XIII SEMISH VI CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO
RECIFE-OLINDA 19 A 25 DE JULHO DE 1986

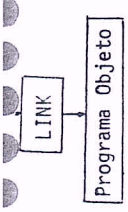
SIC: UM SISTEMA DE SUPORTE A IMPLEMENTAÇÃO DE COMPILADORES

Mariza Andrade da Silva Bigonha* Roberto da Silva Bigonha**

SUMARIO

SIC é uma ferramenta destinada á assistir a implementação de compiladores através de uma linguagem especial baseada no Pascal e denominada LPC. LPC possui, além das construções do Pascal, facilidades para a especificação de sintaxe de linguagens, de meios de associar rotinas semânticas às produções de gramáticas e possibilita, sem perda de eficiência, a implementação de compiladores de vários passos onde cada passo opera diretamente no fonte, eliminando a necessidade de linguagem intermediária.

A partir da especificação da sintaxe, o SIC gera tabelas LALR(1) compactadas e um reconhecedor sintático acrescido de um mecanismo de recuperação de erro independente de linguagem que fornece mensagens de erros automaticamente. O Sistema ainda provê facilidades que permitem a resolução "ad-hoc" de conflitos da Tabela LALR(1) e o uso de gramáticas ambíguas na produção de analisadores sintáticos determinísticos.



ABSTRACT

SIC is a programming tool whose purpose is to assist the development of compilers by means of a special purpose language (LPC) based on Pascal. LPC possesses facilities to specify the syntax of programming languages, associate semantic routines to grammar productions. It also provides, without loss of eficiency, facilities to implement interactive or batch compilers organized into one or more passes, where each pass operates directly on the source code, requiring no intermediate language.

From the given syntax especification, the LPC compiler produces a compressed LALR(1) table and a parser containing a language independent error handling and recovery routine. The System also presents facilities to explicity solve LALR(1) conflicts resulting from the use of ambiguous grammar.

* Mestre em Ciência da Computação (UFMG, 1985). Analista do DCC - UFMG. Interesse: Linguagens e Compiladores.

** Doutor em Ciência da Computação (UCLA, 1981). Professor Adjunto do DCC UFMG. Interesse: Linguagens e Compiladores.
Departamento de Ciência da Computação, UFMG CEP 30161,CP 702 BH. MG.

1 INTRODUÇÃO

O SIC é baseado na linguagem LPC que foi projetada com o objetivo servir como ferramenta adequada à escrita de compiladores, em particul de "front-end" de compiladores. Esta linguagem pode ser vista como linguagem Pascal estendida, ou seja, ela contém todos os recursos linguagem Pascal além das extensões destinadas à facilitar a escrita compiladores [Bigonha M., 1985]. As principais extensões são:

(a) Os mecanismos para definir a gramática da linguagem a ser implem tada juntamente com suas respectivas rotinas semânticas;

(b) Os mecanismos para gerar a pilha semântica automaticamente a par da declaração explícita de atributos (AHO & ULLMAN,1972) associa a símbolos da linguagem;

(c) As facilidades para agrupar declarações de constantes, rótul tipos e variáveis da linguagem Pascal com declarações de procedim tos;

(d) Facilidades para a implementação de compiladores de vários pas onde, cada passo opera diretamente no fonte, eliminando a necessid de linguagem intermediária.

(e) As facilidades que permitem a resolução "ad-hoc" de conflitos Tabela LALR(1) via relação de precedência e associatividade, possi litando o uso de gramáticas ambíguas para produzir analisado sintáticos determinísticos (AHO & ULLMAN, 1977).

A maior parte das novas construções introduzidas visam à facilitar expressão dos mecanismos de compilação. Algumas, entretanto, foram inc poradas a LPC com o objetivo de suprir deficiências do Pascal com rela à programação modular. Por exemplo, é permitido em LPC ao declarar variáveis globais distribuí-las na parte de declaração do programa acordo com o seu uso. O mesmo recurso existe para os outros tipos declarações da linguagem Pascal, permitindo ao projetista do compila simular módulos, ou seja, pode-se agrupar várias declarações de co tantes, tipos, variáveis, rótulos e procedimentos, deixando ao SI incumbência de reconhecer essas declarações e dar-lhes a interpreta correta.

O SIC foi implementado em Pascal, executa em Micros do tipo IBM-PC sistema operacional compatível com o MS-DOS da Microsoft e gera compi dores em Pascal. A configuração mínima requerida é de 256K de RAM e unidade de disquete.

objetivo de
particular,
a como uma
recursos da
escrita de

implemen-

nte a partir
associados

s, rótulos,
procedimen-

rios passos
necessidade

onflitos da
de, possibi-
analisadores

facilitar a
foram incor-
com relação
declarar as
programa de
os tipos de
compilador
ões de cons-
ndo ao SIC a
nterpretação

IBM-PC com
ra compila-
de RAM e 1

Na descrição da sintaxe de LPC será usado o seguinte formalismo: Os não-terminais da gramática são escritos em letras minúsculas e os símbolos terminais sempre colocados entre aspas. Cada produção tem a forma S = E ; onde S denota um símbolo não-terminal e E as alternativas que definem S. O termo E tem a forma T1 | T2 | ... | Tn (n > Ø), onde cada termo Ti, Ø < i < n, tem a forma F1 F2 ... Fm (m > Ø), onde cada elemento Fi, Ø < i <= m, pode ser: um símbolo terminal, um símbolo não-terminal, { T }, [T] ou (E). Uma sequência de F's denota a concatenação dos F's envolvidos. { T } indica sequência de zero ou mais T's. [T] representa um termo T que pode ser omitido e (E) representa um agrupamento de alternativas.

Os símbolos básicos em LPC são:

```
símbolos_básicos = terminal | nãoterminal | índice | texto
                 | identificador | palavra_chave
terminal         = """ sequência de caracteres diferente de aspas """
nãoterminal      = identificador
índice           = " [ " número " ] " "."
identificador    = letra caracter_de_id
caracter_de_id   = letra | dígito | "_"
dígito           = "Ø" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
                 | "9" ;
letra            = "a" | ...| "z" | "A" | ... | "Z"
```

Um índice é utilizado para se formular, no contexto de rotinas semânticas, uma referência não ambígua a símbolo terminal ou não-terminal em uma produção. Texto são sequências de construções do Pascal.

As seguintes palavras-chave são reservadas: %%COMPILER, %%BATCH, %%INTERACTIVE, %%STACK, %%TOKENS, %%CONSTANTS, %%TYPES, %%VARIABLES, %%LABELS, %%PROCEDURES, %%PROGRAM, %%GRAMMAR, %%SCOPEMAP, %%NTMAP, %%END, %%FIRST, %%OTHER , %%CONFLICTS, %%PREC, %%LEFT, %%RIGHT, %%NONE.

2.2 Programa em LPC

Um programa em LPC tem a forma:

```
sic  = "%%COMPILER" texto opções [passos] seção_um [seção_dois] corpo
       "%%END" (";" | ".");
nome = identificador ;
```

O texto após %%COMPILER constitui o cabeçalho do compilador gerado,

permitindo a produção de compilador na forma de procedimento ou de programa Pascal.

Exemplos:
%%COMPILER procedure COMPILADOR;
%%COMPILER program EXEMPLO(input,output);
As demais cláusulas são descritas a seguir.

2.3 OPÇOES DE COMPILAÇÃO

A cláusula opções serve para especificar o tipo de processamento "batch" ou "interativo" a ser usado no compilador a ser gerado.

opções = "%%INTERACTIVE" "NOREC" | "%%INTERACTIVE" "REC" | "%%BATCH"

A palavra REC após %%INTERACTIVE indica ao SIC que o analisador sintático interativo a ser gerado deve tentar identificar todas as correções possíveis que permitiriam a análise sintática prosseguir. Com esta opção, as ações necessárias a uma recuperação automática do erro serão efetuadas e as mensagens correspondentes às correções possíveis emitidas. Ao usuário é dada a opção de ignorar o erro e prosseguir a compilação.

Por outro lado, o uso de NOREC após %%INTERACTIVE, indica que a posição de cada erro de sintaxe deverá ser apenas indicada sem que se tente determinar a real causa do erro. Neste caso, a compilação só poderá continuar após o erro ter sido eliminado.

Em ambos os casos, após a indicação do erro, o controle da execução passa ao editor de texto de forma a permitir ao usuário efetuar as correções necessárias.

Com a opção "%%BATCH", a recuperação automática de erro é sempre tentada, emitindo-se as respectivas mensagens sem que se interrompa a compilação. O SIC adota o método de recuperação de erros proposto por [Burke & Fisher, 1982].

2.4 PASSOS DO COMPILADOR

A cláusula passos serve para informar ao SIC o número de passos que o compilador deverá ter. A presença da cláusula %%FIRST PASS no programa fonte indica que o compilador será de vários passos e que este é o primeiro deles.

A cláusula %%OTHER PASS indica os demais passos do compilador. A omissão de %%FIRST PASS e %%OTHER PASS implica em um compilador de um único

passo.

passos = "%%FIRST" "PASS" | "%%OTHER" "PASS"

Exemplos:

```
%%COMPILER PROGRAM exemplo; (*$R+*) %%BATCH
%%COMPILER PROCEDURE exemplo; %%INTERACTIVE  NOREC
%%COMPILER PROCEDURE exemplo; %%BATCH %%FIRST PASS
%%COMPILER PROGRAM exemplo(input,output); %%INTERACTIVE REC %%FIRST PASS
%%COMPILER PROCEDURE compilador(a,b:Integer); %%OTHER PASS
```

Cláusulas indicativas de opções usadas em conjunção com %%OTHER PASS são ignoradas porque após o primeiro passo somente a modalidade "BATCH" é permitida.

2.5 SEÇÃO_UM

A seção seção_um serve para especificar (a) os símbolos terminais da linguagem sendo implementada; (b) a declaração dos atributos associados aos símbolos não-terminais e terminais da gramática dessa linguagem; (c) a declaração dos mapas de escopo, isto é, dos símbolos terminais que atuarão como abridores de escopo e seus respectivos fechadores; (d) a declaração dos símbolos não-terminais que poderão ser inseridos ou substituídos na recuperação de erro e (e) declarações usuais da linguagem Pascal.

```
seção_um = { def_de_terminais | atributos | mapadeescopo |
             mapadenaoterminais | declarações }
```

As declarações de terminais, de não-terminais, de mapas de escopo e dos atributos só podem aparecer uma vez em cada módulo de compilação. Declarações da linguagem Pascal podem aparecer várias vezes e em qualquer ordem.

A ordem dos cinco elementos que formam a seção_um é irrelevante. Entretanto, se símbolos terminais forem usados em declarações da linguagem Pascal, a definição desses terminais deve aparecer antes.

2.6 SEÇÃO_DOIS

A seção seção_dois serve para especificar as regras da gramática e resolver conflitos do analisador LALR(1).

```
seção_dois =   gramática  resolução_de_conflitos
```

2.7 Seção de Terminais

A finalidade desta seção é estabelecer um mecanismo de comunicação entre os analisadores sintáticos gerados pelo SIC (procedimento YYPARSER) e o analisador léxico (procedimento YYSCAN) que deve ser projetado pelo usuário.

def_de_terminais = "%%TOKENS" terminal "=" identificador [";"]

O usuário deve usar esta seção para associar a cada "token" um identificador que será tratado pelo SIC como uma constante que especifica o tipo léxico do símbolo. Dentro do analisador léxico o usuário deverá usar estes identificadores para definir os valores retornados pelo procedimento YYSCAN através da variável pré-declarada YYSIMB.

O analisador sintático somente reconhece os "tokens" listados nesta seção, referindo-se a eles através dos identificadores das constantes que definem os seus tipos léxicos. É obrigatória a presença de um símbolo terminal associado ao identificador YYEOF, que dá o tipo léxico da marca de fim do arquivo de entrada.

Exemplo

```
%%TOKENS
"id" = xid;     "<" = xmenor  "cte" = xcte;   ">" = xmaior;
"eof" = YYEOF;  "if" = xif
```

2.8 Seção de Atributos

O SIC implementa o esquema clássico de geração pós-fixada de código dirigida por sintaxe (AHO & ULLMAN, 1972, 1973). Neste método, aos símbolos da gramática devem ser associados atributos sintetizados (KNUTH, 1968) os quais têm seus valores determinados na medida que a análise sintática é efetuada. Estes atributos correspondem a posições de uma pilha auxiliar, a pilha semântica, alocada ao lado da pilha sintática de estados do reconhecedor LALR(1). Desta forma, os valores numa dada posição da pilha semântica denotam os atributos do símbolo representado pelo estado na posição correspondente da pilha sintática.

A seção de atributos tem por finalidade prover informações necessárias à geração da pilha semântica. Identifica esta seção a palavra chave %%STACK. O número após %%STACK fornece o tamanho máximo das pilhas sintática e semântica. A omissão desse número implica na geração pelo SIC de um tamanho "default". Após as palavras "OF ATTRIBUTES" são especificados os atributos para cada símbolo terminal e não-terminal da gramáti-

ca, seus tipos e as respectivas ações de inicialização de atributos.

```
atributos         = "%%STACK" [número] "OF" "ATTRIBUTES" def_de_atributos
                    ";"
def_de_atributos  = def_atrib      { def_atrib }
def_atrib         = símbolo "=" "(" lista_de_atrib ")" [ inicialização ]
                    [ ";" ]
lista_de_atrib    = decl_de_atrib    { ";" decl_de_atrib }
decl_de_atrib     = nomes_de_atrib   ":"   nome_do_tipo
nomes_de_atrib    = identificador    { "," identificador }
nome_do_tipo      = identificador
inicialização     = " { "  [ texto ]  " } "
símbolos          = nãoterminal  |  terminal
```

Exemplos

```
%%STACK 200 OF ATTRIBUTES
exp    =  (R,tipo : integer) { exp.r := 0; exp.tipo := 0 };
"id"   =  (valor : integer)  {GERATEMPORARIO(t); "id".valor:=INSTALA(t)};
"cte"  =  (valor  : integer) { "cte".valor := 0 };
dcl    =  (DD     : integer; A,B,C : boolean)
cond   =  (quad   : integer)      { cond.quad := 0 }
```

Ações de inicialização de atributos somente são executadas quando os símbolos da gramática forem inseridos no programa fonte como resultado de ação de recuperação de erro. A ação de inicialização permite ao usuário assegurar que a pilha semântica sempre contém valores bem definidos. A ausência de inicialização de atributos pode causar erros de execução no compilador gerado quando houver inserção ou substituição de símbolos, terminais ou não-terminais, durante a fase de recuperação. Com este mecanismo de inicialização, as rotinas semânticas do usuário podem ser ativadas normalmente, mesmo após erros de sintaxe tiverem sido detectados durante a compilação.

Todo símbolo, terminal ou não-terminal, possui implícitamente os atributos YYLINHA e YYPOS que denotam a posição inicial do símbolo no arquivo de entrada. Estes atributos, que não podem ser modificados, podem ser usados para melhorar a qualidade de mensagens de erro.

2.9 Seção de Abridores e Fechadores de Escopo

A seção mapadeescopo tem por finalidade permitir a especificação dos abridores e fechadores de escopo. Escopo é definido como sendo construções sintaticamente aninhadas, tais como, procedimentos, blocos, estruturas de controle e expressões parentetizadas, [Burke & Fisher, 1982].

Abridores e fechadores de escopo são pares de símbolos que iniciam e terminam, respectivamente, estas construções. Por exemplo, os pares "(" - ")", "begin" - "end"; "if" - "end if", são delimitadores típicos de escopo.

O conhecimento dos símbolos que são abridores e fechadores de escopo da linguagem possibilita a implementação de um sofisticado mecanismo de recuperação de erro, denominado recuperação de escopo.

```
mapadeescopo = "%%SCOPEMAP"  def_scopemap  { def_scopemap }
def scopemap = "terminal"    ":"  listacloser  ";"
listacloser  = dclcloser  { ","   dclcloser }
dclcloser    = "terminal"  { "terminal" }
```

Exemplo:

```
%%SCOPEMAP  "("  :  ")"  ;  "if" : "end" "if" , "end" ; "while" : "do";
```

2.10 Seção Mapa de Não-terminais

A seção mapa_de_não_terminais permite a declaração dos símbolos não-terminais da gramática que poderão ser usados durante a recuperação de erros como candidatos à inserção ou substituição de um símbolo. Recomenda-se listar nesta seção somente os não-terminais que sejam familiar aos usuários.

```
mapa_de_não_terminais  = "%%NTMAP" listant ";"
listant                = nãoterminal  { ","  nãoterminal }
Exemplo:     %%NTMAP exp, cmd, dcl ;
```

2.11 Seção de Declarações

A seção de declaração permite a declaração de rótulos, tipos, constantes, variáveis e procedimentos Pascal.

```
declarações   =   membros
membros  =  "%%LABELS" texto | "%%TYPES" texto |  "%%CONSTANTS"  texto  |
            "%%VARIABLES" texto | "%%PROCEDURES"  texto
Exemplos
%%CONSTANTS   TMID  = 8;
%%LABELS      1,2;
%%VARIABLES   x : alpha;   y : integer;
%%TYPES       alpha = array [1..TMID] of char;
%%PROCEDURES
      procedure YYSCAN; begin ... end;
```

niciam e
ares "(" -
picos de

escopo da
nismo de

"do";

ios não-
ração de
Recomen-
niliar aos

nstantes,

texto |


```
        functior INSTALA( ... ); begin  ...  end;
%%VARIABLES ch : char;
```

Se o programa em LPC contiver a opção de único passo ou de primeiro passo, é obrigatória a declaração do procedimento YYSCAN, responsável pela análise léxica. YYSCAN é um procedimento sem parâmetros que é chamado automaticamente pelo YYPARSER (o analisador sintático gerado) sempre que o próximo "token" no fluxo de entrada for necessário. A cada chamada, YYSCAN deve retornar na variável inteira pré-definida YYSIMB o tipo do "token" lido; em YYLINHA o número da linha fonte que contém o "token" e em YYPOS a posição do "token" dentro da linha. Caso o "token" possua um valor, este deverá ser armazenado em um de seus atributos. Por exemplo, se o "token" "cte" tiver um atributo valor do tipo "INTEGER", YYSCAN poderá retornar o valor inum de uma constante inteira lida mediante a atribuição

```
"cte".valor := inum ;
```

2.12 Seção Gramática e Rotinas Semânticas

No esquema de geração de código dirigida por sintaxe implementado no SIC, rotinas semânticas que definem as ações necessárias à geração de código devem ser associadas às produções da gramática. Essas rotinas serão ativadas quando as produções associadas forem usadas para reduzir elementos gramaticais na região do topo da pilha sintática.

A seção gramática serve para especificar as produções da gramática da linguagem a ser implementada e as rotinas semânticas associadas. A gramática dada é usada para produzir as tabelas do analisador sintático LALR(1).

```
gramática        = "%%GRAMMAR" nãoterminal [ "AND" "SEMANTICS" ]
                     produções
produções        = nãoterminal "="  def_prod { "|" def_prod } ";"
def_prod         = {símbolos} [ precedência ] [ rotina_semântica ]
precedência      = "%%PREC"  token | "%%PREC" identificador
rotina_semântica = " { "  [ texto ]  " } "
```

A rotina ou ação semântica, associada a cada produção é definida por um ou mais comandos em Pascal colocados entre chaves, e como tal, pode fazer entrada e saída, invocar procedimentos, alterar valores de variáveis e valores externos; salvar tabela de símbolos, etc. Comentários neste contexto só podem ser delimitados pelo par (* *), dado que o par { }

é usado como delimitador de rotina semântica.

Para que referências a símbolos da gramática, i.e., terminais e não-terminais, ocorrendo do lado esquerdo ou lado direito de uma produção da gramática, sejam feitas qualifica-se o nome do símbolo pelo seu respectivo atributo da mesma forma que se qualifica um campo de "record" em Pascal. Referências ambíguas são resolvidas via índices. Adota-se a convenção de que um símbolo sem índice designa a sua primeira ocorrência na produção associada, contando com o lado esquerdo, e símbolo com um índice k>=1 designa a sua k-ésima ocorrência na produção. Veja o exemplo:

```
exp  =  exp  +  exp
{if exp[2].tipo <> exp[3].tipo then ERRO(9,exp[2].YYLINHA,exp[2].YYPOS);
 if exp[2].tipo = INTEIRO then op := CMAIS else op := COU;
 temporario := temp; GEN(OP,temporario,exp[2].r,exp[3].r);
exp.r := temporario;   exp[1].tipo := exp[2].tipo;  };
```

No exemplo acima, exp.r e exp[1].tipo designam os atributos r e tipo do exp do lado esquerdo da produção; exp[2].r e exp[2].tipo denotam os atributos r e tipo do primeiro exp do lado direito e exp[3].r, exp[3].tipo do exp mais à direita. As ocorrências de exp na produção propriamente dita não possuem índices.

Note que, pode-se atribuir valores a atributos do não-terminal do lado esquerdo sem que os atributos do primeiro símbolo do lado direito sejam afetados, embora ambos, ocupem a mesma posição na pilha semântica. O SIC salva os valores dos atributos do primeiro símbolo do lado direito em temporários apropriados.

A claúsula %%PREC serve para associar um nível de precedência a uma regra da gramática. %%PREC deve aparecer imediatamente depois da definição do lado direito da regra e antes da ação semântica ou do delimitador ";". O uso de %%PREC faz com que a regra tenha a mesma precedência do símbolo terminal, literal ou identificador especificado na cláusula. A ausência de %%PREC implica que a regra da gramática terá a mesma precedência do símbolo terminal ocorrido mais à direita no seu lado direito. Se não houver este símbolo terminal, a precedência da regra é considerada indefinida. O valor da precedência de uma produção é usado para resolver conflitos da tabela LALR(1). Por exemplo, para que o "-" da produção EXP = "-" EXP tenha a mesma precedência que a multiplicação, ("*"), deve-se escrever exp = "-" exp %%PREC "*";

is e não-
rodução da
respecti-
cord" em
dota-se a
ocorrência
o com um
Veja o

J.YYPOS);

e tipo do
enotam os
exp[3].r,
produção

do lado
ito sejam
ca. O SIC
eito em

a uma
a defini-
elimitador
ência do
usula. A
sma pre-
direito.
nsiderada
resolver
dução EXP
deve-se

2.13 Seção Resolução de Conflitos

Gramáticas livres-do-contexto não-ambíguas constituem uma excelente ferramenta para definição precisa de sintaxe de linguagem de programação e geração automática de reconhecedores determinísticos (HOPCROFT & ULLMAN, 1969). Entretanto, várias construções sintáticas comuns em linguagens de programação são especificadas de forma mais natural e suscinta através de gramáticas ambíguas do que usando uma gramática equivalente não-ambígua.

O SIC concilia estes dois interesses implementando o método de geração de tabelas LALR(1) usando gramáticas ambíguas proposto em (AHO & ULLMAN, 1977). O uso de gramáticas ambíguas levam, inevitavelmente, ao aparecimento de conflitos na tabela LALR(1) que devem ser resolvidos diretamente pelo usuário.

A seção resolução_de_conflitos serve para especificar a precedência de operadores e a associatividade de operadores binários. Esta informação possibilita ao SIC resolver os conflitos existentes na análise sintática e construir um analisador que obedeça as relações de precedência e associatividade estabelecidas.

```
resolução_de_conflitos  =  "%%CONFLICTS"  associação  {[";"]  associação}
                                          [";"]
associação              =  associa_se_a símbolos { "," símbolos }
associa_se_a            =  "%%RIGHT" | "%%LEFT" | "%%NONE"
símbolo                 =  terminal | identificador
```

A declaração de precedência e associatividade é feita através de uma série de claúsulas começando com as palavras chaves %%RIGHT, %%LETF e %%NONE seguida de uma lista de símbolos terminais. Cada claúsula define um novo nível de precedência que é maior do que o da cláusula anterior. Os símbolos terminais declarados na mesma cláusula têm o mesmo nível de precedência e associatividade.

Exemplos

```
%%CONFLITS
%%NONE "<",">","="    %%LEFT "+", "-"    %%RIGHT ZZ    %%RIGHT "*","/"
```

O exemplo descreve a relação de precedência e associatividade para quatro operadores aritméticos e de relação. Em primeiro lugar, "<", ">" e "=" têm precedência menor que "+" e "-" e não se associam. Os operadores "+" e "-" associam-se á esquerda e têm menor precedência que "*" e "/", os quais associam-se à direita.

Uma regra da gramática também pode ter uma precedência que não seja a de

um símbolo terminal da gramática, bastando que se defina um identificador na seção de resolução_de_conflitos, e use tal identificador na cláusula %%PREC correspondente. Por exemplo:

exp = exp "+" exp %%PREC ZZ

indica que essa regra da gramática terá uma precedência (a do identificador ZZ), que é maior que "+" e menor que "*".

2.14 Corpo

A seção corpo serve para especificação do corpo principal do compilador.

corpo = "%%PROGRAM" [texto] ;

Após a palavra chave %%PROGRAM pode ocorrer qualquer comando em Pascal, exigindo-se apenas que o procedimento YYPARSER seja ativado em algum ponto.

3. FUNCIONAMENTO DO SIC

A compilação de um programa em LPC é feita em quatro passos independentes. O primeiro passo recebe como entrada um programa em LPC, armazenado em um arquivo (de entrada) com extensão .SIC, e a partir daí, produz como saída arquivos em disco contendo respectivamente, declarações Pascal, a gramática em uma forma interna, a tabela de produções e uma tabela de terminais e não-terminais. As declarações Pascal encontradas na entrada são armazenadas em arquivos separados conforme sejam declaração de rótulos, declaração de tipo, declaração de variáveis, declaração de constantes ou procedimentos. Também são incluídos no arquivo de procedimentos o procedimento (gerado pelo SIC) que define as rotinas semânticas, o procedimento do analisador sintático com ou sem um recuperador automático de erros sintáticos embutido e o corpo do programa em LPC.

O segundo passo recebe como entrada a tabela de símbolos e a gramática em sua forma interna produzida na fase anterior e armazenados em arquivos, e produz como saída um arquivo contendo a tabela LALR(1) compactada.

O terceiro passo por sua vez, recebe como entrada os arquivos de declarações gerados nos passos anteriores e os une, gerando assim como saída, um procedimento ou um programa completo em Pascal. Para satisfazer a restrições de certos compiladores, por exemplo, Turbo Pascal, o compilador produzido é dividido em até três partes e armazenados em arquivos com extensão .PS1, .PS2 e .PS3 respectivamente. Sendo que cada arquivo pode conter no máximo um fonte de aproximadamente 61.000 bytes. Compete ao usuário unir estes arquivos de alguma forma.

Finalmente, o quarto passo trata da compilação do programa Pascal obtido para produzir um módulo de execução COM que junto com a tabela LALR(1) e a tabela de produções geradas em passos anteriores, formam o compilador desejado.

4 CONCLUSÃO

O SIC baseou-se no YACC [Johnson, 1977] contudo, é mais poderoso que ele em vários aspectos. O SIC possui um mecanismo de recuperação de erro mais elaborado que o do YACC, além de ser transparente ao usuário. O método de recuperação implementado no SIC não faz nenhuma restrição ao uso de reduções "default" e o uso das mesmas provou não prejudicar a recuperação de erro [Bigonha M. & Bigonha R., 1985]. O SIC oferece ainda facilidades para gerar compiladores, em Pascal, com ou sem recuperação de erro.

O mecanismo usado no SIC de associar rotinas semânticas de inicialização a símbolos da gramática utilizados na recuperação de erro viabiliza a estratégia de recuperação envolvendo a inserção ou substituição de não-terminais.

O método de análise sintática adotado é o LALR(1) implementado na forma de tabelas compactadas [Bigonha R. & Bigonha M., 1983; Bigonha R., 1986].

Atualmente o SIC está completamente operacional, ele pode produzir compiladores de um ou mais passos em modalidade "batch". Desenvolvimentos ainda serão necessários para concluir a geração de compiladores interativos com/sem recuperação de erros, em particular, necessita-se concluir a implementação do editor de texto necessário.

5 REFERENCIAS BIBLIOGRAFICAS

Aho,A.V.,Ullman,J.D.,The Theory of Parsing Translation and Compiling,Prentice-Hall,INC,1972.

Aho, A.V., Johnson,S.C., and Ullman, J.D. "Deterministic Parsing of Ambiguous Grammars". Comm. Assoc. Comp. Mach. vol. 18(8) 441-452, August 1977.

Aho,A.V.,Ullman,J.D., Principles of Compiler Design, Addison-Wesley Publishing Company Co. (1973).

Bigonha, Mariza A. S., "SIC : Sistema de Implementação de Compiladores", Tese de Mestrado, Departamento de Ciência da Computação (DCC) Icex-UFMG, junho/1985.

Bigonha, Roberto S. & Bigonha, Mariza A. S.,"Um método de Compactação de Tabelas LR(1), Anais do III Simpósio sobre Desenvolvimento de Software Básico para Micros, 141-151, COPPE UFRJ - 1983.

Bigonha, Roberto S., "Gerador de Tabelas LALR(1)", trabalho a ser publicado como monografia do Departamento de Ciência de Computação - ICEX - UFMG, 1986.

Bigonha, Mariza A.S. & Bigonha, R.S., "SIC: Manual do Usuário", Trabalho publicado como Monografia do Departamento de Ciência de Computação - ICEX - UFMG, 1985.

Bigonha, Mariza A. S. & Bigonha, R. S. "Uma experiência na Implementação de um Recuperador de Erro LR(1)", Anais do V Simpósio sobre Desenvolvimento de Software Básico , UFMG, novembro/1985.

Burke, Michael & Fisher Jr., g.a., "A Practical Method for Syntatic Error Diagnosis and Recovery", ACM, 1982.

Hopcroft, J. E. & Ulmman, J. D. Formal Language and their Relation to Automata. Reading Mass., Addison Wesley, 1969.

Jensen, K. & Wirth, N., PASCAL User Manual and Report, Springer-Verlag 1974.

Johonson,S.C.,"YACC -Yet Another Compiler Compiler".Bell Laboratories,Murray Hill,1977.

Knuth, D. E., "Semantics of Context-Free Languages", Mathematical System Theory 2, (1968) pp. 127-145. Correction : Mathematical System Theory 5 (1971),95-96.

r publi-
- ICEX

Trabalho
utação -

mentação
Desenvol-

ic Error

lation to

-Verlag

ler".Bell

l System
l System

XIII SEMISH VI CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO

RECIFE-OLINDA 19 A 25 DE JULHO DE 1986

UM TRATAMENTO PARA ERROS SINTÁTICOS EM ANALISADORES LR

Zorzo, S.D. * Catto, A.J. **

SUMÁRIO

Apresentamos neste trabalho um algoritmo de tratamento de erros sintáticos em analisadores LR baseado no método proposto por Rohrich, cuja implementação necessita apenas de uma pequena tabela que representa o mapeamento dos estados do analisador em símbolos terminais, obtida no processo de geração das tabelas de análise. O algoritmo diagnostica todos os erros sintáticos encontrados durante a análise sintática sem realizar retrocesso na análise. Seu objetivo é tornar o tratamento de erros independente da linguagem analisada, preservar sua independência da gramática e minimizar a introdução de erros espúrios.

ABSTRACT

An algorithm for syntax error recovery in LR parsers is presented, which is based on a method proposed by Rorich. Its implementation only requires a small table, which maps the states of the parser into the terminal symbols, and which is obtained during the parser's table's generation process. The algorithm diagnoses all syntax errors found during parsing without backtracking. Its aim is to make error recovery independent of the language which is being analysed, to preserve its independence from the underlying grammar and to minimize the introduction of spurious errors.

* Bacharel em Ciência da Computação (UFSCar, 1978), Mestre em Ciência da Computação (ICMSC, USP, 1985); linguagens de programação, construção de compiladores; Universidade Federal de São Carlos, (016) 71-1100 Ramal 143.

** Engenheiro Civil (EESC, USP, 1970), Ph.D in Computer Science (University of Manchester, 1981); computação paralela, arquiteturas dirigidas pelo fluxo de dados, técnicas de programação; Centro Tecnológico para Informática - CTI, (0192) 42-1000 Ramal 176.

1. INTRODUÇÃO

O tratamento de erros sintáticos num processo de compilação é particularmente difícil e muito importante por causa de seus efeitos globais. Um erro é detectado pelo analisador quando não lhe é possível executar uma ação de análise válida.

As ações realizadas no tratamento de um erro irão transformar um trecho incorreto do programa em um novo trecho sintaticamente válido. Tais ações de análise não corresponderão necessariamente à intenção do programador, visto que qualquer trecho incorreto do programa fonte pode ter várias interpretações distintas. Este fato não deve ser interpretado como depreciativo dos métodos de tratamento de erros existentes, pois o objetivo não é corrigir os erros cometidos pelo programador mas, sim, permitir a retomada do processo de análise tão próximo do ponto de erro quanto possível.

O tratamento de erros requer modificações na pilha de análise, na sentença analisada, ou em ambas, por eliminação, inserção ou substituição de símbolos. Os vários métodos de tratamento de erros combinam, de alguma forma, estas três ações.

Dentre as várias alternativas existentes, o método proposto por Rohrich | 4,5 | mostra-se eficiente com respeito a tempo e espaço, é capaz de tratar todos os erros sintáticos e não requer nenhuma interface com o analisador, pois utiliza o próprio algoritmo de análise. Este método utiliza uma pequena tabela que representa um mapeamento dos estados do analisador em símbolos da linguagem, obtido no processo de geração das tabelas de análise sintática.

A seguir, descrevemos este método de tratamento de erros, sua aplicabilidade à análise LR e uma proposta para torná-lo independente da linguagem analisada.

2 . DESCRIÇÃO DO MÉTODO

A idéia básica do método é associar a cada estado q do analisador sintático um símbolo da linguagem f(q). Quando um erro é detectado, a pilha de análise é salva e a análise continua, usando f(q) no lugar da entrada original. A função f é escolhida de tal forma que este processo termine rapidamente, depois de passar pelos estados q1, q2, q3,..., qn. Alguns símbolos da entrada podem ser ignorados até que um símbolo visualizado t seja aceitável em um dos estados qi. O erro é corrigido pela inserção de f(q1), f(q2),...,

mente difícil e
pelo analisador

ncorreto do pro-
corresponderão
rreto do progra-
er interpretado
objetivo não é
da do processo de

a analisada, ou
métodos de tra-

4,5 | mostra-se
os sintáticos e
goritmo de análi-
dos estados do
belas de análise

de à análise LR

co um símbolo da
análise conti-
tal forma que
q3,..., qn. Al-
do t seja acei-
q1), f(q2),...,

f(qi - 1) à esquerda de t. Depois de restaurar a pilha de análise, esta continua normalmente.

Observa-se que o método independe da especificação gramatical da linguagem analisada, pois faz o tratamento dos erros manipulando cadeias de símbolos e não estruturas de frases da gramática.

Seja z uma sentença analisada por um autômato que reconhece sentenças de uma linguagem $L \in Vt^*$. Se $z \notin L$, denotaremos o erro de análise através do par ordenado (u,v), onde z = uv e u é o maior prefixo comum entre z e L. Portanto, deve existir ao menos uma cadeia w, tal que $uw \in L$. A cadeia w é chamada continuação de u.

Podemos corrigir o erro de análise (u,v) da sentença z, substituindo z por z' = uw'v'', onde w = w'w'' e v = v'v''. Esta correção é chamada correção sem retrocesso, pois o prefixo correto u da sentença não foi alterado.

Uma boa correção preservará o máximo de v, ou seja, v'' deve ser tão longo quanto possível, enquanto w' deve ser o menor possível. A sentença corrigida z' = uw'v'' pode ainda conter erros de análise dentro da cadeia v'', ou seja, pode ocorrer o erro (uw'x,y), onde xy = v'', com $x \neq \varepsilon$. O processo de correção será repetido enquanto persistir essa situação. A exigência de $x \neq \varepsilon$ garante a não ocorrência de infinitas repetições

O algoritmo a seguir, proposto em | 4,5 |, corrige erros de análise (u,v) na sentença z da linguagem L, como descrito anteriormente.

```
Determine uma continuação w de u;
Determine o conjunto H dos símbolos terminais,
  chamados âncoras, que são aceitos pelo autômato
  depois de inserir algum prefixo de w, ou seja,
  H.={ h ∈ Vt | ∃ w', onde w = w'w" e uw'h é prefixo de L} ;
Tente decompor v em v'hv", tal que h ∈ H e
  v' tenha o menor comprimento possível;
Se essa decomposição for possível
    Decomponha w em w'w", tal que
      w' tenha o menor comprimento possível e
      uw'h seja prefixo de L;
    Faça a correção em z, substituindo v' por w',
      ou seja, faça z' = uw'hv"
Senão
    Faça a correção em z, substituindo v por w,
      ou seja, faça z' = uw.
```

Esse algoritmo será modificado posteriormente para produzir melhores correções. No entanto, neste momento, o que importa é o fato de que ele sempre corrige, podendo ser aplicado a qualquer sentença z ∈ Vt* e ser repetido até que a sentença resultante pertença à linguagem.

Com exceção da determinação de w, no início, o algoritmo é simples e de fácil implementação, resumindo-se a tarefa de correção à determinação da continuação de um prefixo da linguagem.

3. APLICABILIDADE À ANÁLISE LR

Para que o algoritmo descrito anteriormente possa ser utilizado é necessário que o analisador sintático detecte cada situação de erro antes de processar a parte errada da sentença, permita a determinação da continuação de forma determinística, simples e eficiente, e tenha consistência na análise, isto é, as configurações do analisador na análise de (u,v) devem ser idênticas àquelas na análise de (u,w).

Rohrich | 4,5 | prova que é decidível verificar se um particular autômato com pilha é consistente e continuável, ou seja, se ele permite a construção de uma continuação de um

orreções. No
podendo ser
ante perten-

e de fácil
ção de um pre-

que o anali-
ada da sen-
s e eficien-
na análise de

om pilha é
uação de um

prefixo da sentença analisada. Se for consistente e continuável, esse autômato reconhecedor terá a propriedade dos prefixos corretos, ou seja, ao detectar um erro, o analisador não terá ainda processado qualquer porção incorreta da sentença de entrada.

Analisadores sintáticos LR possuem a propriedade dos prefixos corretos, mas, embora essa técnica de construção sempre produza analisadores consistentes, algumas otimizações podem introduzir inconsistências.

Se um autômato com pilha for continuável, a pesquisa de todas as possibilidades de continuação, a partir de uma dada configuração, permite encontrar uma cadeia de continuação. No entanto, isto é impraticável, pois a continuação deve ser encontrada de forma determinística.

Para encontrar essa continuação de forma determinística, Rohrich sugere que, a partir de um autômato com pilha A, consistente, construa-se um núcleo K de A, que também é um autômato consistente, com todas as transições definidas por A, sem preocupação com os símbolos de entrada que controlam suas transições. O eventual não-determinismo do núcleo K poderá ser eliminado, criando-se um autômato com pilha F, contendo apenas uma transição definida para cada configuração de análise. F será o autômato de continuação terminal para A, que será dito deterministicamente continuável.

É decidível se um autômato com pilha é deterministicamente continuável, ou seja, se ele possui um autômato de continuação terminal. Para um dado autômato com pilha, deterministicamente continuável, podem existir vários autômatos de continuação terminal, razão pela qual deve haver uma política de escolha.

Com a construção do autômato de continuação terminal, a cadeia de continuação poderá ser obtida de forma simples e eficiente.

No caso de analisadores sintáticos LR, LALR e SLR, um autômato de continuação terminal poderá ser obtido por meio dos algoritmos de construção das tabelas de análise. Para isso, aplicamos os princípios de continuação para gramáticas, produzindo uma gramática de continuação.

Uma gramática de continuação para uma determinada gramática é tal que haja apenas uma produção para cada símbolo não terminal e qualquer forma sentencial obtida pela gramática

derivará numa cadeia terminal pela aplicação das regras de produção da gramática de continuação.

Uma gramática de continuação deve ser calculada a partir da gramática da linguagem relacionando-se algumas de suas produções. Esta seleção pode ser realizada escolhendo-se as produções que utilizam o menor número de passos de derivação para a obtenção de uma cadeia terminal, a partir do símbolo não terminal no lado esquerdo da produção.

Uma vez especificada a gramática de continuação o analisador, ao detectar um erro de análise, determinará a continuação usando apenas as leis de formação de sentenças dadas pelas produções da gramática de continuação. Mas, um estado de um analisador LR representa as informações da análise do prefixo de alguma forma sentencial, cujo sufixo não é único. Assim, não é possível obter a continuação de forma determinística, ou seja, existem analisadores LR que não são determinísticamente continuáveis, não tendo, portanto, um autômato de continuação terminal.

Uma alternativa à proposta de Knuth | 1,3 | para analisadores sintáticos LR, que permite a construção simultânea de um autômato de continuação terminal, é o tratamento dos estados do analisador como listas de itens ordenados, em vez de conjuntos de itens. A ordenação é realizada com base na gramática de continuação, de tal forma que o primeiro item de cada estado indique a ação do autômato de continuação terminal, sendo mantido numa tabela auxiliar o símbolo que representa esta ação na tabela de análise.

Um analisador LR, que faça uso das tabelas construídas desta forma, aceita exatamente a linguagem gerada pela gramática. Além disso, o analisador sintático é consistente, deterministicamente continuável e capaz de detectar um erro antes de processar a parte da sentença que o contém.

4. IMPLEMENTAÇÃO

Quando um erro é detectado pelo analisador, um procedimento de tratamento de erros é ativado. Esse procedimento é responsável pela emissão de mensagens de erro, pela indicação das ações realizadas para permitir o prosseguimento da análise e pela retomada desse processo.

Uma implementação baseada no método descrito anteriormente pode ser descrita pelo algo-

ática de con-

inguagem rela-
colhendo-se as
ão de uma ca-
ão.

um erro de
tenças dadas
r LR represen-
sufixo não é
ou seja, exis-
, portanto, um

, que permite
nto dos esta-
itens. A orde-
primeiro item
antido numa ta-

exatamente a
stente, deter-
parte da sen-

e erros é ati-
ela indicação
tomada desse

ta pelo algo-

ritmo abaixo, onde AÇÃO (est, simb) representa a ação de análise no estado est ao visualizar o símbolo simb e AUTCONT (est) contém o símbolo utilizado pelo autômato de continuação no estado est.

```
Salvar a pilha de análise e o estado corrente;
H ← { } ;
w ← ε ;
Repita
    Para todo a ∈ Vt, tal que AÇÃO (estado corrente, a) = ERRO
        Adicione a ao conjunto H;
    Realize a ação de análise especificada em
      AÇÃO (estado corrente, AUTCONT (estado corrente));
    Se a ação realizada foi de empilhamento
        Concatene em w o símbolo AUTCONT (estado corrente)
Até que a análise termine;
Enquanto símbolo visualizado ∉ H
    Leia próximo símbolo de entrada;
Restabeleça a pilha de análise e o estado corrente;
Enquanto AÇÃO (estado corrente, símbolo visualizado) = ERRO
    Realize a ação de análise especificada em
      AÇÃO (estado corrente, AUTCONT (estado corrente));
Emita mensagens sobre o tratamento de erro realizado.
```

Esta implementação pode ser descrita em três fases: a determinação da continuação e do conjunto de âncoras, a eliminação dos símbolos da entrada que não pertençam a este conjunto e, finalmente, o restabelecimento da análise, utilizando a continuação, para que o símbolo visualizado na entrada torne-se válido.

Observamos que o espaço necessário para a implementação deste método de tratamento de erros é pequeno, e que o tempo de análise de programas corretos não é afetado pela sua presença.

Qualquer que seja o erro de análise, o procedimento de tratamento de erros sempre conseguirá retomar a análise, pois sempre existirá na entrada algum símbolo que pertença ao conjunto de âncoras construído na obtenção da continuação.

A idéia básica deste algoritmo de tratamento de erros é ignorar um número mínimo de símbolos da sentença analisada. Este fato, aliado à determinação da continuação pelo menor número possível de transições de análise, imposta pela política de escolha da gramática de continuação, pode provocar o tratamento de erros de forma indesejável. Essa situação ocorre, por exemplo, quando da presença de um símbolo END supérfluo em linguagens tipo ALGOL, o que poderá provocar a interrupção do processo de análise sintática.

As linguagens livres de contexto podem ser encaradas como estruturas de listas hierarquicamente balanceadas. Erros de parentização, quando detectados, são difíceis de solucionar, pois em muitos casos não é possível saber se um parêntese à esquerda foi omitido, ou se um parêntese à direita é supérfluo. Além disso, em linguagens de programação tipo ALGOL, parênteses como BEGIN e END, que delimitam a estrutura sintática de um bloco, delimitam também o escopo de identificadores. Um tratamento incorreto dessa estrutura causará uma avalanche de erros sintáticos e semânticos.

Outro problema com esse algoritmo refere-se às listas de estruturas sintáticas criadas por produções da forma A::=B e A::=A;B, onde o símbolo ; é um separador de itens da lista. Se o símbolo separador ; for omitido, o tratamento será feito, de acordo com a escolha da gramática de continuação, ignorando todos os demais ítens da lista.

Além disso, os identificadores que podem estar presentes em muitas posições sintáticas não representam âncoras seguras para a retomada do processo de análise.

Nesses casos, o algoritmo apresentado nem sempre trata os erros de forma adequada, embora, para uma particular linguagem de programação, possam-se usar essas informações para orientar o tratamento dos erros, de modo a minimizar a produção de erros espúrios.

5. IMPLEMENTAÇÃO INDEPENDENTE DA LINGUAGEM

Apresentamos a seguir um algoritmo que implementa o mecanismo de tratamento de erros proposto por Rohrich | 4,5 | procurando utilizar melhor as informações disponíveis no momento da análise do erro, sem a necessidade de inclusão de um tratamento especial para estruturas e símbolos de uma particular linguagem.

São dois os fatores que nos levam a sugerir um algoritmo alternativo. O primeiro refere-se à determinação da âncora, que originalmente é feita sem considerar quer o número de

passos necessários para que o analisador atinja uma configuração válida, quer o número de símbolos inseridos nesse processo. O segundo fator refere-se à simples eliminação dos símbolos de entrada que não pertencem ao conjunto de âncoras determinado pelo autômato de continuação, sem um tratamento especial para uma grande sequência de símbolos.

Nesta alternativa, o conjunto de âncoras H passa a ser um conjunto de pares ordenados (s,i), onde s é um símbolo da linguagem e i é o número de símbolos cuja inserção é necessária para validar a âncora s, e pode ser obtido na determinação da cadeia de continuação pelo algoritmo seguinte.

```
H ← {};
i ← 0;
w ← ε;
Repita
    Para todo a ∈ Vt, tal que AÇÃO (estado corrente, a) = ERRO
        Adicione (a,i) ao conjunto H;
    Realize a ação de análise especificada em
      AÇÃO (estado corrente, AUTCONT (estado corrente));
    Se a ação realizada foi de empilhamento
        Concatene em w o símbolo AUTCONT (estado corrente);
        i ← i + 1
Até que a análise termine.
```

O tratamento do erro, que originalmente ignorava os símbolos da entrada que não eram âncoras e fazia o restabelecimento da análise utilizando o autômato de continuação até que a âncora fosse válida, passará a evitar ao máximo a eliminação de símbolos da entrada e a encontrar a menor cadeia de inserção, minimizando a distância entre os pontos de erro e de retorno da análise. Esta distância estará limitada àquela determinada pelo autômato de continuação no algoritmo original. A tentativa de fazer o tratamento utilizando os símbolos que não são âncoras da continuação está baseada nos princípios de correções locais propostos por Boullier [2], que representam os caminhos alternativos da análise, limitados ao tamanho da cadeia de símbolos que originalmente seriam ignorados.

```
Cadeia errada ← ε ;
Término do tratamento ← Falso;
Repita
    Se símbolo visualizado ∈ H
        Obtenha o número de símbolos de inserção para a âncora;
        Determine a menor cadeia de inserção;
        Restabeleça a análise;
        Término do tratamento ← Verdadeiro
    Senão
        Concatene em cadeia errada o símbolo visualizado;
        Compr ← |cadeia errada|;
        j ← Compr;
        Enquanto (não término do tratamento e j > 0) faça
            Se for possível inserir compr símbolos antes do j-ésimo
              símbolo da cadeia errada
                Restabelecer a análise;
                Término do tratamento ← Verdadeiro
            Senão
                j ← j - 1;
        Se (não término do tratamento)
            Se for possível trocar compr símbolos
                Restabeleça a análise;
                Término do tratamento ← Verdadeiro
            Senão
                Leia próximo símbolo de entrada
Até que Término do tratamento.
```

Esta modificação na proposta original não altera sua propriedade de corrigir qualquer que seja o erro encontrado na análise. Os casos de inserção de END's supérfluos e não análise dos ítens restantes em estruturas sintáticas sob a forma de listas são automaticamente resolvidos por esta implementação.

Há, porém, que se observar a ocorrência de um acréscimo no tempo de tratamento dos erros.

6. CONCLUSÃO

O método de tratamento de erros proposto por Rohrich foi incorporado ao Gerador de Tabelas de Análise Sintática LR, SLR e LALR desenvolvido no DCEs-UFSCar | 6 | e implementado para testes em um sub-conjunto da linguagem PASCAL. Através de estudos de casos, surgiu esta proposta de uma implementação alternativa que independe da linguagem que está sendo analisada.

7. REFERÊNCIAS

| 1 | AHO, A.V. & ULLMAN, J.D. - "Principles of Compiler Design" - 2 ed. Addison-Wesley, 1977.

| 2 | BOULLIER, P. - "Automatic Syntatic Error Recovery for LR-Parsers". Implementation and Design of Algorithmic Languages, Proc. of the 5'th Annual Conf., Guidel, Roquencourt, IRIA, 348-361, 1977.

| 3 | KNUTH, D.E. - "On the translation of Languages from Left to Right". Information and Control. 8 : 607-639, 1965.

| 4 | ROHRICH, J. - "Automatic Construction of Error Correcting Parsers". Bericht Nr. 8, Institut fur Informatic II, Universitat Karlsruhe, 1978.

| 5 | ROHRICH, J. - "Methods for Automatic Construction of Error Correcting Parsers". Acta Informatica - 13 : 115-139, 1980.

| 6 | ZORZO, S.D. - "Um Gerador Automático de Analisadores Sintáticos LR com Recuperação de Erros". Tese de Mestrado. ICMSC-USP, 1985.