vez em $\alpha\ z\ \beta$ e

3) A tem WPLU em G(A).

Um símbolo terminal t em G terá WPLU se e sómente se existir um único caminho simples a partir da raiz para o nó t no grafo de dominancia de G. Um símbolo ter WPLU é uma condição suficiente para ele ser fiducial fraco.

Teorema: Se um símbolo terminal z tem WPLU em uma gramática G, e $S \Longrightarrow^{*} \alpha$ é uma derivação LRNR da cadeia α, então z ocorre no máximo uma vez em α.

Teorema: Seja z com WPLU em G e seja $\beta = A_n, A_{n-1}, \ldots A_1, A_o = z$ a cadeia de dominancia para z. Seja α_i o z-sufixo mínimo para A_i, $i = 1,2,\ldots n$. Seja $A_i \Longrightarrow^{*} \gamma_i\ z\ \phi_i$ a derivação LRNR z-imersível. Então $\alpha_i \Longrightarrow^{*} z\ \phi_i$.

Seja, na recuperação de contexto global, A o símbolo do topo da pilha o qual DERIV (A,z) = verdadeiro, onde z é o símbolo fiducial encontrado. Se

1) z tem WPLU em G

2) A está na cadeia de dominancia para z, e

3) dentro do programa, z é produzido a partir de A por uma derivação não recursiva.

então pelo teorema anterior pode-se substituir A na pilha pelo z-sufixo mínimo de A e a cadeia resultante na pilha pode ser capaz de derivar o resto da entrada.

Se A não está na cadeia de dominancia é porque A tem mais que um z-sufixo mínimo e um deles deve ser escolhido arbitrariamente.

Se a suposição 3 não é válida, então após a reconfiguração da pilha, vai-se ter a capacidade de analisar com sucesso apenas um prefixo do restante da entrada.

Pode-se notar que para um símbolo A não terminal ter um único z-sufixo para algum símbolo z, é suficiente que exista um único caminho simples de A para z no grafo de dominancia para a gramática.

Um bom candidato para símbolo fiducial é um símbolo z para o qual existe no máximo um caminho simples a partir de qualquer nó até z no grafo de dominancia para a gramática.

Pode-se escrever uma gramática G para o Pascal na qual a maior parte das palavras reservadas, tais como if, then, else, while, etc. Tenha WPLU em G. Na gramatica original do Pascal pode-se verificar que só mente a palavra reservada program possui WPLU.

Note-se que WPLU de um símbolo em uma dada gramática pode ser decidido algoritmicamente. Então a escolha de símbolos fiduciais pode ser automatizada. Mas é de responsabilidade, do usuário prover uma gramática para uma dada linguagem, na qual um número máximo de símbolos tenha WPLU.

Pai não fornece a gramática utilizada para o Pascal em sua implementação, que segundo suas conclusões apresentou resultados altamente satisfatórios. Mas afirma que para obter tais resultados foram adicionados ao conjunto de símbolos com WPLU, alguns símbolos que não possuem WPLU mas que melhoram sensívelmente a recuperação de erros. Isto reflete que a definição dos símbolos fiduciais é insatisfatória e imprecisa, necessitando-se adicionalmente recorrer-se ao conhecimento prático da utilização da linguagem. Portanto não é conseguida a automatização desejada, mas nos parece ser este o caminho mais inovador tentado até agora no campo de recuperação de erros. Estudos procurando obter definições mais precisas de símbolos tais como os fiduciais de Pai deveriam ser continuados.

CAPÍTULO 4

CONCLUSÕES

Caso já foi dito durante a apresentação do trabalho, fizemos a implementação de quatro dos métodos apresentados:

- o de pânico original, onde na ocorrência de um erro descarta-se a entrada até encontrar um delimitador de comando.

- o de Turner, que por ser extremamente simples queriamos verificar a eficiência.

- o de Wirth, que é um dos mais usados em aplicações reais.

- o de Ghezzi, onde tem-se combinado um acerto local do texto de entrada com uma recuperação "global" nos moldes do método de Wirth.

Nossas implementações foram feitas em um analisador sintático, com pilha explícita e sem retrocesso para um subconjunto bastante completo da linguagem Pascal.

Para testar os métodos, utilizamos uma coleção de 120 programas, encontrados em [Ripley, Druseiks, 1978], que também foi utilizada para testar alguns dos métodos encontrados na bibliografia. Estes programas contém no total 203 erros, sendo estes os mais comumente encontrados em programas Pascal. São programas pequenos de no mínimo 10 linhas de texto.

Ao obter os resultados tivemos uma certa dificuldade em determinar um modo não subjetivo de analisá-los. Infelizmente não o conseguimos e então estabelecemos um critério de classificar as recuperações, por programa e não por erro, da seguinte forma:

excelente: (E) quando o que é feito é exatamente o que o programador faria.

bom: (B) quando o que é feito não é exatamente o que um programador faria, mas não temos muitos erros ignorados e não temos erros inseridos.

regular: (R) quando temos alguns erros inseridos, mas que não chegam a tumultuar demais a deteção dos erros originais.

MAU: (M) quando percebe-se que o analisador se perdeu completamente devido a ocorrência de erros.

Observa-se que é um critério subjetivo e pessoal, mas que de alguma forma pode nos oferecer termos de comparação entre os métodos. Estabelecido o critério, construímos as tabelas que podem ser vistas nas figuras 4.1 e 4.2.

Adicionalmente às tabelas temos os seguintes dados:

- no método de pânico tivemos 79 erros não detetados, enquanto que no de Ghezzi tivemos 41, no de Wirth 45 e no de Turner 65.
- no método de pânico detetamos 22 erros espúrios, no de Wirth 40, no de Ghezzi 46 e no de Turner 60.

A partir da observação destas tabelas é que tiramos algumas das seguintes conclusões:

— se formos observar estatisticamente as percentagens chegariamos à conclusão de que não existe uma diferença sensível entre os métodos testados. Mesmo o método de pânico funciona de maneira satisfatória em 54% dos programas testados. Mas neste percentual deve-

se levar em conta que cerca de 40% dos erros, neste método, não foram detetados.

Ao se escrever um compilador geralmente tem-se uma finalidade para o seu uso. Caso ele seja usado para ensino seria desejável que tivéssemos a maioria dos erros detetados em uma única compilação, pois eles geralmente são muitos. Daí dever-se optar por um método que não descarte tanto texto de entrada detetando a maior parte dos erros, como é o caso da método de Wirth e de Ghezzi. Este último devido à tentativa de acerto local, além de ser mais complexo de se implementar insere mais erros, e acreditamos que em programas maiores, com uma maior quantidade de erros, isto pode se tornar crítico, o que não aconteceu em nossos testes.

Se o compilador for voltado para o uso de profissionais o que portanto não fazem programas com muitos erros, o método de pânico torna-se satisfatório. Deve-se levar em conta também, a estrura da linguagem compilada na escolha de um determinado método de recuperação. Por exemplo, no caso da linguagem Fortran acreditamos ser o método de pânico o mais apropriado, pois nunca teremos muito texto de entrada descartado visto que cada linha é um comando e estes de uma maneira geral são curtos.

Portanto, tratando-se de compiladores que serão mais usados por profissionais deve-se optar pela facilidade de implementação, visto que as diferenças de eficiência não são significativos neste caso. Também deve-se levar em conta os métodos, aos moldes do método de Ghezzi, que tentam um acerto local, que embora sejam de implementação mais complexa, devem resolver a grande parte dos erros encontrados, visto a natureza dos erros que serão encontrados nos programas submetidos

a estes compiladores. E isto ocorrendo, dispensam-se compilações posteriores, o que geralmente é muito desejável em tais sistemas.

— o método de Turner, pelo menos para o caso de linguagem Pascal mostrou-se bastante insatisfatório. Comparando-se com o método de pânico sómente apresenta uma melhora em relação à quantidade de erros detetados, mas a inserção de erros espúrios é por demais significativa. Acreditamos que isto acontece por não ser absolutamente verdadeira a afirmação de Turner de que a maioria dos erros é de primeira ordem, pelo menos não é o que mostra a coleção de programas utilizada em nossos testes e que foram coletados como sendo padrão de erros encontrados na maioria dos programas reais.

— acreditamos que nenhum dos métodos seja restrito à qualquer tipo de implementação. Caso seja utilizada a implementação recursiva não é aconselhável o uso dos métodos que fazem acertos locais, pois complica, de maneira sensível, o analisador se formos coletar o contexto ao redor do ponto do erro de modo a se fazer um acerto local satisfatório.

— comparando-se os métodos de Wirth e Ghezzi verificamos que em poucos casos o acerto local acrescenta alguma melhoria ao método, e na maioria das vezes em que é feito contribui para a inserção de erros espúrios. Isto vem confirmar o que previamos de que acerto locais não são satisfatórios pois para serem o programa deveria conhecer as intenções do programador e isto só é possível em pouquíssimos casos.

FIG. 4.1 - TABELA DE CLASSIFICAÇÃO DOS MÉTODOS

	PÂNICO	GLEZZI	WIRTH	TURNER
1	E	E	E	M
2	B	M	M	M
3	M	M	M	M
4	M	R	R	R
5	M	R	M	R
6	R	E	M	R
7	E	E	E	E
8	E	E	E	E
9	E	E	E	E
10	R	R	R	R
11	E	E	E	E
12	R	R	R	R
13	M	E	E	R
14	M	B	B	B
15	B	B	B	B
16	R	M	R	E
17	R	B	B	M
18	R	R	R	R
19	M	R	E	R
20	R	R	R	R
21	B	B	B	M
22	E	B	E	B
23	E	E	E	R
24	R	M	B	R
25	E	E	E	M
26	B	M	M	M
27	E	E	E	E
28	B	B	B	R
29	R	M	M	B
30	B	B	B	B
31	B	B	B	B
32	B	B	B	M
33	E	E	E	E
34	B	R	B	M
35	R	R	R	R
36	E	E	E	E
37	R	R	R	R
38	R	R	R	R
39	B	B	B	B
40	B	B	B	B

	PÂNICO	GLEZZI	WIRTH	TURNER
41	B	B	B	M
42	E	E	E	B
43	E	E	E	E
44	M	M	E	E
45	R	B	M	R
46	E	E	B	M
47	E	E	E	E
48	B	E	E	M
49	E	B	E	E
50	R	E	E	R
51	E	E	E	E
52	E	E	R	M
53	E	E	E	E
54	E	E	E	E
55	B	E	E	E
56	M	M	M	M
57	R	M	B	B
58	R	B	B	B
59	M	M	M	M
60	B	R	R	R
61	B	B	B	B
62	E	E	E	E
63	E	E	E	E
64	M	M	M	M
65	E	E	E	E
66	E	E	E	B
67	B	B	B	B
68	E	E	E	E
69	E	B	B	R
70	E	E	E	E
71	B	M	B	R
72	B	B	B	M
73	B	B	B	R
74	M	B	B	B
75	B	E	E	M
76	R	R	R	B
77	E	E	E	E
78	R	E	R	R
79	R	R	R	M
80	B	E	E	R

FIG. 4.1 - (Cont.)

	PANICO	GLEZZI	WIRTH	TURNER
81	M	R	R	M
82	R	R	R	R
83	M	E	E	M
84	M	M	B	R
85	M	M	M	B
86	B	B	B	B
87	B	M	M	M
88	R	R	R	R
89	E	R	R	E
90	R	R	R	M
91	R	R	R	E
92	B	B	B	B
93	M	M	M	M
94	E	E	E	E
95	M	M	M	M
96	B	R	R	R
97	E	E	E	E
98	B	R	B	R
99	M	M	M	M
100	R	M	M	R
101	M	M	M	M
102	E	E	E	E
103	E	E	E	E
104	M	M	R	M
105	R	M	R	B
106	M	E	M	R
107	M	M	M	M
108	B	B	B	R
109	M	R	R	R
110	M	M	M	R
111	B	B	B	B
112	M	M	E	E
113	E	R	R	M
114	M	M	M	M
115	E	E	E	E
116	M	M	M	M
117	M	M	M	R
118	E	E	E	E
119	R	M	M	R
120	E	E	E	M

FIG. 4.2 - TABELA RESUMO COM PERCENTAGENS

MÉTODO / recuperação %	PÂNICO	GLEZZI	WIRTH	TURNER
E	29	35	35	27
B	25	21	24	17
R	23	20	21	29
M	23	24	20	27

BIBLIOGRAFIA

[1] Aho, A.V. e Ullman, J.D.
Principles of Compiler Design
Addison Wesley Publisting Company, 1977

[2] Amman, U e outros,
Compilador Pascal - 1977.

[3] Amman, U
Summary on Error Recovery in Recursive Descent Parsers
Preliminary Document (Vol 2) of State of The Art and Future Trends in Compilation Monte pellier, january 9-20, 1978, p 231-233.

[4] Conway, R.W. e Wilcox, T.R.
Design and Implementation of a Diagnostic Compiler for PL/I.
CACM, março 1973, vol 16, num 3, p 169-179.

[5] Fisher, C.N. e Milton, D.R., Quiring, S.B.
An Efficient Insertion - Only Error Corrector for LL(1) Parsers.
In Proc 4Th ACM Symp Principles of Programming Languages 1977 - p 97-103.

[6] Freeman, D.N.
Error Correction in CORC: The Cornell Computing Language,
Ph. D. Th , Cornell U., Ithaca, N.Y, Sept 1963.

[7] Ghezzi, C.
Automatic Recovery and Correction of Syntactic Errors in top-down Compilers
Journal of cybernetics, 1976, vol 5, nº 3

[8] Grahan, S.L. e Rhodes, S.P.
Practical Suntatic Error Recovery,
CACM, nov 1975, vol 18, nº 11

[9] Gries, D.
The Use of Transition Matrices in Compiling
CACM, vol 11, jan 1978, p 26-34

[10] Gries, D.
Compiler Construction for Digital Computer John Wiley e Sons, 1971.

[11] Hartmann, A.C.
A Concurrent Pascal Compiler for Minicomputers
Springer - Verlag, Berlin, 1977.

[12] Horning, J.
What The Compiler Shoud Tell The User
Springer Lectures Notes in Computer Science, nº 21

[13] Irons, E.T.
An Error-Correcting Parse Algorithm
CACM - vol 6 - nov. 1963 - p 669-673.

[14] James, E.B. e Partridge, D.P.
Adaptative Correction of Program Statements
CACM - vol 16 - jan. 1973, p 27-37

[15] Kowaltowski, T.
Implementação de Linguagens de Programação
Escola de Computação - S.P. - 1979.

[16] La France, J.E.
Optimization of Error Recovery
ACM Sigplon Notices, dez. 1970, p 3-17

[17] Morgan, H.L.
Spelling Correction in System Programs
CACM - vol 13, fev. 1970, p 90-94

[18] Musinski, J.E.
Lookatead Recall Error Recovery for LALR Parsers.
Sigplan Notices, out. 1977, p 48-60.

[19] Pai, A.B.
Suntax Driven Error Recovery in Top-Downs Parsing
Ph. D. Th, ago. 1978.

[20] Pai, A.B. e Kieburtz, R.B.
Global Context Recovery: A new Strategy for Sintatic Error Recovery by Table-Driven Parsers.
ACM Transaction on Programming Languages and Systems, vol 2, nº 1, jan. 80, p 18-41.

[21] Pemberton, S.
Comments on an Error-Recovery by Hartmann
Software-Practice and Experience vol 10, p231-240 (1980)

[22] Ripley, G.D. e Druseiks, F.C.
A Statistical Analysis of Sintax Errors
Comput Lang 3 - 1978, p 227-240.

[23] Setzer, V.W. e Melo, I.S.H.
A Construção de um Compilador, parte I
Segunda Escola de Computação, 1981.

[24] Turner, D.A.
Error Diagnosis and Recovery in one Pass Compiler
Information Processing Letters
Vol 6 - nº 4 - ago. 77

[25] Vanini, F.A. e Burnier, R.
Um Gerador de Compiladores com Tratamento Automático de Erros
DCC - IMECC - UNICAMP - 1975.

[26] Wagner, A.
Order - n Correction for Regular Languages
CACM - mai 1974 - vol 7 - nº 5 - p 265-268.

[27] Wirth, N.
A Programming Language For The 360 Computers
JACM - vol 15 - jan 1968 - p 37-74

[28] Wirth, N.
Algorithms + Data Structures = Programs.
Prentice - Hall, INC, 1976.